MANUAL

INTRODUTÓRIO E PRÁTICO

DO DESENVOLVEDOR PARA A

WEB MODERNA

Fischer Jônatas Ferreira

© Copyright 2026, Fischer Jônatas Ferreira.

1ª edição

(Publicado em abril de 2026)

Todos os direitos reservados e protegidos pela lei no 9.610, de 19/02/1998. Nenhuma parte deste livro, sem autorização prévia por escrito do detentor dos direitos, poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

Dados Internacionais de Catalogação na Publicação (CIP)

FERREIRA, Fischer Jônatas

MANUAL INTRODUTÓRIO E PRÁTICO DO DESENVOLVEDOR PARA A WEB MODERNA. Fischer Jônatas Ferreira. Pará de Minas, MG: VirtualBooks Editora, Publicação 2026. E-book, formato PDF.

ISBN 978-85-434-1982-4

CDD- 004 Ciência da Computação. HTML e CSS. JavaScript e TypeScript. Fundamentos da Internet. Banco de dados para Web. Banco de dados para Web. Arquitetura de Software. Brasil. Título.

CONSELHO EDITORIAL
Jaime Mendonça \ editor
Assistente editorial \ Geovanna Gravet
Revisora \ Jacqueline Hagop
Preparadora de texto \ Janaína Jaakko Mello
Cao Ypiranga \ designer

Livro publicado pela
VIRTUALBOOKS EDITORA – livros impressos, e-books e audiobooks.
https://www.virtualbooks.com.br
https://virtualbooksonline.com.br/doi
WhatsApp 37991733583 - capasvb@gmail.com

Manual Introdutório e Prático do Desenvolvedor para a Web Moderna

Agradecimentos

Gostaria de expressar minha sincera gratidão a todas as pessoas que contribuíram, direta ou indiretamente, para a realização deste livro.

Agradeço, de forma especial, às alunas carinhosamente apelidadas de "estagiárias do projeto", Antonia Iara A. de Sousa e Iara Costa de Oliveira, bem como à Raquel Luis Duarte, pelo apoio, dedicação e contribuição ao longo do desenvolvimento desta obra.

Sou profundamente grato ao Prof. Carlos Elmano de Alencar e Silva por ter acreditado neste projeto desde as primeiras conversas, oferecendo incentivo e confiança fundamentais para sua concretização.

Registro também meu agradecimento à minha esposa, Michele Faria de Oliveira, cujo apoio, compreensão e incentivo foram essenciais ao longo de toda a trajetória deste trabalho.

Agradeço à Universidade Federal de Itajubá (UNIFEI), minha atual instituição, e à Universidade Federal do Ceará (UFC), onde este projeto teve início, pelo suporte técnico e intelectual, indispensável para a construção desta obra.

Agradeço, ainda, a todos os professores revisores de cada capítulo. As contribuições de vocês trouxeram segurança, qualidade e viabilidade à execução deste projeto.

Por fim, destaco os verdadeiros protagonistas desta obra: os alunos conteudistas, autores de cada capítulo. Sem a participação, a persistência e a dedicação de vocês, este livro não seria possível. Agradeço por acreditarem neste projeto e por investirem seu tempo, conhecimento e esforço na construção deste trabalho coletivo.

E, finalmente, agradeço a você, leitor, pois é para você que este livro foi escrito.

Fischer Jônatas Ferreira
2026

Prefácio

Caro leitor, o livro que agora tens nas mãos é a materialização da poderosa combinação de inspiração, trabalho duro e escrita cooperativa, catalisados pela necessidade. Contextualizo e explico: o prof. Fischer Jônatas Ferreira, atualmente docente da Universidade Federal de Itajubá, junto ao Campus de Itabira, mas que, então, havia recém desembarcado em uma cidade do interior do Ceará, chamada Sobral, após aprovação em Concurso Público de provas e títulos, a fim de compor o corpo docente do Campus mantido naquela cidade pela Universidade Federal do Ceará, junto ao curso de Engenharia de Computação, se vê às voltas com um número crescente de discentes que demandavam, sedentamente, por temas, sugestões bibliográficas, revisões, suporte, apoio, incentivo, enfim, alunos ávidos por orientação acadêmico-profissional.

Tranquilo, simpático e quieto, como só os mineiros sabem ser, estando determinado a contribuir com o Curso nas mais diversas atividades que compõem o metiê docente em uma Instituição Federal de Ensino Superior, e que vão muito além da sala de aula, o prof. Fischer, a fim de não deixar sem orientação nenhum dos estudantes que nele depositava confiança, tem a ideia de desenvolver um projeto de escrita colaborativa de um livro que abordasse diversos temas afetos à área de Desenvolvimento Web e correlatos entre si, cada tema se convertendo em um ou mais capítulos a serem desenvolvidos pelos discentes sob sua orientação.

Bem, é aqui que surge o gancho perfeito para falar desses diversos autores, estudantes de engenharia, não necessariamente aqueles com os melhores índices de rendimento acadêmico ou com os históricos escolares imaculados mas, sem dúvidas, todos com um enorme potencial aliado a muita determinação, vontade de se desenvolver, crescer e vencer. Nesse contexto, o prof. Fischer, me parece, se transformou num poderoso ímã para os aguerridos. E o resultado dessa união entre inspirador e inspirados é o que você encontrará nos capítulos que se seguem. Desfrute. Faça bom proveito. Desejo que, muito além dos relevantes ensinamentos técnicos sobre Desenvolvimento Web e assuntos adjacentes, esta obra inspire também a você.

É uma grande honra ter assistido ao nascimento, ter participado singelamente do desenvolvimento e agora prefaciar a concretização desse magnífico projeto. “Parabéns aos envolvidos, é só o começo!”

Prof. Carlos Elmano de Alencar e Silva
Eng. de Computação - UFC Sobral
30 de março de 2025
Sobral/CE

Sobre o projeto para criação do livro

Introdução

A *World Wide Web* (web) está cada vez mais presente no modo de vida das sociedades modernas, sendo inegáveis a evolução e praticidade proporcionadas pelo desenvolvimento tecnológico. No entanto, o desenvolvimento e a manutenção desse aparato tecnológico, que dá suporte às novas tecnologias e ao amplo acesso às populações, requerem a constante evolução dos sistemas de software, o que, em geral, passa despercebido pelo usuário final, o que não pode ocorrer com os desenvolvedores e estudantes de tecnologia, sob pena de ficarem desatualizados.

Especificamente na área de desenvolvimento web, o caminho é ainda mais turbulento, com uma infinidade de tecnologias, linguagens e ferramentas, tanto existentes quanto novas, que são lançadas com frequência. Além disso, conciliar a teoria e a prática nem sempre é uma tarefa fácil, exigindo que tais profissionais saibam buscar conteúdo de qualidade para se manterem competitivos em seus respectivos campos de atuação.

Neste contexto, este livro é fruto de um projeto de ensino realizado com estudantes de graduação, sendo os alunos os participantes e os principais criadores dos conteúdos de cada capítulo. Desta forma, este trabalho foi escrito a várias mãos, com dedicação de 16 alunos participantes diretamente na escrita dos capítulos, e outras três alunas (cariosamente chamadas de "estagiárias") atuando como mantenedoras dos patrões adotados e na revisão de conteúdo. "De aluno para aluno", essa foi nossa principal frase durante os dois anos de projetos. E, no final da confecção dos capítulos, a palavra que mais li e ouvi foi "confiança"; sempre os alunos me relataram a confiança dada a eles para participarem deste desafiador projeto.

Organização do projeto

Cada capítulo deste livro foi confeccionado por alunos de graduação sob a orientação de professores participantes do projeto, aplicando a metodologia de sala de aula invertida, com o objetivo fundamental de que o conteúdo seja o mais

compreensível possível a outros alunos e associado a exemplos práticos, indo além da mera descrição sobre os assuntos abordados e desenvolvendo nos autores, além das competências técnicas, as habilidades de síntese, criatividade e didática.

O projeto foi conduzido no Campus Sobral da Universidade Federal do Ceará (UFC), junto ao Curso de Engenharia de Computação, de 2023 a 2025 e com a participação de 19 alunos. Durante esse período, foram realizadas reuniões semanais e individuais de alinhamento com cada autor. Além disso, os alunos poderiam utilizar o conhecimento adquirido na elaboração do capítulo do livro para realizar seus trabalhos de conclusão de curso (TCC), o que estimula o engajamento.

Característica do projeto do livro

Criação de um livro com uma abordagem mais prática, baseada em apresentação de exemplos: Como o foco do livro é trabalhar com tecnologias para web, os alunos foram instruídos a constituírem os capítulos da seguinte forma: apresentação da tecnologia, finalidade de uso, breve histórico, visão geral da arquitetura da tecnologia, identificação dos motivos do uso da tecnologia, identificação das situações nas quais ela tem maior aplicabilidade, passo-a-passo de instalação do ferramental necessário, delineamento do público alvo da tecnologia, apresentação do exemplo "Olá Mundo", apresentação dos recursos da tecnologia, apresentação de exemplos práticos simples para trabalhar com os recursos da tecnologia e, finalmente, apresentação de um exemplo de uso da tecnologia por meio de um sistema pequeno e completo, mas que possa apresentar todos os recursos apresentados sobre a tecnologia.

Foco em uma boa didática: O alunos foram motivados a fazerem o capítulo com foco na didática. Desta forma, foi promovida a interação entre os alunos para que os textos fossem compartilhados e lidos por, no mínimo, três outros alunos participantes do projeto, a fim de avaliarem a clareza e domínio manifestados pelo colega autor, fornecendo um *feedback* sincero que permita o seu desenvolvimento como escritor.

Disponibilização dos trechos de código via GitHub: Foi recomendado aos alunos participantes que todos os artefatos de software discutidos estejam disponíveis publicamente no *GitHub*. Isso foi necessário para que todos os exemplos do livro pudessem ser reproduzidos e funcionassem nos computadores dos leitores.

Uso de tecnologias atuais: As tecnologias para desenvolvimento web ficam obsoletas com muita rapidez, representando um desafio para os desenvolvedores e estudantes, tornando a autonomia no aprendizado uma característica primordial para se manter atualizado nessa área. Desta forma, uma restrição importante imposta aos alunos participantes do projeto foi que as tecnologias escolhidas para serem descritas nos capítulos deveriam ser atuais, consolidadas e bem avaliadas nos ranques do *Stack Overflow* e *GitHub*.

Papéis dos participantes do projeto

Perfil dos alunos participantes: Para participar do projeto, os interessados deveriam já ter cursado com sucesso a disciplina de Tecnologias Web, a disciplina de Engenharia de Software, ter feito parte de alguma monitoria e estar matriculados na atividade de Seminário de Monografia sob a orientação de algum professor participante do projeto, restringindo o projeto aos alunos que estavam próximos de colar grau.

Alunos Revisores: Os alunos participantes do projeto foram convidados pelos professores orientadores a ajudar na revisão de capítulos de outros alunos. Essa interação foi fundamental para a evolução dos capítulos revisados, bem como para a validação e replicação dos exemplos discutidos em cada capítulo. Embora essa tenha sido a etapa prevista no projeto para a interação entre os participantes, percebeu-se que os autores mantiveram interação constante em todas as demais etapas.

Estagiárias do projeto: O projeto contou com a participação fundamental de três estagiárias, cujas atribuições foram: manter a comunicação entre os membros do projeto, criar um documento para padronização do estilo e formato do texto em *Latex*, leitura dos textos produzidos, reprodução dos exemplos apresentados nos capítulos, execução de ferramentas para detecção de plágio, identificação de problemas com o texto, tais como: ortografia, concordância, norma culta da linguagem, coesão e coerência do texto, e falta de padronização com imagens, tabelas e listagem de código fonte.

Professores revisores de conteúdo: Os professores revisores participaram na orientação dos alunos conteudistas. Como os alunos participantes do projeto estavam matriculados na atividade de Seminário de Monografia, os capítulos elaborados para o livro foram apresentados como Memorial de Monografia e, portanto, submetidos à revisão e aprovação de uma comissão composta por dois professores do Curso de Engenharia de Computação.

Professores orientadores dos alunos participantes: Os professores tinham a missão de orientar os alunos em relação aos conteúdos criados, respeitando a autonomia do aluno autor, dando liberdade na escolha dos exemplos apresentados, das referências utilizadas, da forma de apresentação e outras características que tornaram o livro uma coleção de capítulos escritos a várias mãos. A orientação contou com reuniões individuais semanais de alinhamento e discussão sobre a construção e o andamento de cada capítulo. Adicionalmente, professores de outras instituições foram convidados a coorientar o desenvolvimento dos capítulos do livro, o que resultou na colaboração de três professores das Universidades Federais de Lavras e Ouro Preto com o projeto.

Professor idealizador do projeto. O projeto contou com a participação do professor que cuidou da organização geral do projeto. Esse professor teve as seguintes atribuições: discussão dos temas de cada capítulo para compor o livro, revisão dos capítulos, realização de reuniões periódicas com os alunos participantes do projeto (autores e estagiárias), reuniões com os professores orientadores, promoção

da interação entre os alunos autores e agregação de professores orientadores para participação no projeto. O professor idealizador também foi revisor de conteúdos e orientador de alunos.

Como foi desenvolvido o projeto

Ao entrar no projeto, o aluno autor recebia o fluxograma apresentado na Figura 1, no qual estão esquematizadas as cinco etapas para a criação dos capítulos do livro.

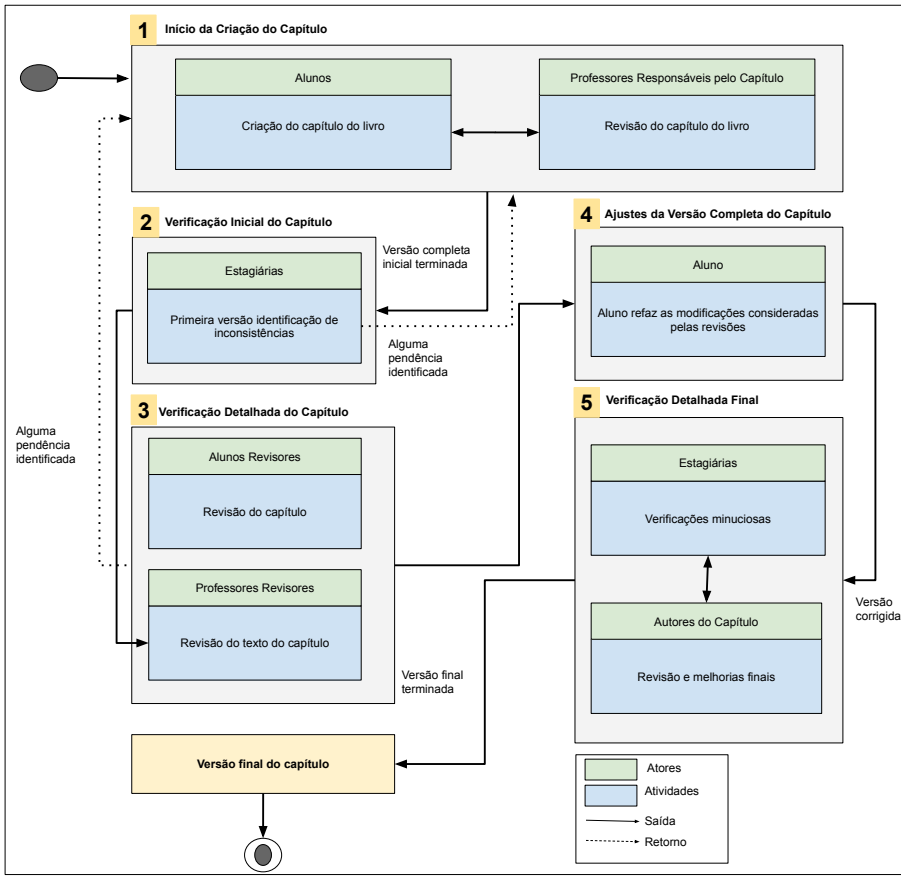


Figura 1: Etapas do projeto

1: Início da criação de conteúdo. Nessa etapa inicial o aluno autor, juntamente com o professor responsável, seleciona o tema do capítulo na lista de temas de interesse para compor o livro. Em seguida, passa-se à revisão da literatura

relacionada ao tema escolhido. Então, de forma interativa e incremental, a versão preliminar do capítulo é escrita pelo aluno e revisada pelo orientador.

2: Verificação inicial do capítulo: As estagiárias verificam a versão preliminar do capítulo. As etapas 1 e 2 duram, em média, seis meses.

3: Verificação mútua do capítulo: Nessa etapa os alunos autores revisam os textos uns dos outros. O capítulo também é revisado pelos professores revisores. Essa etapa exigiu, em média, duas semanas de trabalho.

4: Ajustes da Versão completa do capítulo: Nessa etapa o aluno conteudista aprecia as sugestões feitas nas etapas de revisão, tendo a oportunidade de perceber as inconsistências no texto e nos exemplos fornecidos, as inclusões e/ou exclusões de conteúdo necessárias (atualização de tabelas, figuras e listagem de código), bem como qualquer outra observação feita pelos revisores. Essa etapa foi feita, em média, em um mês.

5: Verificação detalhada do capítulo: Nessa etapa as estagiárias reverificam, com mais rigor, o texto corrigido pelo aluno autor, observando a padronização do estilo e a adequação do conteúdo às observações feitas pelos revisores. Ainda nessa etapa, o rol de autores é definido, contemplando os indivíduos responsáveis pelo resultado final, inclusive os professores revisores, proporcionando forte adesão de todos envolvidos.

Finalmente, alcançada a anuência de todos os agora autores, chega-se à versão final do capítulo, o qual é incluído no livro pelo professor idealizador do projeto.

Autores dos capítulos

Autores dos capítulos

Relação dos capítulos e respectivos autores

Capítulo 1
HTML e CSS
Jadson Faustino Mororó (UFC)
Michele Faria de Oliveira (UFC)
Fischer Ferreira (UNIFEI)

Capítulo 2
JavaScript e TypeScript
Jadson Faustino Mororó (UFC)
Erick Aguiar Donato (UFC)
Iális Cavalcante de Paula Júnior (UFC)
Fischer Ferreira (UNIFEI)

Capítulo 3
Fundamentos da Internet
Victória Tomé Oliveira (UFC)
Sayonara Santos Araújo (UFC)
Wendley Souza da Silva (UFC)
Fischer Ferreira (UNIFEI)

Capítulo 4
Ruby on Rails, Laravel e Django
Vitória Nobre Lima (UFC)
Carlos Elmano de Alencar e Silva (UFC)
Evilasio Costa Junior (UFC)
Fischer Ferreira (UNIFEI)

Capítulo 5
Banco de dados para Web
Francisca Jannielly Garcia da Costa (UFC)
Carlos Elmano de Alencar e Silva (UFC)
Fernando Rodrigues de Almeida Júnior (UFC)
Fischer Ferreira (UNIFEI)

Capítulo 6
Back-end
Klayver Ximenes Carmo (UFC)
Antonio Josefran de Oliveira Bastos (UFC)
Fischer Ferreira (UNIFEI)

Capítulo 7
Interface de programação de aplicações - API
Samyle de Sousa Araújo (UFC)
Fischer Ferreira (UNIFEI)

Capítulo 8
Front-end
Marcos Vinícius Andrade de Sousa (UFC)
Evilasio Costa Junior (UFC)
Fischer Ferreira (UNIFEI)

Capítulo 9
Engenharia de Requisitos para Web
Joana Sthefanny Gomes Costa dos Santos (UFC)
Michele Faria de Oliveira (UFC)
Evilasio Costa Junior (UFC)
Fischer Ferreira (UNIFEI)

Capítulo 10
Web Semântica
Alanna Maria Machado Alves Paiva (UFC)
Fischer Ferreira (UNIFEI)

Capítulo 11
Teste de Software para Web
Raíssa Ellen de Sousa (UFC)
Cleiton Tavares (PUC-MG)
Euler Horta Marinho (UFOP)
Fischer Ferreira (UNIFEI)

Capítulo 12
Visão geral sobre DevOps
Ana Cristina Pousa Machado (UFOP)
Márcio Bruno Loiola Gomes (UFC)
Marco Antônio Alves de Almeida (UFOP)
Igor Muzetti Pereira (UFOP)
Fischer Ferreira (UNIFEI)

Capítulo 13
Arquitetura de Software
William Bruno Sales de Paula Lima (UFC)
Gustavo Vale (UNILAVRAS)
Fischer Ferreira (UNIFEI)

Capítulo 14
Ferramentas para Web
Izaias Machado Pessoa Neto (UFC)
Fischer Ferreira (UNIFEI)

Capítulo 15
As Principais Hard Skills de um Desenvolvedor para Web Atual
Maxela Martins Pontes (UFC)
Johnatan Alves Oliveira (UFLA)
Jermana Lopes de Moraes (UFC)
Fischer Ferreira (UNIFEI)

Apresentação

Atualmente, as ferramentas para a web estão profundamente integradas em nossa rotina, facilitando atividades que vão desde a mobilidade urbana com sistemas de GPS até a comunicação por meio de áudios, videochamadas e redes sociais. A internet permanece como elemento central, servindo de base para a criação de quase todo o software desenvolvido recentemente, seja por meio de aplicações nativas ou adaptadas para a web. Os usuários já estão familiarizados com o uso da web, tendo aprendido a navegar por links, enviar formulários, explorar páginas, utilizar caixas de busca e aplicar filtros para refinar informações. Essa familiaridade torna a experiência com aplicações web mais intuitiva, já que os elementos e mecanismos presentes nesses sistemas refletem o que o usuário encontra diariamente ao acessar a internet.

O crescimento exponencial das tecnologias web gera, naturalmente, uma demanda crescente por desenvolvedores qualificados. No entanto, o surgimento constante de novas ferramentas, *frameworks* e linguagens pode ser intimidante para quem está iniciando nesse universo. Embora exista uma vasta documentação com exemplos de uso, falta ao principiante uma orientação clara sobre quais tecnologias são essenciais, qual a função de cada uma e como elas se integram no desenvolvimento de aplicações web. É comum que o novo programador se sinta perdido diante da infinidade de informações disponíveis, sem saber por onde começar ou quais áreas priorizar.

O desenvolvimento web envolve a criação de aplicações que abrangem tanto o front-end, responsável pela interface e pela interação com o usuário, quanto o back-end, onde se implementam as regras de negócio e ocorre a comunicação com bancos de dados e com a infraestrutura necessária para suportar o sistema. Esses componentes, embora distintos, dependem uns dos outros para formar soluções completas e funcionais. Compreender essa integração é fundamental para qualquer iniciante, pois permite visualizar o fluxo completo de uma aplicação e entender como cada camada contribui para o desempenho e a robustez do sistema.

Diante desse cenário, a criação de um material didático voltado a novos programadores é essencial. Um livro introdutório que ofereça uma visão geral das tecnologias modernas da web pode ser o guia necessário para que o iniciante

aprenda, de maneira prática e organizada, a importância de cada componente—seja a interface de usuário, a lógica de aplicação ou a gestão de dados.

Em resumo, embora a variedade e a constante evolução das tecnologias para web possam parecer desafios intimidadoras no início, uma orientação estruturada e acessível transforma esse cenário em uma oportunidade rica de aprendizado. Ao compreender a integração entre front-end, back-end, bancos de dados e infraestrutura, o iniciante passa a ter uma visão holística do desenvolvimento web, capacitando-o a criar soluções inovadoras e eficientes que atendam às necessidades da sociedade atual.

Neste contexto, este livro foi concebido para introduzir o desenvolvedor iniciante às principais tecnologias utilizadas na web, apresentando, de forma prática, como elas se aplicam na criação de aplicações robustas. Cada capítulo não apenas expõe os conceitos fundamentais e a descrição das tecnologias, mas também oferece um primeiro contato prático por meio de exemplos didáticos, que evoluem até a construção de um exemplo mais completo e integrado. Essa abordagem gradual permite que o estudante amplie seus conhecimentos teóricos e práticos, consolidando o aprendizado de maneira progressiva.

A estrutura do livro foi pensada para facilitar a consulta, já que cada capítulo foi elaborado de forma independente, permitindo que o leitor escolha a ordem de leitura conforme sua necessidade ou interesse. Essa organização modular possibilita que o estudante foque nas áreas que mais deseja aprofundar sem comprometer a compreensão global do desenvolvimento web. Vale ressaltar que cada capítulo foi escrito por um estudante de graduação, que se dedicou a estudar o tema, desenvolver exemplos práticos e, com o apoio de professores, estruturou o conteúdo de maneira clara e objetiva.

Destinado a indivíduos que estão dando os primeiros passos na programação de computadores e no desenvolvimento para web, este livro serve tanto para alunos de cursos de computação quanto para aqueles que buscam se atualizar ou ingressar nesse campo, mesmo sem formação técnica prévia. Com uma linguagem acessível e sem rodeios, a obra apresenta os fundamentos necessários para que o iniciante entenda onde e como as tecnologias se integram, fornecendo uma base sólida que o ajudará a navegar e a se desenvolver no vasto universo da web.

Organização do Livro

- Capítulo 1 - **Introdução ao HTML e CSS**: Este capítulo é dedicado à construção das bases de qualquer página ou aplicação web, focando na estruturação e apresentação de conteúdos. O leitor aprenderá sobre a marcação semântica em HTML e como utilizar CSS para estilizar elementos, criando interfaces visualmente agradáveis e funcionais. São apresentados exemplos práticos que demonstram a aplicação dos conceitos, facilitando o entendimento dos elementos e atributos essenciais. Este capítulo é um ponto de partida

indispensável para quem está começando a programar para a web e quer criar uma página da web.

- Capítulo 2 - **Introdução ao *JavaScript* e *TypeScript***: Neste capítulo, o desenvolvimento dinâmico e interativo de aplicações web é abordado por meio do *JavaScript*, a linguagem central do *front-end*. Além disso, o capítulo introduz o *TypeScript* como uma evolução do *JavaScript*, destacando como a tipagem estática e os recursos avançados podem contribuir para a criação de projetos mais robustos e escaláveis. São apresentados exemplos práticos e comparações que elucidam as vantagens e as situações em que cada linguagem se destaca. Por meio de exemplos nas duas linguagens *JavaScript* e *TypeScript* você perceberá como essas linguagem se complementam e perceberá que o *JavaScript* é o coração da web.

- Capítulo 3 - **Principais fundamentos da Internet**: Neste capítulo, são explorados os conceitos básicos que sustentam a comunicação digital e o funcionamento da rede global. São abordados as arquiteturas de rede, os protocolos de comunicação (como TCP/IP e HTTP) e os principais conceitos que diferenciam a Internet da Web. A apresentação clara e direta permite ao leitor entender como os dispositivos se conectam e trocam informações, preparando o terreno para a compreensão dos demais tópicos do livro. Com uma linguagem acessível, o capítulo oferece uma base teórica básica que é fundamental para qualquer profissional que deseje atuar no desenvolvimento web.

- Capítulo 4 - **Ruby on Rails, Laravel e Django**: Este capítulo apresta três *frameworks* utilizados no desenvolvimento web, que possibilitem que o *front-end* e o *back-end* de um sistema Web possam ser implementados juntos. Você poderá acompanhar a descrição dos *frameworks* da instalação até suas utilizações por meio de exemplos práticos. Ainda, é apresentada um comparação entre essas tecnologias que podem nortear escolhas futuras.

- Capítulo 5 - **Banco de Dados para web**: O capítulo sobre Banco de Dados para web aborda os princípios e práticas de armazenamento e manipulação de dados em aplicações para web. Discutem-se os diferentes tipos de bancos de dados, como os relacionais e não relacionais, e os critérios para a escolha de um sistema de gerenciamento de banco de dados adequado. Com exemplos práticos, você aprenderá a utilizar os principais bancos de dados para a web e poderá fazer suas próprias bases de dados.

- Capítulo 6 - **Introdução ao *Back-end* de aplicações para web**: Focado na lógica e processamento que ocorrem do lado do servidor, este capítulo explora as tecnologias e técnicas necessárias para construir a parte "invisível"das aplicações web. São apresentados alguns dos principais *frameworks* para*back-end* ((`Node.js`, `Django DRF` e `ASP.NET`)). Com uma abordagem prática e

clara, o capítulo mostra como criar o *back-end* de aplicações Web, além de apresentar exemplos práticos que você pode acompanhar e reproduzir junto com o capítulo.

- Capítulo 7 - **Interface de programação de aplicações (API)**. Neste capítulo, são apresentados os conceitos fundamentais relacionados às APIs, destacando seu papel na comunicação entre sistemas e aplicações web. São abordadas as principais arquiteturas e estruturas de requisições e respostas, incluindo métodos HTTP, endpoints e formatos de dados. Além disso, o capítulo explora diferentes abordagens de desenvolvimento de APIs, como REST, GraphQL e gRPC, evidenciando suas características, vantagens e limitações. Por meio de exemplos práticos, o leitor compreenderá como implementar e consumir APIs, consolidando conhecimentos essenciais para o desenvolvimento de aplicações modernas e integradas.

- Capítulo 8 - **Uma visão geral sobre*Front-end*: React.js e Angular**: Neste capítulo, o foco está na criação de interfaces de usuário interativas e intuitivas, que representam a face visível das aplicações web. São abordadas as tecnologias e ferramentas que permitem a criação de layouts responsivos e modernos, bem como a implementação de elementos que aprimoram a experiência do usuário. Você será introduzido a conceitos fundamentais, como a renderização dinâmica e o uso de *frameworks* e bibliotecas específicas para o *front-end*. Os *frameworks*: React e Angular são discutidos neste capítulo.

- Capítulo 9 - **Engenharia de requisitos para web**: Neste capítulo, são exploradas as estratégias e ferramentas utilizadas para validar e garantir a qualidade das aplicações web. O leitor aprenderá sobre a importância dos testes, desde abordagens manuais até a automação de testes, e como essas práticas podem identificar e corrigir problemas antes da entrega do produto. São apresentados exemplos que demonstram a aplicação de testes em diferentes níveis, evidenciando a relevância de um processo de validação contínua. A abordagem clara e prática permite que o iniciante compreenda como integrar testes eficazes ao ciclo de desenvolvimento.

- Capítulo 10 - **Web Semântica**: Neste capítulo, apresenta-se a evolução da Web, desde seus fundamentos até o surgimento da Web Semântica, destacando a necessidade de tornar os dados compreensíveis não apenas para humanos, mas também para máquinas. São explorados conceitos essenciais, como metadados, ontologias, agentes inteligentes e *Linked Data*, evidenciando como essas estruturas permitem a organização, a integração e a interpretação automática de informações. O capítulo também aborda as principais tecnologias e padrões utilizados nesse contexto, como RDF, OWL, SPARQL e JSON-LD, além de apresentar aplicações práticas em áreas como assistentes virtuais, saúde e educação. Com uma abordagem clara e contextualizada, você compreen-

derá como a Web Semântica contribui para a construção de uma Web mais inteligente, interoperável e orientada ao significado.

- Capítulo 11 - **Teste de software para web**: Neste capítulo, são exploradas as estratégias e ferramentas utilizadas para validar e garantir a qualidade de aplicações web. Você aprenderá sobre a importância da automação de testes por meio de exemplos práticos. Para isso serão demonstradas duas ferramentas para criação de testes automatizados em aplicações web, o `Selenium` e `Cypress`. Ainda, será discutido e demonstrado a criação de uma suíte de teste automatizados para uma aplicação uma sistema de banco online.

- Capítulo 12 - **Visão Geral sobre o DEVOPS**: O capítulo sobre DevOps oferece uma introdução aos conceitos e práticas que unificam as equipes de desenvolvimento e operações, promovendo uma cultura de colaboração e integração contínua. São discutidas as metodologias ágeis, os pipelines de integração e entrega contínua (CI/CD) e as ferramentas que facilitam a automação dos processos. O leitor compreenderá como o DevOps contribui para a melhoria da qualidade e da velocidade de entrega de produtos, reduzindo barreiras e promovendo uma comunicação eficaz entre os times. A apresentação é didática e objetiva, evidenciando os benefícios dessa abordagem para o desenvolvimento web moderno.

- Capítulo 13 - **Arquitetura de software**: Neste capítulo, são discutidas as principais arquiteturas de software utilizadas no desenvolvimento de aplicações Web: monolítica, cliente-servidor, orientada a mensagens e publicador-consumidor. O capítulo apresenta um levantamento teórico sobre as arquiteturas e exemplos práticos de sua aplicação a problemas do mercado de trabalho. Assim, a partir destes exemplos, são apresentadas as diferenças de utilização das arquiteturas por meio de exemplos práticos.

- Capítulo 14 - **Ferramentas para web**: Este capítulo oferece uma visão prática de algumas ferramentas que apoiam o desenvolvimento web. Você será apresentado, por meio de exemplos práticos, às ferramentas aplicáveis a diversos tipos de projetos, independentemente do domínio do que está sendo desenvolvido. Essas ferramentas atendem a demandas comuns na implementação de aplicações Web. Ao utilizar essas ferramentas, você poderá lidar melhor com versionamento de código, comunicação, colaboração, hospedagem de código e implantação de aplicações, abrangendo desde sistemas de versionamento e ambientes de integração até plataformas de deploy e colaboração. São discutidas as características e funcionalidades de cada tipo de ferramenta, bem como orientações para sua escolha de acordo com as necessidades do projeto. O leitor aprenderá como essas ferramentas podem agilizar o processo de desenvolvimento e reduzir a complexidade na gestão de projetos. A exposição prática e direta permitirá a você identificar quais soluções são mais

adequadas para cada etapa do desenvolvimento web, facilitando a adoção de uma abordagem mais organizada e eficiente.

- Capítulo 15 - **As principais *Hard Skills* de um desenvolvedor para Web**: Este capítulo apresenta as competências técnicas essenciais que formam a base do desenvolvimento web moderno. Nele, o leitor encontrará uma discussão sobre as linguagens de programação, *frameworks* e ferramentas que o mercado valoriza, além de boas práticas de codificação e resolução de problemas. A abordagem é prática e objetiva, destacando quais habilidades técnicas são prioritárias para quem deseja se inserir nesse campo e se manter atualizado diante da constante evolução tecnológica. Dessa forma, o capítulo serve como um guia para que o desenvolvedor iniciante possa identificar e desenvolver as *hard skills* necessárias para os primeiros passos na carreira de desenvolvedor para a web.

Manual Introdutório e Prático do Desenvolvedor para a Web Moderna

Fischer Ferreira

Lista de Figuras

1 Etapas do projeto x

1.1 Exemplo do HTML sendo exibido no navegador 35
1.2 Ilustração do *Box Model* em HTML/CSS. 64
1.3 Visualização resultante do código HTML/CSS da Listagem 1.31. 67
1.4 Visualização resultante do código HTML/CSS da Listagem 1.32. 69
1.5 Visual da página do projeto exemplo prático desenvolvido na seção. 80

2.1 Código JavaScript sendo interpretado pelo navegador Web 83
2.2 Exemplo de Compilação de JavaScript para TypeScript 87

3.1 Protocolo HTTP. 143
3.2 Comparação entre múltiplas solicitações HTTP/2 x QUIC 148
3.3 Comparação entre números de viagens de ida e volta. 149
3.4 Comparação entre os serviços de Nuvem IaaS, CaaS, PaaS, FaaS e SaaS. 156
3.5 Comparação entre (a) máquina virtual e (b) contêiner. 158

4.1 Arquitetura `Django`. Adaptado de (RAMOS, 2023) 163
4.2 *Object-Relational Mapper* (ORM) 164
4.3 Fluxo de trabalho das migrações 165
4.4 Fluxo de processamento de requisições de URLs. Adaptado de (ANKUSH_953, 2016) 168
4.5 Diagrama de *Middleware*. Adaptado de (BOUCHER, 2016) . . . 168
4.6 Fragmento de tela inicial do site de administração 172
4.7 Arquivos de Inicialização 173
4.8 Tela inicial de uma aplicação `Django` 174
4.9 Fragmento de tela de *login* do site de administração 177
4.10 Tela do projeto finalizado em `Django` 181
4.11 Arquitetura `Laravel`. Adaptado de (PATEL, 2023a). 184
4.12 *Middleware* no `Laravel`. Adaptado de (STAUFFER, 2019). . . . 187
4.13 Tela inicial de uma aplicação `Laravel` 192
4.14 Fragmento da Página inicial da aplicação 200
4.15 Fragmento da Página inicial da aplicação estilizada 200

4.16 Arquitetura `Ruby on Rails`. Adaptado de (PATEL, 2023b) . . . 203
4.17 Página Inicial do `Ruby on Rails` . . . 210
4.18 Fragmento da página inicial do projeto após a adição da rota *root* 212
4.19 Fragmento da página inicial do projeto após a adição do *link* de criação de novo *post* . . . 221
4.20 Fragmento da página de visualização de um *post* . . . 224
4.21 Fragmento da página de visualização de um *post* após adição do *link* de exclusão . . . 226
4.22 Fragmento da página inicial estilizada . . . 227
4.23 Tela "Cadastrar atividade" . . . 228

5.1 Comunicação de aplicações web . . . 241
5.2 Fluxo dos níveis de modelagem . . . 243
5.3 Representação do diagrama ER da relação de venda. . . . 244
5.4 Operações de conjuntos . . . 245
5.5 Ilustração da arquitetura cliente-servidor . . . 254
5.6 Página inicial para download do `MySQL` . . . 256
5.7 Página de login *Oracle account* . . . 257
5.8 Tela de escolha do tipo de instalação do `MySQL` . . . 257
5.9 Tela de escolha dos utilitários do `MySQL` . . . 258
5.10 Tela de configurações de rede do `MySQL` . . . 258
5.11 Tela de configurações de autenticação do `MySQL` . . . 259
5.12 Tela de configuração de contas e usuários do `MySQL` . . . 259
5.13 Tela inicial `MySQL` Workbench . . . 260
5.14 Tela de configuração do servidor . . . 260
5.15 Tela inicial `MySQL` Workbench com o servidor criado . . . 261
5.16 Tela de gerenciamento de bancos de dados do `MySQL` Workbench . 261
5.17 Comandos SQL . . . 262
5.18 Página inicial `MongoDB` . . . 271
5.19 Página inicial do `MongoDB` com ênfase no menu . . . 271
5.20 Página de download do `MongoDB Server` . . . 272
5.21 Primeiro passo da instalação do `MongoDB` . . . 272
5.22 Tela de termos e licenças do `MongoDB` . . . 273
5.23 Tela de escolha do tipo de instalação do `MongoDB Server` . . . 273
5.24 Tela de configuração de rede e usuário para o `MongoDB Server` . 274
5.25 Tela de instalação do `MongoDB Compass` . . . 274
5.26 Tela inicial do `MongoDB Compass` . . . 275
5.27 Tela de gerenciamento dos bancos de dados no `MongoDB Compass` 275
5.28 Tela de criação de um banco de dados no `MongoDB Compass` . . . 276
5.29 Tela de gerenciamento do banco de dados criado . . . 276
5.30 Microsoft Store - Ubuntu . . . 282
5.31 Terminal do Ubuntu . . . 282
5.32 `Redis`-Cli . . . 283
5.33 Comunicação usando `Entity Framework` . . . 288

6.1 Modelo arquitetural de uma *Application Programming Interface* (API) *Representational State Transfer* (REST) 291
6.2 Página de *download* do *Workbench* `MySQL` 292
6.3 Página de configuração do *Workbench* `MySQL` 293
6.4 Página inicial de *download* da ferramenta `Insomnia` 294
6.5 Painel inicial da ferramenta `Insomnia` 295
6.6 Interface da ferramenta `Insomnia` 295
6.7 Funcionamento do motor V8 297
6.8 Diagrama de operações do *loop* de eventos 298
6.9 Diagrama de operações do *loop* de eventos 299
6.10 Funcionamento do motor V8 304
6.11 Utilização do método GET (API Node.js) 305
6.12 Utilização do método POST (API Node.js) 306
6.13 Utilização do método PUT (API Node.js) 307
6.14 Utilização do método DELETE (API Node.js) 307
6.15 Representação modelo *Model View Template* (MVT) 309
6.16 Estrutura de arquivos inicial 312
6.17 Utilização do método GET (API Django REST) 317
6.18 Utilização do método POST (API Django REST) 317
6.19 Utilização do método PUT (API Django REST) 318
6.20 Utilização do método DELETE (API Django REST) 318
6.21 Painel administrativo do Django 319
6.22 Disponibilização e consumo de pacotes por meio do NuGet 320
6.23 Página inicial do instalador do *Software Development Kit* (SDK) 322
6.24 Instalação do ambiente ASP.NET 323
6.25 Árvore de arquivos inicial 323
6.26 Geração dos controladores com *Scaffold* 327
6.27 Configuração dos controladores 327
6.28 Exemplo método GET (API ASP.NET Core) 329
6.29 Exemplo método POST (API ASP.NET Core) 330
6.30 Exemplo método PUT (API ASP.NET Core) 332
6.31 Exemplo método DELETE (API ASP.NET Core) 333

7.1 Requisição para a API Google Books 340
7.2 Arquitetura Cliente-Servidor 342
7.3 Componentes de uma requisição 343
7.4 Exemplo do corpo de uma requisição POST 345
7.5 Exemplo de requisição com envio de parâmetro 345
7.6 Cabeçalhos de uma requisição 345
7.7 Resposta enviada após uma requisição de leitura 346
7.8 Cabeçalhos de uma resposta 348
7.9 Requisição realizada na API Google Books utilizando o Insomnia 352
7.10 Requisição realizada em uma API REST usando o Insomnia . . . 355
7.11 Resposta retornada por uma API REST 355

7.12 Representação visual do banco de dados 373
7.13 Exemplos dos dados fictícios gerados pela biblioteca Faker para a tabela Endereço 374

8.1 Tópicos discutidos neste capítulo 387
8.2 Visualização simplificada da comunicação web 388

9.1 Fluxo de Desenvolvimento Web 444
9.2 O ciclo de vida dos requisitos 447
9.3 Técnicas de Elicitação de Requisitos 449
9.4 Exemplo de Verificação e Validação de Requisitos 451
9.5 Especificação e Modelagem de Requisitos 453
9.6 Mapa mental de Modelagem de Requisitos 455
9.7 Modelo de histórias de usuários 456
9.8 Exemplo de histórias de usuário 458
9.9 Diagrama de caso de uso de um site 459
9.10 Diagrama de caso de uso de um site 461
9.11 Exemplos de Requisitos Não Funcionais em uma aplicação web . 462
9.12 Matriz GUT 468
9.13 Matriz GUT 469
9.14 Matriz BASICO 470
9.15 JIRA 472
9.16 Helix RM 473
9.17 Visure 474

10.1 Estrutura do HTML5 479
10.2 Assistente de voz em ambiente de saúde personalizado 482
10.3 Web Semântica em Assistentes Virtuais 483
10.4 Representação gráfica de uma ontologia voltada para humanos . . 489
10.5 Tabela de ontologia para o termo “Curso de Programação Avançado” 490
10.6 Representação de agentes inteligentes com base em grafos de conhecimento. 493
10.7 Agente como “Camada Inteligente” sobre web services 494
10.8 Web Service como fonte de dados para um agente 495
10.9 Exemplo de uso de ontologia na oferta de serviços na Web 497
10.10 Funcionamento do Linked Data 499
10.11 Estrutura RDF para representação de relações entre conceitos . . 502
10.12 Analogia de representação do SHACL 512

11.1 Diferença entre os conceitos de defeito, erro e falha 520
11.2 Interface do *plugin Selenium* IDE para o Chrome. 525
11.3 Arquitetura do Selenium RC 527
11.4 Arquitetura do Selenium Grid 529
11.5 Interface da tela de login da aplicação SauceDemo 531

11.6 Interface da tela de produtos da aplicação SauceDemo 532
11.7 Resultado do caso de teste com Selenium 536
11.8 Arquitetura do Cypress 537
11.9 Interface da tela inicial do Cypress 539
11.10 Interface da tela de configuração do Cypress 540
11.11 Estrutura de arquivos do projeto em Cypress 541
11.12 Resultado do caso de teste com Cypress 542
11.13 Tela inicial da aplicação web *Parabank* 544
11.14 Modelagem dos casos de teste 545
11.15 Menus da tela inicial da aplicação Web Parabank 551
11.16 Menus da tela principal da aplicação Web Parabank 552
11.17 Relatório de testes via terminal utilizando o Selenium 558
11.18 Relatório de testes via terminal utilizando o Cypress 558

12.1 A cadeia de processos de DevOps 563
12.2 Visão geral de um quadro no Trello 565
12.3 Exemplo de cartão no Trello 566
12.4 Outros elementos de um cartão no Trello 566
12.5 Visão geral do *Board Menu* 567
12.6 Quadro para etapas do Pipeline CI/CD 567
12.7 Inicialização de um repositório Git local 568
12.8 Interface de um repositório no GitHub 569
12.9 Execução do comando `git clone` 569
12.10 Verificação do repositório remoto associado 570
12.11 *Status* do repositório antes das modificações 570
12.12 Preparação das modificações e criação do *commit* 571
12.13 Envio de alterações para o repositório remoto 571
12.14 Tela inicial de pull requests 572
12.15 Pull requests no repositório Next.js 572
12.16 Criação de um pull request 573
12.17 Visualização das modificações propostas 573
12.18 *Merge* do pull request 574
12.19 Pull request finalizado e integrado à *main* 574
12.20 Página inicial do Docker Desktop 575
12.21 Execução de teste com `docker run hello-world` 575
12.22 Documentação oficial da sintaxe de um Dockerfile 576
12.23 Página inicial do Docker Hub 576
12.24 Configurações do usuário. 578
12.25 Configurações do desenvolvedor. 578
12.26 Tokens de acesso pessoal. 578
12.27 Tokens (clássico). 579
12.28 Gerar novo token 579
12.29 Escopos do Token. 580
12.30 Gerando token. 580

12.31 Fork. 580
12.32 Criando um novo fork. 581
12.33 Criando arquivo experimento-ci-cd.yml. 583
12.34 Configurações do repositório. 588
12.35 Actions. 588
12.36 Configurando postgres user. 589
12.37 Configurando postgres password. 590
12.38 Arquivo update-user-spec.ts. 591
12.39 Mudança no código. 592
12.40 Criar Pull Request. 593
12.41 Visualização da falha no pipeline. 593
12.42 Criar Pull Request com correção novamente 594
12.43 Pipeline aceito. 595

13.1 Exemplo de arquitetura cliente-servidor 600
13.2 Exemplo de aplicativo de e-commerce 601
13.3 microsserviços do aplicativo de *e-commerce* 602
13.4 Conjuntos de tecnologias de microsserviços 603
13.5 Chamada via API 604
13.6 Comunicação assíncrona via corretor de mensagens de microsserviços 605
13.7 Estrutura mono repositório de microsserviços 606
13.8 Estrutura das pastas de microsserviços no mono repositório . . . 606
13.9 Estrutura poli repositório de microsserviços 608
13.10 Estrutura poli repositório de microsserviços 609
13.11 Diagrama ilustrativo dos microsserviços das funcionalidades do sistema da livraria 610
13.12 Diagrama de modelos de dados do sistema de livraria 611
13.13 Resultado a execução da ativação do serviço de usuários 613
13.14 Resultado a execução da ativação do serviço de pedidos 613
13.15 Resultado a execução da aplicação web interagindo com microsserviços 614
13.16 Diagrama ilustrativo da arquitetura orientada a mensagens . . . 614
13.17 Arquitetura orientada a mensagens com múltiplos servidores . . . 615
13.18 Resultado do recebimento dos dados de empréstimos no servidor 621
13.19 Diagrama ilustrativo de uma arquitetura publicador-consumidor . 622
13.20 Diagrama ilustrativo de tópicos na arquitetura publicador-consumidor 623
13.21 Execução do serviço publicador 626
13.22 Execução do serviço assinante 626
13.23 Sistema de *streaming* 627

14.1 Propagação das modificações 644
14.2 Seções de um projeto Git 644
14.3 Ciclo de vida dos arquivos 645
14.4 Ilustração de um histórico de *commits* linear 646

14.5 Ilustração de um histórico de *commits* com *branches* e *merges* . . 646
14.6 Inicialização do projeto Git 647
14.7 *Commit* do arquivo *index.html* 648
14.8 Histórico de modificações do repositório 648
14.9 Restauração do diretório de trabalho -- Parte 01 649
14.10 Restauração do diretório de trabalho -- Parte 02 650
14.11 Criação e listagem de *branches* 651
14.12 *Commits* e *merge* de *branches* 651
14.13 Histórico de *commits* em linha única e *branches* em modo gráfico 652
14.14 Erro de *merge* e conteúdo conflituoso dos arquivos 652
14.15 Resolução do conflito 653
14.16 Resultado após realizar um *fetch* 655
14.17 Resultado após realizar um *pull* ou *fetch* seguido de *merge* 655
14.18 Repositório *Developer Roadmap* 656
14.19 *README* do repositório *Developer Roadmap* 657
14.20 Histórico de *commits* do repositório *Developer Roadmap* 657
14.21 Resultado após realizar um *pull* ou *fetch* seguido de *merge* 658
14.22 Corpo de um *pull request* 659
14.23 Corpo de uma *issue* 659
14.24 Menu de criação 660
14.25 Criação de um repositório -- Parte 01 661
14.26 Criação de um repositório -- Parte 02 662
14.27 Repositório criado 663
14.28 Fazer o clone do repositório 663
14.29 Menu para adicionar arquivos 664
14.30 Página para criação de novo arquivo 664
14.31 *Commit* para criação do novo arquivo 665
14.32 *Pull request* -- Parte 01 666
14.33 *Pull request* -- Parte 02 667
14.34 *Pull request* -- Parte 03 667
14.35 Repositório *github-example* 668
14.36 Criação de um repositório *fork* 669
14.37 Repositório *fork* criado 670
14.38 Repositório *fork* após um *commit* 671
14.39 Criação do *pull request* do repositório *fork* para o repositório base 671
14.40 Arquitetura do Docker 673
14.41 Lista de imagens disponíveis 675
14.42 Lista de *containers* em execução 677
14.43 Requisição GET para *node-api-dev* 677
14.44 Requisição GET para *node-api-prod* 677
14.45 Tela inicial do WSO2 API Manager 680
14.46 Registro de *service provider* 680
14.47 Fragmento da tela de configuração do *service provider* 681

14.48 Menu de configuração do protocolo *OpenID Connect* 682
14.49 Ilustração do fluxo *authorization code* 683
14.50 Montagem da URL de login 684
14.51 Página de login da aplicação 685
14.52 Requisição do token de acesso 685
14.53 Requisição de informações do usuário 686

15.1 Automação Power Automate 695
15.2 Fluxograma de extração dos dados 698
15.3 Fluxograma de Tratamento dos Dados no MATLAB 699
15.4 Gráfico de Habilidades sobre *Front End* 702
15.5 Gráfico de Habilidades Back-end 703
15.6 Gráfico Habilidades Full Stack 705
15.7 Gráfico de Garantia de Qualidade 706
15.8 Gráfico de Soft Skills 707

Lista de Tabelas

1.1 Exemplos de ferramentas para validar de código HTML 52

2.1 Tipos de dados em *JavaScript* (JS) e *TypeScript* (TS) 94
2.2 Métodos e Atributos do Objeto Number em JS e TS 96
2.3 Modificadores de Acesso em *TypeScript* 124

3.1 Métodos e Definição HTTP. 150
3.2 Códigos de Status HTTP. 152
3.3 Comparação entre os modelos de implantação. 154

4.1 Métodos *Resource controller*. Adaptado de (STAUFFER, 2019) . 186
4.2 Comparação da camada de segurança entre os *frameworks* 238

5.1 Tabela colunar 249
5.2 Tabela que representa a entidade *empresas* 263
5.3 Tabela que representa a entidade *clientes*. 263
5.4 Resultado do *INNER JOIN* 263
5.5 Resultado do *CROSS JOIN* 264
5.6 Resultado do *LEFT JOIN* 265
5.7 Resultado da operação com *RIGHT JOIN* 265
5.8 Comparação Banco de dados relacionais vs. Banco de dados não relacionais (`MongoDB`) 269

6.1 Métodos *Hypertext Transfer Protocol* (HTTP) 290
6.2 Suporte das tecnologias aos principais Sistema Operacional (SO)s. 334
6.3 Suporte das tecnologias aos principais bancos de dados. 335
6.4 Suporte das tecnologias ao ORM nativo 335
6.5 Suporte das tecnologias à documentação nativa. 336

7.1 Verbos HTTP 343
7.2 Tabela de endpoints extraída da documentação da API Google Books 344
7.3 Códigos de status HTTP 347
7.4 Recursos e endpoints da API REST 354

7.5 Entidades representadas no banco de dados desenvolvido para as APIs 370
7.6 Operações que as APIs desenvolvidas podem executar 370
7.7 Entidades e atributos do banco de dados das APIs 372
7.8 Relacionamentos entre as tabelas do banco de dados 373
7.9 Frameworks e Bibliotecas integradas nas APIs 377

8.1 Comparação das abordagens de estilização no *React* 409

10.1 Elementos do padrão Dublin Core 486
10.2 Comparativo entre *Dublin Core* e *Schema.org* 488
10.3 Componentes de uma Tríplice RDF 501
10.4 Recursos e suas características em RDF 505
10.5 Descrição das sublinguagens do OWL 510

11.1 Níveis de testes de software 521
11.2 Especificação do caso de teste para o exemplo prático 532
11.3 Suporte das ferramentas Selenium e Cypress aos principais SOs . 548
11.4 Suporte das ferramentas Selenium e Cypress a diferentes linguagens de programação 548
11.5 Suporte das ferramentas Selenium e Cypress a diferentes navegadores 549
11.6 Instalação e configuração das ferramentas Selenium e Cypress . . 550
11.7 Recursos disponíveis das ferramentas Selenium e Cypress 550
11.8 Requisitos funcionais do SUT 552
11.9 Casos de Uso do SUT 553
11.10 Casos de Testes do SUT 553
11.11 Especificação do caso de teste TC01 554
11.12 Especificações do ambiente e máquina de desenvolvimento e execução dos testes 554
11.13 Métricas do desenvolvimento dos *scripts* de testes em Selenium e Cypress 556
11.14 Tempo de execução dos testes em Selenium e Cypress 557

12.1 Visão geral de ferramentas DevOps 564
12.2 Comparativo de tamanhos de imagens comuns e versões Alpine . . 571

14.1 Requisições HTTP do fluxo *authorization code* 682

15.1 Soft Skills para Desenvolvimento Web 707

A.1 Comparação entre as ferramentas Selenium e Cypress. 716

Capítulo 1

HTML e CSS

Jadson Faustino Mororó (UFC)

Michele Faria de Oliveira (UFC)

Fischer Ferreira (Unifei)

1.1 Introdução

O HTML (Linguagem de Marcação de HiperTexto, do inglês *HyperText Markup Language*) é a base da construção de páginas web. Ele define a estrutura e a organização do conteúdo, permitindo que os navegadores interpretem e exibam as informações de forma coerente para os usuários. Diferentemente de linguagens de programação, como JavaScript ou Python, o HTML é uma linguagem de marcação, ou seja, não executa lógica ou cálculos, mas sim estrutura o conteúdo para que seja apresentado corretamente na web (NETWORK, 2024d; GALVÃO, 2024).

A principal função do HTML é atribuir significado aos diferentes elementos da página por meio de suas tags e atributos. Isso garante que os navegadores saibam como renderizar cada parte do conteúdo e permite que mecanismos de busca (como o Google) compreendam melhor o propósito e a relevância de cada seção de um site. Essa estrutura é crucial para a otimização de mecanismos de busca (SEO), pois um código bem estruturado facilita a indexação do site, melhorando sua visibilidade nos resultados de pesquisa (NETWORK, 2024d; SANTOS, 2024).

O HTML trabalha em conjunto com outras tecnologias para a construção da web moderna. Enquanto o HTML define a estrutura e o conteúdo da página, o CSS (Cascading Style Sheets) é responsável pela estilização, definindo cores, fontes, espaçamentos e layouts. Já o JavaScript adiciona interatividade e dinamismo, permitindo a manipulação do conteúdo HTML em tempo real. Essa combinação de HTML, CSS e JavaScript forma a base do desenvolvimento web moderno (SANTOS, 2024; GALVÃO, 2024).

Para criar um código HTML e executá-lo no navegador, basta seguir um processo simples. Primeiro, você pode escrever seu código HTML em um editor de texto básico, como o Bloco de Notas (no *Windows*) ou TextEdit (no *macOS*). Por exemplo, você pode escrever o código HTML apresentado na Listagem 1.1.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale
        =1.0">
    <title>Minha Primeira Página</title>
</head>
<body>
    <h1>Olá, Mundo!</h1>
    <p>Esta é a minha primeira página HTML.</p>
</body>
</html>
```

Listagem 1.1: Exemplo um código em HTML

Depois de escrever o código, salve o arquivo com a extensão ".html", por exemplo, "minha_primeira_pagina.html". Para que o arquivo possa ser aberto no navegador e interpretado como HTML, o arquivo deve ser salvo com a extensão

".html". Em seguida, basta abrir o arquivo no seu navegador de internet, como o *Google Chrome*, *Firefox* ou *Edge*, clicando duas vezes sobre o arquivo salvo. O navegador interpretará o código HTML e exibirá o conteúdo da página conforme o definido pelas *tags*. Nesse exemplo, o navegador exibirá o título "Minha Primeira Página"na barra de título do navegador, seguido de um cabeçalho com o texto "Olá, Mundo!"e um parágrafo com o texto "Esta é a minha primeira página HTML."Assim, com HTML, você pode facilmente criar páginas web e visualizar os resultados imediatamente no seu navegador, conforme demonstrado na Figura 1.1. A Figura 1.1 apresenta uma demonstração do navegador interpretando o código em HTML da Listagem 1.1 que foi salvo com o nome "minha_primeira_pagina.html".

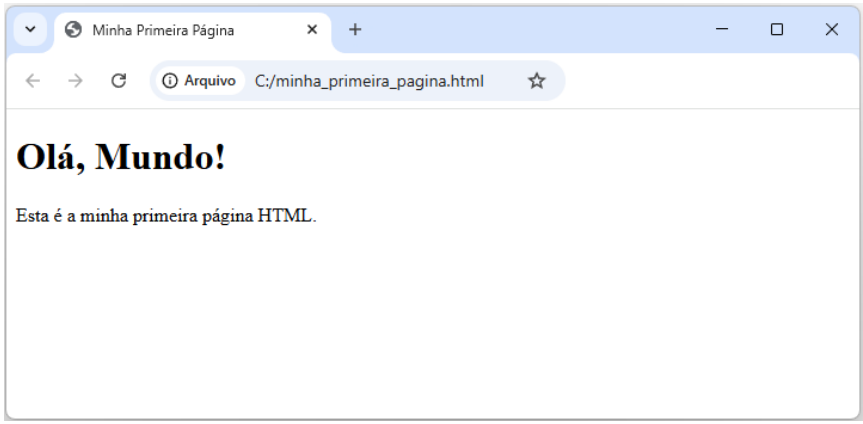


Figura 1.1: Exemplo do HTML sendo exibido no navegador

1.2 Sintaxe do HTML

O HTML utiliza marcações predefinidas para indicar ao navegador como exibir o conteúdo de uma página. Essas marcações são compostas por uma série de elementos que atuam como rótulos para diferentes partes do conteúdo, como títulos, parágrafos, tabelas, imagens, vídeos e áudios.

Os elementos do HTML são representados por *tags*, que estruturam o documento e permitem sua correta renderização pelo navegador. As *tags* são delimitadas por símbolos de menor que "<"e maior que ">"e podem conter atributos que definem suas propriedades, como cor, tamanho e posição.

Para aplicar a marcação a um texto, é necessário utilizar uma *tag* de abertura, seguida pelo conteúdo a ser formatado e, em seguida, uma tag de fechamento, que inclui uma barra "/". Por exemplo, como mostrado na Listagem 1.2, onde a tag `<p>` indica um parágrafo e deve ser fechada com `</p>`.

```
<p>Este é um parágrafo.</p>
```

Listagem 1.2: Exemplo de *tag* que abre e fecha

Há também algumas *tags* que são vazias e não possuem uma versão de fechamento, ou seja, não possuem conteúdo, como a *tag* `<br>`, usada para inserir quebras de linha e a *tag* `<img>`, usada para adicionar imagens. A Listagem 1.3 apresenta um exemplo de utilização da *tag* `img`, no qual você pode observar que ela não é fechada.

```
<img src="minha_imagem.jpg">
```

Listagem 1.3: Exemplo de *tag* que só abre

As *tags* HTML possuem atributos, que são usados para fornecer informações adicionais sobre um elemento e modificar seu comportamento ou aparência. O atributo é informado dentro da própria *tag*, após o nome da mesma, separado por espaço, mas antes do símbolo de maior que "`>`"que encerra a abertura da *tag*. A sintaxe básica de um atributo consiste no nome do atributo seguido do sinal de igual "`=`"e um valor entre aspas `" "` (NETWORK, 2024b; ALURA, 2024b). Como por exemplo na Listagem 1.3, já apresentada, onde a *tag* `<img>` é seguida de seu atributo `src`, que indica o caminho para o arquivo da imagem a ser exibida na página. Além disso, é possível combinar vários atributos em uma mesma *tag*, como na Listagem 1.4, onde, além do `src` são definidos também os atributos: `alt`, que fornece um texto alternativo para acessibilidade, e `width`, que especifica a largura da imagem exibida.

```
<img src="minha_imagem.jpg" alt="Descrição da imagem" width="300">
```

Listagem 1.4: Exemplo de uso de atributos na *tag*

Outra informação relevante sobre a sintaxe do HTML é que não há diferenciação entre letras maiúsculas e minúsculas, ou seja, é *case-insensitive*, mas, por convenção, as *tags* são escritas em minúsculas. Contudo, vale ressaltar que o conteúdo das *tags* e o valor de alguns atributos são *case sensitive*, ou seja, há sim diferenciação entre letras maiúsculas e minúsculas, como por exemplo no valor do atributo `src` da *tag* `<img>`, onde o valor "IMAGEM.PNG"não encontrará a imagem com nome "imagem.png"(NDUGU, 2020; GEEKSFORGEEKS, 2022).

Além disso, as *tags* podem ser aninhadas, ou seja, uma pode conter outra. Porém, a ordem de fechamento deve ser respeitada. Uma *tag* externa só pode ser fechada após todas as *tags* internas terem sido fechadas corretamente (CODING, 2024; DIGITALOCEAN, 2024). A Listagem 1.5 apresenta na primeira linha uma forma incorreta de aninhamento, onde as *tags* `<h1>` e `<strong>` são abertas, nessa ordem, e fechadas na ordem inversa. Já a segunda linha da Listagem 1.5 apresenta a forma correta.

```
<h1>Aninhamento <strong>incorreto</h1> aqui</strong>
<h1>Aninhamento <strong>correto</strong> aqui</h1>
```

Listagem 1.5: Exemplo de forma correta e incorreta de aninhamento de *tags*

Quando as *tags* são aninhadas, ou seja, uma *tag* está dentro de outra, é fundamental indentar corretamente as linhas de código. A indentação consiste em adicionar espaços ou tabulações antes de uma *tag* interna, com o objetivo de destacá-la em relação à *tag* externa. A indentação do código HTML é uma prática essencial para garantir a legibilidade e facilitar a compreensão da estrutura da página. Isso ajuda a visualizar a hierarquia entre as *tags* e facilita a identificação de erros (FREECODECAMP, 2024; ELEMENTOR, 2024). As Listagens 1.6 e 1.7 ilustram a indentação incorreta e correta, respectivamente, de um código HTML.

```
<html>
<head>
<title>Exemplo</title>
</head>
<body>
<h1>Título Principal</h1>
<p>Parágrafo de exemplo.</p>
</body>
</html>
```

Listagem 1.6: Exemplo de forma incorreta de indentação de código HTML

```
<html>
  <head>
    <title>Exemplo</title>
  </head>
  <body>
    <h1>Título Principal</h1>
    <p>Parágrafo de exemplo.</p>
  </body>
</html>
```

Listagem 1.7: Exemplo de forma correta de indentação de código HTML

Outro ponto importante é que o HTML ignora quebras de linha e múltiplos espaços em branco no código. Isso significa que, independentemente de como o conteúdo for formatado no código-fonte, o navegador interpretará o código de forma contínua, considerando apenas um único espaço entre palavras. Embora a indentação e o uso de quebras de linha sejam importantes para a legibilidade do código e para facilitar a manutenção, o descumprimento dessas práticas não causará erros na exibição do código HTML pelo navegador, pois o navegador não considera o espaçamento extra no código. Para forçar uma quebra de linha na exibição da página, é necessário utilizar a *tag* `<br>`, e para preservar a formatação original de

um bloco de texto, pode-se utilizar a *tag* `<pre>`. Mais *tags* serão apresentadas e melhor explicadas na Seção 1.4.

Em HTML, os comentários são usados para adicionar anotações no código que não serão exibidas no navegador. A sintaxe de um comentário em HTML começa com "`<!--`" e termina com "`-->`". Tudo o que estiver entre essas tags será ignorado pelo navegador. Comentários são úteis para documentar partes do código, desativar temporariamente trechos de código ou adicionar notas explicativas para outros desenvolvedores. A Listagem 1.8 ilustra o uso de comentários em código HTML.

```
<!-- Este é um comentário em HTML -->
<p>Este parágrafo será exibido no navegador.</p>
<!-- <p>Este parágrafo não será exibido.</p> -->
```

Listagem 1.8: Exemplo de comentários em HTML

1.3 Estrutura Básica do HTML

O arquivo HTML é dividido em duas partes principais: cabeçalho, definido pela *tag* `<head>`, e corpo, definido pela *tag* `<body>`. O `<head>` contém metainformações sobre a página HTML, como controle e caracterização, tais como título da página, links para folhas de estilo, *scripts* e configurações diversas. Por outro lado, o `<body>` contém todo o conteúdo visível do HTML (ALURA, 2024a; DIO, 2024). A Listagem 1.9 apresenta um exemplo simples de uma página HTML.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Minha primeira página</title>
    </head>
    <body>
        <h1>Olá Mundo!</h1>
        <p>Exemplo de parágrafo</p>
    </body>
</html>
```

Listagem 1.9: Exemplo de página HTML

Os demais elementos presentes na Listagem 1.9 são os seguintes:

- `<!DOCTYPE html>`: indica que o documento utiliza a versão mais recente do HTML (HTML5).
- `<html>`: *tag* raiz de uma página HTML.
- `<meta>`: *tag* que define metadados. No caso, apresenta o atributo `charset` definido como "UTF-8". Isso especifica que o conjunto de caracteres utilizado

será o UTF-8 (*Unicode Transformation Format – 8 bits*), uma codificação amplamente usada para suportar caracteres de praticamente todas as línguas do mundo, incluindo acentos e símbolos especiais.

- `<title>`: *tag* que especifica o título da página HTML, exibido na barra de título do navegador.
- `<h1>`: *tag* de título principal.
- `<p>`: *tag* de parágrafo.

1.4 *Tags* do HTML

O principal objetivo dessa seção é apresentar exemplos práticos de algumas *tags* mais utilizadas no HTML. Não é o intuito dos autores apresentar todas as *tags*, atributos e possibilidades disponíveis no HTML. Apesar disso, a seguir, são listadas algumas referências para que o leitor possa expandir a leitura e conhecer mais *tags* e possibilidades no HTML.

Algumas boas fontes de estudo das *tags* HTML são:

- W3Schools[1]
- MDN Web Docs[2]

1.4.1 *Tags* de texto

As *tags* de texto em HTML são usadas para estruturar, formatar e organizar o conteúdo textual em uma página da web. Elas permitem definir títulos, parágrafos, ênfases, listas e outros elementos essenciais para a apresentação e acessibilidade do conteúdo. As principais *tags* de texto do HTML são:

- `<h1>` **a** `<h6>` – **Títulos e subtítulos:** Essas tags definem a hierarquia de títulos em uma página, sendo `<h1>` o mais importante e `<h6>` o menos importante.
- `<p>` – **Parágrafos:** É utilizado para definir blocos de texto em parágrafos.
- `<span>` – **Texto em linha sem significado semântico:** Essa tag é usada para aplicar estilos a partes específicas do texto sem indicar um significado especial.
- `<strong>` – **Texto em negrito (ênfase forte):** Indica que o texto dentro da tag deve ser exibido em negrito para dar ênfase.

[1] **Disponível em:** ⟨https://www.w3schools.com/tags/default.asp⟩.
[2] **Disponível em:** ⟨https://developer.mozilla.org/pt-BR/docs/Web/HTML/Element⟩.

- `<em>` – **Texto em itálico (ênfase leve):** Usado para indicar ênfase leve, geralmente exibido em itálico.
- `<mark>` – **Texto destacado:** Destaca o texto como se tivesse sido marcado com um marca-texto.
- `<small>` – **Texto menor:** Reduz o tamanho do texto, geralmente usado para notas de rodapé.
- `<del>` – **Texto riscado:** Indica que um texto foi removido ou está obsoleto, sendo apresentado com um risco ao redor dele.
- `<ins>` – **Texto sublinhado:** Usado para indicar um texto inserido ou adicionado, geralmente exibido em sublinhado.
- `<blockquote>` – **Citação longa:** É utilizado para indicar citações extensas de outras fontes, geralmente recuado.
- `<cite>` – **Citação curta:** Indica o título de uma obra citada, como livros, filmes ou artigos.

A Listagem 1.10 mostra como usar cada uma das *tags* de texto apresentadas anteriormente.

```
<h1>Título Principal</h1>
<h2>Subtítulo</h2>
<p>Este é um parágrafo de texto.</p>
<span>Tag genérica somente para estilizar</span>
<strong>Texto com ênfase forte e em negrito</strong>
<em>Texto com ênfase leve e em itálico</em>
<mark>Texto destacado</mark>
<small>Texto menor</small>
<del>Texto riscado</del>
<ins>Texto sublinhado</ins>
<blockquote>"Citação longa aqui."</blockquote>
<p><cite>O Senhor dos Anéis</cite> é um livro...</p>
```

Listagem 1.10: *Tags* de texto em HTML

1.4.2 *Tags* de multimídia

As *tags* de multimídia são usadas para incorporar e manipular conteúdos, como áudio, vídeo e imagens, em uma página da web. A seguir, são apresentadas as principais *tags* de multimídia do HTML:

- `<img>` – **Exibir imagens:** Exibe imagens em uma página web. Seus principais atributos são:
  - `src` – Define o caminho da imagem.

  - `alt` – Fornece um texto alternativo caso a imagem não possa ser carregada, o que é útil para acessibilidade.
  - `width` – Define a largura da imagem.
  - `height` – Define a altura da imagem.

- `<audio>` – **Inserir áudio:** Insere arquivos de áudio na página. Seus principais atributos são:
  - `src` – Define o caminho do arquivo de áudio.
  - `controls` – Adiciona controles de reprodução (play, pause, volume).
  - `autoplay` – Faz o áudio iniciar automaticamente.
  - `loop` – Faz o áudio repetir continuamente.

- `<video>` – **Inserir vídeo:** Exibe vídeos na página. Seus principais atributos são:
  - `src` – Define o caminho do arquivo de vídeo.
  - `controls` – Adiciona controles de reprodução.
  - `autoplay` – Inicia o vídeo automaticamente.
  - `loop` – Faz o vídeo se repetir continuamente.
  - `poster` – Define uma imagem exibida antes do vídeo começar.

- `<source>` – **Fonte alternativa para `<audio>` e `<video>`:** Define diferentes formatos de mídia para compatibilidade entre navegadores. Seus principais atributos são:
  - `src` – Caminho do arquivo de multimídia.
  - `type` – Formato do arquivo (`audio/mpeg`, `video/mp4`, etc.).

- `<track>` – **Legendas e descrições para vídeos:** Permite adicionar legendas e descrições em vídeos para acessibilidade. Seus principais atributos são:
  - `src` – Define o caminho do arquivo de legenda.
  - `kind` – Especifica o tipo da faixa (`subtitles`, `captions`, `descriptions`, etc.).
  - `srclang` – Define o idioma da legenda.
  - `label` – Nome da faixa de legenda exibido no player.
  - `default` – Indica se a faixa deve ser usada por padrão.

- `<figure>` – **Agrupar imagens e legendas:** Agrupa elementos de mídia, como imagens e vídeos, com suas respectivas legendas, facilitando a organização do conteúdo.

- `<figcaption>` – **Legenda para `<figure>`:** Fornece uma legenda descritiva para os elementos contidos em um `<figure>`.

A Listagem 1.11 mostra como usar cada uma das *tags* de multimídia apresentadas nesta subseção.

```
<!-- Imagem -->
<img src="imagem.jpg" alt="Descrição da imagem">

<!-- Audio -->
<audio controls>
    <source src="audio.mp3" type="audio/mpeg">
    Seu navegador não suporta o elemento de áudio.
</audio>

<!-- Video/Legenda -->
<video controls>
    <source src="video.mp4" type="video/mp4">
    <source src="video.ogg" type="video/ogg">
    Seu navegador não suporta a reprodução de vídeo.
    <track src="legendas.vtt" kind="subtitles" srclang="pt" label="
        Português">
</video>

<!-- Figura/Legenda -->
<figure>
    <img src="imagem.jpg" alt="Descrição da imagem">
    <figcaption>Legenda da imagem</figcaption>
</figure>
```

Listagem 1.11: *Tags* de multimídia em HTML

1.4.3 *Tags* de listagem

As listas são elementos essenciais no HTML para organizar e estruturar informações de forma clara. Existem três tipos principais de listas: não ordenadas, ordenadas e de definição. As principais *tags* utilizadas para criar listas em HTML são:

- `<ul>` – **Listas não ordenadas:** Cria uma lista em que os itens são apresentados com marcadores visuais, como bolinhas ou quadrados.
- `<ol>` – **Listas ordenadas:** Cria uma lista em que os itens são numerados automaticamente, indicando uma ordem de importância.
- `<li>` – **Itens de lista:** Representa um item dentro de uma lista, podendo ser usado dentro de `<ul>` ou `<ol>`.
- `<dl>` – **Listas de definição:** Define uma lista de termos e suas respectivas descrições, usada em glossários ou descrições estruturadas.

- `<dt>` – **Termo de definição:** Usado dentro de `<dl>` para representar o termo a ser definido.
- `<dd>` – **Descrição de definição:** Usado dentro de `<dl>` para fornecer a explicação ou definição de um termo listado com `<dt>`.

A Listagem 1.12 apresenta um código HTML exemplificando o uso das *tags* de listagem dos três tipos diferentes de lista apresentadas.

```
<!-- Lista não ordenada -->
<ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>

<!-- Lista ordenada -->
<ol>
    <li>Primeiro item</li>
    <li>Segundo item</li>
    <li>Terceiro item</li>
</ol>

<!-- Lista de definição -->
<dl>
    <dt>Termo 1</dt>
    <dd>Definição do termo 1</dd>
    <dt>Termo 2</dt>
    <dd>Definição do termo 2</dd>
</dl>
```

Listagem 1.12: *Tags* de listagem em HTML

1.4.4 *Tags* de tabela

As tabelas são usadas no HTML para exibir dados de forma estruturada, organizando informações em linhas e colunas. As *tags* relacionadas a construção de tabelas em HTML são:

- `<table>` – **Criar uma tabela:** Define uma tabela no HTML, agrupando todas as suas partes. Seus principais atributos são:
  - `border` – Define a espessura da borda da tabela.
  - `cellspacing` – Controla o espaçamento entre as células da tabela.
  - `cellpadding` – Define o espaçamento interno entre as células.
- `<tr>` – **Linha da tabela:** Representa uma linha da tabela, agrupando células de dados ou cabeçalhos.

- `<td>` – **Célula da tabela:** Representa uma célula de dados em uma linha da tabela. Seus principais atributos são:
  - `colspan` – Define quantas colunas a célula deve ocupar.
  - `rowspan` – Define quantas linhas a célula deve ocupar.
- `<th>` – **Cabeçalho da tabela:** Define uma célula como cabeçalho dentro de uma linha, geralmente exibida em negrito. Seus principais atributos são:
  - `scope` – Indica se o cabeçalho se aplica a uma coluna (`col`) ou a uma linha (`row`).
  - `colspan` – Define quantas colunas a célula do cabeçalho deve abranger.
  - `rowspan` – Define quantas linhas a célula do cabeçalho deve abranger.
- `<thead>` – **Cabeçalho do conjunto de dados:** Agrupa o cabeçalho da tabela, separando-o do restante dos dados.
- `<tbody>` – **Corpo da tabela:** Agrupa as linhas que contêm os dados principais da tabela.
- `<tfoot>` – **Rodapé da tabela:** Define um rodapé para a tabela, útil para totais e resumos.
- `<caption>` – **Legenda da tabela:** Adiciona um título descritivo à tabela, proporcionando maior acessibilidade e organização.
- `<colgroup>` **e** `<col>` – **Definir colunas da tabela:** Permite aplicar estilos e configurações às colunas da tabela. Seus principais atributos são:
  - `span` – Define quantas colunas devem ser afetadas pelo grupo.
  - `width` – Define a largura das colunas.

A Listagem 1.13 apresenta um exemplo de código HTML que utiliza as *tags* de tabela mencionadas.

```
<table>
    <caption>Exemplo de Tabela</caption>
    <colgroup>
        <col style="background-color: #bbb">
        <col style="background-color: #ddd">
    </colgroup>
    <thead>
        <tr>
            <th>Nome</th>
            <th>Idade</th>
            <th>Cidade</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>Ana</td>
            <td>30</td>
            <td>São Paulo</td>
        </tr>
        <tr>
            <td>Maria</td>
            <td>35</td>
            <td>Belo Horizonte</td>
        </tr>
    </tbody>
    <tfoot>
        <tr>
            <td colspan="3">Total de pessoas: 2</td>
        </tr>
    </tfoot>
</table>
```

Listagem 1.13: *Tags* de tabela em HTML

1.4.5 *Tags* de formulário

Os formulários são utilizados para coletar informações dos usuários por meio de diferentes tipos de campos de entrada. As *tags* mais usadas para a construção de formulários em HTML são:

- `<form>` – **Container do formulário:** Define um formulário e agrupa seus campos. Seus principais atributos são:
  - `action` – Define o URL para onde os dados serão enviados.
  - `method` – Especifica o método HTTP de envio (`GET` ou `POST`).
- `<input>` – **Campo de entrada:** Permite a entrada de diversos tipos de dados. Seus principais atributos são:
  - `type` – Define o tipo do campo (`text`, `password`, `email`, `checkbox`, etc.).

  - `name` – Nome do campo, usado para identificar os dados no envio.
  - `value` – Define um valor inicial para o campo.
  - `placeholder` – Exibe um texto sugestivo dentro do campo.
  - `required` – Indica que o campo é obrigatório.

- `<textarea>` – **Área de texto longa:** Permite a entrada de múltiplas linhas de texto. Seus principais atributos são:
  - `rows` – Número de linhas visíveis.
  - `cols` – Número de colunas visíveis.
  - `placeholder` – Texto sugestivo dentro da área de texto.

- `<button>` – **Botão interativo:** Cria um botão que pode ser usado para enviar um formulário. Seus principais atributos são:
  - `type` – Define o tipo do botão (`submit`, `reset`, `button`).

- `<label>` – **Rótulo associado a um campo:** Fornece uma descrição textual para os campos de entrada. Seu principal atributo é:
  - `for` – Associa o rótulo a um campo específico por meio do `id` do campo.

- `<select>` – **Lista suspensa:** Permite ao usuário selecionar uma opção de uma lista. Seus principais atributos são:
  - `name` – Nome do campo no envio do formulário.
  - `multiple` – Permite a seleção de múltiplas opções.

- `<option>` – **Opção dentro de `<select>`:** Define uma opção em um menu suspenso. Seus principais atributos são:
  - `value` – Define o valor a ser enviado quando a opção é selecionada.
  - `selected` – Indica que a opção está selecionada por padrão.

- `<optgroup>` – **Agrupar opções dentro de `<select>`:** Permite agrupar visualmente as opções dentro de um menu suspenso. Seu principal atributo é:
  - `label` – Define um rótulo para o grupo de opções.

- `<fieldset>` – **Agrupar campos de formulário:** Agrupa logicamente campos relacionados dentro de um formulário.

- `<legend>` – **Legenda do `<fieldset>`:** Define um título descritivo para um `<fieldset>`.

- `<datalist>` – **Sugestões para `<input>`:** Define uma lista de sugestões para um campo de entrada.

- `<output>` – **Exibir saída de um cálculo:** Mostra um resultado calculado dentro do formulário.
- `<meter>` – **Indicador de medição:** Representa um valor dentro de um intervalo conhecido (como nível de bateria ou temperatura). Seus principais atributos são:
  - `min` – Valor mínimo aceitável.
  - `max` – Valor máximo aceitável.
  - `value` – Valor atual da medição.
- `<progress>` – **Barra de progresso:** Exibe o progresso de uma tarefa em andamento. Seus principais atributos são:
  - `max` – Valor total da tarefa.
  - `value` – Progresso atual da tarefa.

A Listagem 1.14 apresenta um exemplo de código HTML que utiliza as *tags* de formulário mencionadas.

```
<form action="/enviar" method="post">
    <fieldset>
        <legend>Informações Pessoais</legend>

        <label for="nome">Nome:</label>
        <input type="text" id="nome" name="nome" required>

        <label for="email">E-mail:</label>
        <input type="email" id="email" name="email" required>

        <label for="msg">Mensagem:</label>
        <textarea id="msg" name="msg"></textarea>

        <label for="opcoes">Escolha uma opção:</label>
        <select id="opcoes" name="opcoes">
            <optgroup label="Grupo 1">
                <option value="1">Opção 1</option>
                <option value="2">Opção 2</option>
            </optgroup>
        </select>

        <label for="progresso">Progresso:</label>
        <progress value="50" max="100"></progress>

        <button type="submit">Enviar</button>
    </fieldset>
</form>
```

Listagem 1.14: *Tags* de formulário em HTML

1.4.6 *Tags* de estrutura e organização

As *tags* de estrutura e organização no HTML são utilizadas para definir a hierarquia e a disposição dos elementos dentro da página, proporcionando uma melhor organização do conteúdo. As *tags* mais comuns dessa categoria são:

- `<html>` – **Elemento raiz do HTML:** Define o início e o fim de um documento HTML, englobando todo o conteúdo da página.
- `<head>` – **Metadados e configurações:** Contém informações sobre a página, como título, links para estilos e scripts. Seus principais elementos são:
  - `<title>` – Define o título da página exibido na aba do navegador e nos mecanismos de busca.
  - `<meta>` – Especifica metadados, como codificação de caracteres e descrição da página.
  - `<link>` – Importa folhas de estilo (CSS) e ícones.
  - `<script>` – Inclui arquivos ou códigos *JavaScript*.
- `<body>` – **Conteúdo visível da página:** Contém todo o conteúdo visível pelo usuário, incluindo texto, imagens, vídeos e outros elementos interativos.
- `<header>` – **Cabeçalho da página ou seção:** Representa uma seção introdutória ou de navegação, geralmente contendo logotipos, menus ou títulos.
- `<nav>` – **Barra de navegação:** Define um conjunto de links de navegação, como menus principais e breadcrumbs.
- `<main>` – **Conteúdo principal:** Representa o conteúdo principal da página, excluindo cabeçalhos, rodapés e barras laterais.
- `<section>` – **Seção de conteúdo:** Agrupa conteúdos relacionados dentro da página, auxiliando na organização do documento.
- `<article>` – **Artigo ou postagem complementar:** Define um conteúdo independente e autocontido, como postagens de blog ou notícias.
- `<aside>` – **Conteúdo complementar:** Contém informações adicionais ou secundárias, como barras laterais e widgets.
- `<footer>` – **Rodapé da página ou seção:** Representa a parte final de uma página ou seção, geralmente contendo informações de contato, direitos autorais ou links adicionais.
- `<div>` – **Divisão de conteúdo sem significado semântico:** Usada para agrupar e estilizar elementos sem significado semântico próprio, auxiliando na estruturação e personalização do layout.

- `<a>` – **Link de navegação:** Define um link de navegação para outras páginas ou recursos. O atributo principal é `href`, que especifica o destino do link. Também pode ter atributos como:
  - `target` – Define onde o link será aberto (por exemplo, em uma nova aba com o valor `_blank`).
  - `title` – Fornece uma dica ou descrição sobre o link, exibida quando o usuário passa o mouse sobre o link.

```
<header>
  <h1>Bem-vindo ao Meu Site</h1>
  <nav>
    <ul>
      <li><a href="#home">Home</a></li>
      <li><a href="#about" target="_blank">Sobre</a></li>
      <li><a href="#contact">Contato</a></li>
    </ul>
  </nav>
</header>

<main>
  <section>
    <article>
      <h2>Artigo 1</h2>
      <p>Conteúdo do artigo 1...</p>
    </article>
    <article>
      <h2>Artigo 2</h2>
      <p>Conteúdo do artigo 2...</p>
    </article>
  </section>

  <aside>
    <p>Conteúdo complementar, como links ou informações extras.</p>
  </aside>
</main>

<footer>
  <p>Rodapé do site &copy; 2025</p>
</footer>
```

Listagem 1.15: *Tags* de estrutura e organização em HTML

1.5 Atributos comuns de *tags* HTML

Os atributos comuns de *tags* do HTML são aqueles que podem ser utilizados em praticamente qualquer *tag*, independentemente do elemento. Eles permitem personalizar o comportamento, a acessibilidade e a estilização dos elementos. A seguir, são apresentados os principais atributos de *tags*:

- `id` – **Identificador único:** Define um identificador único para o elemento, permitindo referenciá-lo via CSS ou *JavaScript*. Não se deve atribuir o mesmo valor de `id` a diferentes elementos da página.
- `class` – **Classe:** Permite associar um elemento a uma ou mais classes, facilitando a estilização e a manipulação de um grupo de elementos via CSS ou *JavaScript*. As diferentes classes de um elemento devem ser separadas por espaço no valor de `class`.
- `style` – **Estilização *inline*:** Define estilos CSS diretamente no elemento, permitindo personalizações específicas sem a necessidade de um arquivo CSS externo. O valor de `style` segue a mesma sintaxe do CSS (apresentada na Seção 1.7), mas sem seletores pois se trata de um elemento em específico.
- `title` – **Texto descritivo:** Exibe um texto quando o usuário passa o mouse sobre o elemento, fornecendo informações adicionais.
- `hidden` – **Ocultar elemento:** Indica que o elemento não deve ser exibido na página, mas ainda pode ser acessado via *JavaScript*.
- `tabindex` – **Ordem de navegação:** Define a ordem de navegação do teclado ao pressionar a tecla `Tab`, melhorando a acessibilidade.
- `aria-*` – **Acessibilidade:** Conjunto de atributos que melhoram a experiência para usuários com deficiência, fornecendo informações adicionais para leitores de tela.
- `data-*` – **Armazenamento de dados personalizados:** Permite armazenar dados personalizados em um elemento HTML para uso em *scripts* no *JavaScript*.

O asterisco `*` em atributos como `aria-*` e `data-*` significa que esses atributos são dinâmicos e podem ter diversos sufixos personalizados. Eles funcionam como "espaços reservados" para diferentes valores, permitindo que os desenvolvedores definam variantes específicas conforme necessário.

1.6 Esclarecimentos e boas práticas em HTML

Essa seção busca esclarecer algumas confusões comuns e desincentivar algumas práticas ruins em relação ao HTML e recomendar boas práticas desde já. Os esclarecimentos e recomendações elencados para essa seção são:

- **HTML não é uma linguagem de programação:** O HTML é uma linguagem de marcação utilizada para estruturar e organizar conteúdo na web. Diferentemente de linguagens como Java, C++ e Python, usadas para criar aplicações, o HTML serve para definir a estrutura dos documentos web.

- **Separação entre estrutura e estilo:** O HTML deve ser utilizado apenas para estruturar o conteúdo, enquanto o CSS deve ser responsável pela estilização. A definição de estilos diretamente no HTML, como o uso do atributo `style`, deve ser evitada, pois dificulta a manutenção do código.

- **Evitar o uso de elementos e atributos obsoletos:** Alguns elementos e atributos foram descontinuados no HTML5 e não devem mais ser utilizados. Alguns exemplos são as *tags* `<font>`, `<center>`, `<frame>` e `<menu>`. O uso de elementos obsoletos pode prejudicar a compatibilidade com navegadores modernos, usabilidade, acessibilidade e indexação da página nos mecanismos de busca.

- **Uso correto e semântico das *tags*:** O HTML5 introduziu diversas *tags* semânticas, como `<header>`, `<nav>`, `<section>` e `<article>`, que ajudam a organizar o conteúdo de forma mais clara e acessível. Utilizar essas *tags* corretamente melhora a usabilidade e a indexação pelos mecanismos de busca. Usar as *tags* de título (`<h1>` a `<h6>`) em ordem hierárquica, a *tag* `<p>` exclusivamente para parágrafos, as *tags* `<ul>` ou `<ol>` para listas com `<li>` para os itens são exemplos de usos corretos para as *tags*. O uso inadequado dessas *tags* compromete a acessibilidade e a otimização para mecanismos de busca.

- **Evitar tabelas para layout:** As tabelas devem ser usadas apenas para organizar dados tabulares. Criar layouts de páginas usando `<table>` torna o código menos flexível, dificulta a acessibilidade e prejudica a adaptação para dispositivos móveis.

- **Uso correto dos atributos:** Os atributos HTML devem ser utilizados corretamente para garantir a acessibilidade e a compatibilidade. Um exemplo é o atributo `alt` em imagens, que fornece uma descrição alternativa para usuários com deficiência visual.

- **Validação do código HTML:** Validar o código HTML ajuda a garantir sua conformidade com os padrões da web. IDEs modernas oferecem ferramentas integradas para validação, e há serviços online, como o *W3C Validator*, que permitem identificar erros e melhorar a qualidade do código. A Tabela 1.1 apresenta algumas ferramentas *online* que podem ser utilizadas para validar o código HTML, o que pode evitar problemas de compatibilidade e garantir a melhor qualidade do código, caso não esteja utilizando uma IDE.

O seguimento das boas práticas citadas melhora a experiência de codificação, além de melhorar a qualidade e acessibilidade da página produzida (NETWORK, 2024a; KINSTA, 2024).

Tabela 1.1: Exemplos de ferramentas para validar de código HTML

Ferramentas	URL
W3C Markup Validation Service	⟨https://validator.w3.org/⟩
HTML5 Validator	⟨https://html5.validator.nu/⟩
Validator.nu	⟨https://validator.nu/⟩
Total Validator	⟨https://www.totalvalidator.com/⟩
Code Beautify	⟨https://codebeautify.org/⟩

1.7 CSS e sua Sintaxe

CSS (*Cascading Style Sheets*) é uma linguagem de estilo utilizada para definir a aparência e o layout de elementos em páginas da web. Ele trabalha em conjunto com o HTML, permitindo que os desenvolvedores controlem cores, fontes, espaçamentos, tamanhos e outros aspectos visuais dos elementos (NETWORK, 2024c; W3SCHOOLS, 2024).

O CSS pode ser aplicado de três formas: inline (diretamente no elemento HTML), interno (dentro da *tag* `<style>` no HTML) ou externo (arquivo separado .css). A melhor prática é usar arquivos externos para facilitar a manutenção do código e reaproveitamento dos estilos em diferentes páginas (HOSTINGER, 2024).

A sintaxe do CSS é composta por regras que definem estilos para elementos HTML. Cada regra consiste em um seletor e um bloco de declarações. O seletor define o elemento HTML que será estilizado. Dentro do bloco de declarações, cada linha especifica uma propriedade e um valor correspondente. As propriedades determinam os aspectos visuais, como cor, espaçamento e tamanho da fonte (NETWORK, 2024c). A Listagem 1.16 apresenta a sintaxe básica de declarações CSS.

```
seletor-1 {
    propriedade-1: valor-1;
    propriedade-2: valor-2;
}
seletor-2 {
    propriedade-3: valor-3;
}
```

Listagem 1.16: Sintaxe básica de declarações CSS

Um exemplo prático é apresentado na Listagem 1.17, onde todos os parágrafos "`<p>`"da página são estilizados. Eles serão exibidos na cor azul, com tamanho de fonte de 16 pixels e uma margem de 10 pixels ao redor.

```
p {
    color: blue;
    font-size: 16px;
    margin: 10px;
}
```

Listagem 1.17: Sintaxe básica de declarações CSS

No exemplo apresentado, o seletor é o nome da *tag*, porém, mais formas de seletores serão apresentadas na Seção 1.8. Além disso, no exemplo, a cor é o nome da cor em inglês e a unidade de medida usada na margem e no tamanho da fonte é o pixel; contudo, mais formas de definir valores, unidades e cores serão apresentadas na Seção 1.11.

Como o HTML, o CSS é majoritariamente *case insensitive*, o que significa que ele não diferencia letras maiúsculas de minúsculas. Contudo, o uso de letras minúsculas é recomendado, isso se aplica tanto aos seletores quanto aos nomes de propriedades e valores, exceto para os valores que exigem letras maiúsculas em alguns casos, como nomes de fontes, URLs, etc. (CROZAT, 2024).

Uma parte também importante da sintaxe do CSS são os comentários, que podem ser adicionados ao código utilizando a sintaxe "`/* comentário */`". Um código de exemplo da utilização de comentários em CSS é apresentado na Listagem 1.18.

```
/* Isso é um comentário sobre o estilo da tag <p> */
p {
  color: blue; /* Define a cor do texto como azul */
  font-size: 16px; /* 16 pixels de tamanho da fonte */
  margin: 10px; /* Margem de 10 pixels */
/*text-decoration: underline; Estilo desconsiderado */
}
/*
  Isso é
  um comentário
  de várias linhas
*/
```

Listagem 1.18: Comentários em CSS

O termo "*Cascading*" no CSS descreve como os estilos são aplicados e priorizados. Quando várias regras afetam o mesmo elemento, o CSS segue uma hierarquia de prioridade para determinar qual estilo será aplicado. Sendo o item 1 a menor prioridade e o 3 a maior, a ordem de prioridade é a seguinte:

1. **Especificidade dos seletores:** Regras com seletores mais específicos (como IDs ou classes) têm maior prioridade do que as com seletores mais gerais (como nomes de *tag*).
2. **Ordem das declarações:** Se duas regras tiverem a mesma especificidade, a última regra definida no código será a aplicada.

3. **Sinalizador de importância:** O uso do sinalizador `!important` pode sobrescrever outras regras, independentemente da especificidade ou da ordem de declaração. No entanto, deve ser usado com moderação para evitar problemas de manutenção.

Na Listagem 1.19, pode-se observar o uso de `!important`. No exemplo, o texto do parágrafo será estilizado com a cor azul devido ao uso do `!important`, mesmo que uma outra definição de cor vermelha para o texto seja feita depois e com um seletor mais específico. Caso a declaração `important!` seja removida, o texto passará a ficar vermelho.

```
p {
  color: blue !important;
  font-size: 16px;
}

p.special {
  color: red;
}
```

Listagem 1.19: Ordem de prioridade de declarações em CSS

Além disso, o CSS tem herança, o que significa que algumas propriedades, como cor e fonte, são automaticamente herdadas dos elementos pai para os filhos, a menos que sejam sobrescritas por regras mais específicas. Um elemento pai é aquele que contém outros elementos, ou seja, sua *tag* aninha um ou mais outros elementos, denominados elementos filhos. Por exemplo, na Listagem 1.20 o elemento `<span>` é filho do elemento `<p>`, e o elemento `<p>` é o pai do elemento `<span>`. O exemplo apresenta um código HTML com CSS interno, através da *tag* `<style>`, onde o primeiro parágrafo apresentará a palavra "Estilo" em vermelho, por herança, e o segundo parágrafo a apresentará em azul, por haver regras mais específicas e assim sobrescrevendo o estilo de herança.

```
<style>
    p {
        color: red;
    }

    #azul {
        color: blue;
    }
</style>
<p><span>Estilo</span> herdado</p>
<p><span id="azul">Estilo</span> sobrescrito</p>
```

Listagem 1.20: Herança de estilo em CSS

1.8 Seletores em CSS

Os seletores em CSS são responsáveis por definir quais elementos HTML serão estilizados. Eles permitem que o desenvolvedor aplique estilos de forma específica e eficiente, utilizando diferentes abordagens para selecionar elementos individuais ou grupos de elementos com base em seus atributos e relações.

O seletor de *tag* aplica estilos a todos os elementos de um determinado tipo. Isso significa que, se um seletor for aplicado a uma tag, todos os elementos dessa tag na página receberão o estilo. Por exemplo, como o código apresentado na Listagem 1.21, usado para estilizar todos os parágrafos de uma página.

```
p {
    color: blue;
}
```

Listagem 1.21: Seletor de nome de *tag* em CSS

Se for necessário aplicar estilos a um elemento específico, pode-se usar o seletor de ID. Esse seletor é precedido pelo caractere de cerquilha "`#`"e deve ser único na página. Um exemplo de uso de seletor de ID é mostrado na Listagem 1.22.

```
#cabecalho {
    background-color: gray;
}
```

Listagem 1.22: Seletor de ID em CSS

Quando se deseja aplicar estilos a vários elementos sem precisar repetir regras, o seletor de classe é uma opção mais flexível. Ele é representado pelo caractere de ponto "`.`"e pode ser aplicado a múltiplos elementos. Pode-se observar na Listagem 1.23 um código que ilustra o uso do seletor de classe em CSS.

```
.destaque {
    font-weight: bold;
}
```

Listagem 1.23: Seletor de classe em CSS

Além desses seletores básicos, é possível criar seletores mais específicos utilizando seletores combinados. O caminho do seletor pode ser definido para estilizar elementos dentro de um contexto específico. Por exemplo, para estilizar apenas os parágrafos que estejam dentro de um elemento com a classe `conteudo`, utiliza-se o seletor presente no código da Listagem 1.24.

```
.conteudo p {
    text-align: justify;
}
```

Listagem 1.24: Seletor de caminho em CSS

Os seletores combinados também podem incluir modificadores para definir relações mais específicas entre elementos. O seletor com símbolo de maior que “>”seleciona elementos filhos diretos, ou seja, apenas aqueles que estão imediatamente dentro do elemento pai especificado. O código apresentado na Listagem 1.25 ilustra um exemplo de uso desse seletor.

```
div > p {
    color: red;
}
```

Listagem 1.25: Seletor de elemento filho em CSS

O seletor com símbolo de mais “+”seleciona um elemento imediatamente seguinte ao elemento especificado, desde que ambos compartilhem o mesmo elemento pai. Por exemplo, para estilizar apenas o primeiro parágrafo logo após um título `h1`, utiliza-se o seletor apresentado na Listagem 1.26.

```
h1 + p {
    font-size: 14px;
}
```

Listagem 1.26: Seletor de elemento irmão imediato em CSS

Já o seletor com símbolo de til “~”funciona de forma semelhante ao seletor com símbolo de mais “+”, porém, ao invés de afetar apenas o primeiro elemento seguinte, ele aplica estilos a todos os elementos irmãos subsequentes. A Listagem 1.27 apresenta um código que exemplifica o uso desse seletor.

```
h1 ~ p {
    font-style: italic;
}
```

Listagem 1.27: Seletor de elementos irmãos gerais em CSS

Esses modificadores proporcionam um controle mais refinado sobre quais elementos serão afetados pelas regras CSS, tornando o código mais eficiente e organizado, além de evitar a aplicação desnecessária de estilos a elementos não desejados.

1.9 Pseudo-classes e Transições em CSS

As pseudo-classes em CSS permitem a estilização de elementos com base em seu estado ou posição no documento, sem a necessidade de modificar o HTML diretamente. Elas são representadas pelo uso do caractere de dois pontos “:”seguido do nome da pseudo-classe.

As pseudo-classes em CSS são as seguintes:

- `hover`: aplica estilos a um elemento quando o usuário posiciona o cursor sobre ele.
- `focus`: estiliza um elemento quando ele recebe foco, como um campo de entrada de formulário quando está aguardando entrada do usuário.
- `active`: aplica estilos a um elemento enquanto ele está sendo ativado, geralmente ao ser clicado.
- `nth-child(n)`: permite selecionar elementos com base em sua posição dentro de um elemento pai. A expressão passada dentro dos parênteses é o que define a posição ordinária (ordem dos elementos dentro do elemento pai) dos elementos que serão afetados. O `n` na expressão assume qualquer valor inteiro a partir de 1 (um). Apenas os resultados positivos da expressão são considerados, então, mesmo que a expressão resulte em zero para um dos valores de `n`, por não existir "zerésimo" elemento, esse valor é ignorado. Alguns exemplos de expressões para essa pseudo-classe são:
  - A expressão `nth-child(n)` afeta todos os filhos, pois $n \in [0, 1, 2, 3, ...)$. Então, os elementos filhos afetados serão o primeiro, o segundo, terceiro e assim por diante. Como já mencionado, por não existir "zerésimo" elemento, esse valor é desconsiderado.
  - A expressão `nth-child(2n)` afeta todos os elementos pares, pois $n \in [0, 1, 2, 3, ...)$, logo, $2n \in [0, 2, 4, 6, ...)$. Então os elementos filhos afetados serão o segundo, o quarto, a sexta e assim por diante.
  - A expressão `nth-child(2n + 1)` afeta todos os elementos ímpares, pois $n \in [0, 1, 2, 3, ...)$, logo, $2n + 1 \in [1, 3, 5, 7, ...)$. Então os elementos filhos afetados serão o primeiro, o terceiro, o quinto e assim por diante.
  - A expressão `nth-child(n + 2)` afeta todos os elementos a partir do segundo elemento, pois $n \in [0, 1, 2, 3, ...)$, logo, $n + 2 \in [2, 3, 4, 5, ...)$. Então os elementos filhos afetados serão o segundo, o terceiro, o quarto e assim por diante.
  - A expressão `nth-child(3)` afeta unicamente o terceiro elemento, por não conter `n` na expressão, trata-se de uma constante.

A Listagem 1.28 apresenta um código CSS que exemplifica o uso das pseudo-classes mencionadas. Com a estilização do exemplo, os parágrafos `<p>` têm cor de texto verde, mas, quando o usuário passar o ponteiro do mouse sobre o elemento, ele ficará com a cor do texto vermelha e, ao clicar no elemento, com a cor do texto azul. Enquanto o usuário não estiver realizando nenhuma dessas interações com o elemento, o elemento permanecerá com a cor de texto verde. Além disso, os elementos `<input>` ficarão com a cor de fundo vermelha quando estiverem em foco.

```
p {
    color: green;
}
p:hover {
    color: red;
}
p:active {
    color: blue;
}
input:focus {
    background-color: red;
}
```

Listagem 1.28: Exemplo de uso de pseudo-classes em CSS

Existem várias outras pseudo-classes disponíveis no CSS3, porém, como já mencionado anteriormente, não é do interesse dos autores apresentar neste material todas as possibilidades no CSS, o que engloba as pseudo-classes.

Com o uso das pseudo-classes, que permitem dinamizar o estilo dos elementos, as transições em CSS podem ser úteis para mudanças suaves ou personalizadas no estilo, com um tempo de transição definido. As transições em CSS permitem a animação suave das mudanças nas propriedades de um elemento ao longo do tempo, proporcionando uma experiência mais fluida ao usuário. Para utilizá-las, a propriedade `transition` deve ser definida no elemento, especificando quais propriedades serão animadas, a duração da animação e outros parâmetros.

A Listagem 1.29 apresenta um código CSS que aplica transição à estilização do elemento `<p>` quando o cursor do mouse estiver sobre ele. Enfatiza-se que, no código, a transição é aplicada somente à propriedade `color`, ou seja, somente a cor do texto terá efeito de transição com duração de 1 segundo (definida por "`1s`" no código), enquanto a transição do tamanho da fonte ocorrerá de forma instantânea. Destaca-se também que, como a propriedade `transition` só está definida no `p:hover`, a transição só ocorrerá no momento em que o cursor do mouse ficar sobre o elemento, e não ocorrerá no retorno ao estilo padrão quando o cursor do mouse sair de cima do elemento.

```
p {
    color: black;
    font-size: 1em;
}
p:hover {
    color: red;
    font-size: 2em;
    transition: color 1s;
}
```

Listagem 1.29: Exemplo de transição em CSS

Para que a transição de estilo ocorra no momento do evento da pseudo-classe e também no retorno ao estilo padrão, basta definir a propriedade `transition` no estilo padrão do elemento, pois as pseudo-classes herdarão essa transição, assim como ocorre com as demais definições, se não houver sobrescrição. O código da Listagem 1.30 apresenta um exemplo para esse caso, além de definir a transição para todas as propriedades (definido pelo "`all`" no código).

```
p {
    color: black;
    font-size: 1em;
    transition: all 1s;
}
p:hover {
    color: red;
    font-size: 2em;
}
```

Listagem 1.30: Exemplo de transição em CSS

Caso se deseje aplicar a transição apenas no retorno ao estilo padrão, basta definir a propriedade `transition` como "`none`" na pseudo-classe em que a transição não deve ocorrer.

1.10 Propriedades Básicas em CSS

Esta seção apresenta as principais propriedades de estilização do CSS, explicando sua função e exemplos de valores que podem ser atribuídos a elas.

1.10.1 Propriedades de Texto

- `color`: Define a cor do texto. Exemplos:
  - `color: red;` (cor nomeada)
  - `color: #ff0000;` (cor hexadecimal)
  - `color: rgb(255, 0, 0);` (cor RGB)
  - `color: hsl(0, 100%, 50%);` (cor HSL)
- `font-size`: Define o tamanho da fonte. Exemplos:
  - `font-size: 16px;` (tamanho fixo)
  - `font-size: 120%;` (relativo ao tamanho do pai)
  - `font-size: 1.2em;` (relativo ao elemento pai)
  - `font-size: 1.2rem;` (relativo ao elemento raiz)
- `font-family`: Define a família de fontes. Exemplos:

  - `font-family: Arial, sans-serif;`
  - `font-family: "Times New Roman", serif;`

- `text-align`: Define o alinhamento do texto. Exemplos:
  - `text-align: left;` (esquerda)
  - `text-align: center;` (centralizado)
  - `text-align: right;` (direita)
  - `text-align: justify;` (justificado)

- `font-weight`: Define o peso da fonte. Exemplos:
  - `font-weight: normal;` (normal)
  - `font-weight: bold;` (negrito)
  - `font-weight: lighter;` (mais fino)
  - `font-weight: 700;` (valores numéricos)

1.10.2 Propriedades de Espaçamento

- `margin`: Define a margem externa do elemento. Exemplos:
  - `margin: 10px;` (margem uniforme)
  - `margin: 10px 5px;` (vertical e horizontal)
  - `margin: auto;` (centraliza elementos em layouts flexíveis)

- `padding`: Define o espaçamento interno do elemento. Exemplos:
  - `padding: 20px;` (preenchimento uniforme)
  - `padding: 10px 5px 15px 8px;` (topo, direita, baixo, esquerda)

1.10.3 Propriedades de Layout

- `display`: Controla o tipo de exibição de um elemento. Exemplos:
  - `display: block;` (ocupa toda a largura disponível)
  - `display: inline;` (mantém na mesma linha)
  - `display: flex;` (usa o modelo de layout flexível)
  - `display: grid;` (usa o modelo de layout em grade)
  - `display: none;` (oculta o elemento)

- `position`: Define o posicionamento do elemento na página. Exemplos:
  - `position: static;` (posição padrão)

  - `position: relative;` (relativa à posição original)
  - `position: absolute;` (relativa ao ancestral posicionado)
  - `position: fixed;` (fixo na tela)
  - `position: sticky;` (gruda ao rolar)

- `float`: Define o fluxo do elemento em relação ao conteúdo. Exemplos:
  - `float: left;` (alinha à esquerda)
  - `float: right;` (alinha à direita)
  - `float: none;` (sem efeito de flutuação)

1.10.4 Propriedades de Fundo e Bordas

- `background-color`: Define a cor de fundo do elemento. Exemplos:
  - `background-color: blue;`
  - `background-color: #f5f5f5;`
  - `background-color: transparent;` (transparente)
- `background-image`: Define uma imagem de fundo. Exemplos:
  - `background-image: url("imagem.jpg");`
  - `background-image: none;` (sem imagem de fundo)
- `border`: Define a borda do elemento. Exemplos:
  - `border: 1px solid black;` (linha sólida preta de 1 pixel)
  - `border: 2px dashed red;` (linha tracejada vermelha de 2 pixels)
  - `border: none;` (sem borda)
- `border-radius`: Define o arredondamento dos cantos. Exemplos:
  - `border-radius: 10px;` (cantos arredondados)
  - `border-radius: 50%;` (cria um círculo)

1.10.5 Propriedades de Visibilidade e Opacidade

- `visibility`: Controla a visibilidade do elemento. Exemplos:
  - `visibility: visible;` (visível)
  - `visibility: hidden;` (oculto, mas mantém espaço)
- `opacity`: Define a opacidade do elemento. Exemplos:

- `opacity: 1;` (totalmente visível)
- `opacity: 0.5;` (50% transparente)
- `opacity: 0;` (completamente invisível)

As propriedades do CSS permitem estilizar e controlar o layout dos elementos de forma eficiente. O conhecimento e a aplicação corretos dessas propriedades são fundamentais para o desenvolvimento de interfaces responsivas e esteticamente agradáveis.

Apesar da breve explicação dos valores atribuídos às propriedades apresentados nesta seção, alguns desses valores, como unidades de medida, cores e palavras-chave especiais, serão abordados com mais detalhes na Seção 1.11.

1.11 Valores, Unidades e Cores em CSS

No CSS, propriedades são definidas por meio de valores que podem ser expressos em diferentes formatos, dependendo do tipo de propriedade em questão. Esses valores incluem unidades de medida para dimensões, representações de cores e outros tipos específicos.

1.11.1 Unidades de Medida

As unidades de medida no CSS podem ser divididas em dois grupos principais: absolutas e relativas.

- **Unidades absolutas**: possuem tamanho fixo e não variam conforme o contexto. Exemplos incluem `px` (pixels), `cm` (centímetros), `mm` (milímetros), `in` (polegadas), `pt` (pontos tipográficos) e `pc` (paicas).
- **Unidades relativas**: Seu valor depende do contexto em que são aplicadas. Entre as mais comuns estão:
  - `em`: Relativa ao tamanho da fonte do elemento pai.
  - `rem`: Relativa ao tamanho da fonte do elemento raiz (`html`).
  - `%`: Relativa ao tamanho do elemento pai.
  - `vw` e `vh`: Representam uma fração da largura e altura da janela de exibição, respectivamente.
  - `vmin` e `vmax`: Baseadas na menor e maior dimensão da viewport.

1.11.2 Representação de Cores

As cores no CSS podem ser especificadas de diferentes formas, o que permite flexibilidade no design e na personalização visual das páginas. Os principais formatos são:

- **Nomes de cores**: CSS suporta nomes predefinidos como `red`, `blue`, `green`, `black`, entre outros.
- **RGB (`rgb(r, g, b)`)**: Define cores com valores de 0 a 255 para os canais vermelho, verde e azul.
- **RGBA (`rgba(r, g, b, a)`)**: Igual ao formato RGB, mas inclui um canal de transparência (`a`) com valores entre 0 (transparente) e 1 (opaco).
- **Hexadecimal (`#RRGGBB` ou `#RGB`)**: Representa cores usando valores hexadecimais, como `#ff0000` para vermelho.
- **HSL (`hsl(h, s, l)`)**: Define cores por matiz (*hue*), saturação (*saturation*) e luminosidade (*lightness*). Cada valor pode variar entre 0% e 100%.
- **HSLA (`hsla(h, s, l, a)`)**: Versão do HSL que inclui um canal de transparência.

1.11.3 Valores Especiais

O CSS possui valores especiais que determinam o comportamento de uma propriedade de forma mais ampla:

- `none`: Indica a ausência de um valor ou efeito. É frequentemente usado para remover estilos como bordas, fundos ou animações.
- `inherit`: Força a propriedade a herdar o valor do elemento pai, independentemente do comportamento padrão da propriedade. Útil para manter estilos consistentes.
- `initial`: Restaura a propriedade ao seu valor inicial definido pelo CSS padrão.
- `unset`: Para propriedades herdáveis (como `color`), funciona como `inherit`. Para propriedades não herdáveis (como `display`), age como `initial`.
- `revert`: Similar ao `unset`, mas reverte ao estilo definido pela folha de estilos do usuário ou do navegador, ignorando estilos aplicados manualmente.
- `auto`: Permite que o navegador calcule automaticamente o valor da propriedade com base no contexto. É frequentemente usado em propriedades como `width`, `height`, `margin` e `position`.

O uso adequado desses valores especiais pode simplificar a manutenção do código CSS e permitir maior controle sobre a estilização dos elementos. Eles são especialmente úteis ao trabalhar com temas dinâmicos ou ao redefinir estilos padrão para um comportamento mais previsível.

1.12 *Box Model*

No HTML/CSS, o modelo de caixa (*Box Model*) define como os elementos são representados na página, determinando seu tamanho e espaçamento em relação a outros elementos. Cada elemento é tratado como uma caixa retangular composta por quatro áreas principais:

- ***Content* (Conteúdo)**: A área onde o conteúdo do elemento é exibido, como texto ou imagens. O tamanho dessa área pode ser controlado pelas propriedades `width` e `height`.
- ***Padding* (Preenchimento)**: Espaço entre o conteúdo e a borda do elemento. Definido pelas propriedades `padding`, ele aumenta o tamanho total do elemento sem alterar a área de conteúdo.
- ***Border* (Borda)**: A borda ao redor do elemento, que pode ter diferentes estilos, cores e larguras. Definida pela propriedade `border`.
- ***Margin* (Margem)**: Espaço externo entre a borda do elemento e outros elementos ao seu redor. Definida pela propriedade `margin`, controla o espaçamento entre os elementos na página.

A Figura 1.2 ilustra o layout da *Box Model* em HTML/CSS. A área azul representa o *content*, que tem seu *width* (largura) e *height* (altura), a área verde representa o *padding*, a área amarela representa o *border* e a área marrom representa o *margin*.

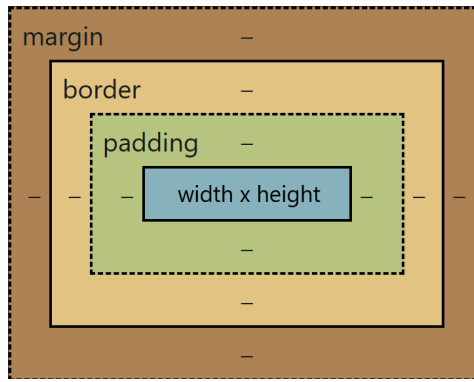


Figura 1.2: Ilustração do *Box Model* em HTML/CSS.

O tamanho total de um elemento no layout é calculado considerando todas as suas áreas. Por exemplo, a largura total do elemento é a soma do: `width`, `padding` (esquerdo e direito), `margin` (esquerdo e direito) e `border` (esquerdo e direito).

Se a propriedade `box-sizing` estiver configurada como `content-box` (padrão), o valor de `width` e `height` corresponde apenas ao conteúdo, e padding e borda são adicionados ao tamanho total. Se estiver configurada como `border-box`, `width` e `height` incluirão padding e borda, facilitando o controle do tamanho do elemento.

O conhecimento acerca do *Box Model* é essencial para uma boa configuração de layout das páginas web, pois permite um controle preciso sobre o espaçamento e a organização dos elementos.

1.13 Layout e Posicionamentos

O layout e posicionamento dos elementos em CSS são fundamentais para a estruturação e design das páginas web. Duas das abordagens mais poderosas para organização de elementos são o *Flexbox* e o Grid Layout. Ambas permitem um controle avançado sobre a distribuição e alinhamento dos elementos na página.

1.13.1 *Flexbox*

O *Flexbox* (*Flexible Box Layout*) é um modelo de layout unidimensional que organiza os elementos em linhas ou colunas, facilitando a distribuição e o alinhamento dos itens dentro de um contêiner flexível.

Para usar o *Flexbox*, o valor `flex` deve ser atribuído à propriedade `display` do elemento pai, que comporta os elementos que serão organizados de acordo com esse layout. Algumas das propriedades principais do contêiner flexível são:

- `flex-direction`: Define a direção principal do eixo flexível. Pode assumir os valores:
  - `row` (padrão) - os itens são organizados em linha.
  - `column` - os itens são organizados em coluna.
  - `row-reverse` - os itens são organizados em linha invertida.
  - `column-reverse` - os itens são organizados em coluna invertida.
- `justify-content`: Define o alinhamento dos itens ao longo do eixo principal (definido no `flex-direction`). Pode assumir os valores:
  - `flex-start`: Alinha os itens ao início do contêiner.
  - `flex-end`: Alinha os itens ao final do contêiner.
  - `center`: Centraliza os itens.
  - `space-between`: Distribui os itens uniformemente, com o primeiro item alinhado à esquerda e o último à direita.
  - `space-around`: Distribui os itens uniformemente, com espaços iguais ao redor deles.

  - `space-evenly`: Distribui os itens com espaçamentos iguais entre eles e nas extremidades.

- `align-items`: Define o alinhamento dos itens ao longo do eixo transversal. Pode assumir os valores:
  - `stretch`: Os itens ocupam toda a altura do contêiner.
  - `flex-start`: Alinha os itens no início do eixo transversal.
  - `flex-end`: Alinha os itens no final do eixo transversal.
  - `center`: Centraliza os itens no eixo transversal.
  - `baseline`: Alinha os itens de acordo com a linha base do texto.

- `flex-wrap`: Controla se os itens devem ou não quebrar linha. Pode assumir os valores:
  - `nowrap`: Os itens permanecem em uma única linha (padrão).
  - `wrap`: Os itens quebram para a próxima linha quando não há espaço suficiente.
  - `wrap-reverse`: Semelhante ao `wrap`, mas invertendo a ordem das linhas.

- `gap`: Define o espaçamento entre os itens do contêiner. Pode ser especificado em pixels "`px`", porcentagem "`%`", `em`, `rem`, etc.

A Listagem 1.31 mostra um código HTML/CSS que apresenta um caso de uso do *Flexbox*, onde os elementos são organizados em linha, alinhados ao fim do eixo transversal do elemento pai e com espaçamento uniforme dentre eles. Esse código resulta na visualização apresentada na Figura 1.3.

```
<style>
    #container-pai {
        width: 500px;
        height: 100px;
        border: solid 3px red;
        padding: 5px;
        display: flex;
        flex-direction: row;
        justify-content: space-between;
        align-items: flex-end;
    }
    .item-filho {
        width: 60px;
        height: 60px;
        background-color: green;
    }
</style>
<div id="container-pai">
    <div class="item-filho"></div>
    <div class="item-filho"></div>
    <div class="item-filho"></div>
    <div class="item-filho"></div>
</div>
```

Listagem 1.31: Uso do *Flexbox* em HTML/CSS

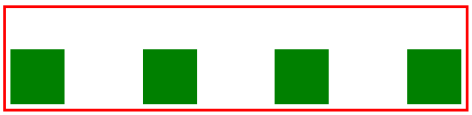


Figura 1.3: Visualização resultante do código HTML/CSS da Listagem 1.31.

Recomenda-se ao leitor experimentar variações do código apresentado na Listagem 1.31 para explorar as diferentes possibilidades de uso do *Flexbox*.

1.13.2 *Grid Layout*

O *Grid Layout* em CSS é um modelo de layout bidimensional que permite organizar elementos em linhas e colunas, facilitando o posicionamento e alinhamento dos itens dentro de um contêiner *grid*.

Para usar o *Grid Layout*, o valor `grid` deve ser atribuído à propriedade `display` do elemento pai, que comporta os elementos que serão organizados de acordo com esse layout. Algumas das propriedades principais do contêiner grid são:

- `grid-template-columns`: Define o número e o tamanho das colunas do grid. Pode assumir valores como:

  - `auto` - Ajusta automaticamente a largura da coluna conforme o conteúdo.
  - Medidas em `px`, `%`, `em`, `rem` - definem tamanhos fixos.
  - `fr` - Unidade flexível que distribui espaço disponível entre as colunas.
  - `repeat(n, tamanho)` - Cria `n` colunas com o tamanho especificado.

- `grid-template-rows`: Define o número e o tamanho das linhas do grid, seguindo os mesmos valores possíveis da propriedade `grid-template-columns`.

- `justify-content`: Define o alinhamento do grid dentro do contêiner pai. Pode assumir os valores:

  - `start` - Alinha o grid ao início do contêiner.
  - `end` - Alinha o grid ao final do contêiner.
  - `center` - Centraliza o grid dentro do contêiner.
  - `space-between` - Distribui as colunas igualmente, com espaços entre elas.
  - `space-around` - Distribui espaço uniformemente ao redor das colunas.

- `align-items`: Define o alinhamento dos itens dentro das células do grid. Pode assumir os valores:

  - `stretch` (padrão) - Expande os itens para preencher a altura da célula.
  - `start` - Alinha os itens ao início da célula.
  - `end` - Alinha os itens ao final da célula.
  - `center` - Centraliza os itens dentro da célula.

- `grid-gap` ou `gap`: Define o espaçamento entre as linhas e colunas do grid. Pode ser especificado em pixels "`px`", porcentagem "`%`", `em`, `rem`, etc.

A Listagem 1.32 apresenta um código HTML/CSS que ilustra um caso de uso do *Grid Layout*, em que os elementos são organizados em um layout de 3 colunas e 2 linhas, com espaçamento de 10px entre os itens. Esse código resulta na visualização apresentada na Figura 1.4.

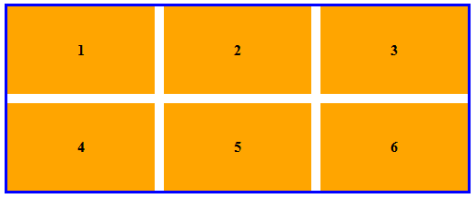


Figura 1.4: Visualização resultante do código HTML/CSS da Listagem 1.32.

```
<style>
    #container-grid {
        width: 500px;
        height: 200px;
        border: solid 3px blue;
        display: grid;
        grid-template-columns: repeat(3, 1fr);
        grid-template-rows: repeat(2, 1fr);
        gap: 10px;
    }
    .item-grid {
        background-color: orange;
        display: flex;
        justify-content: center;
        align-items: center;
        font-weight: bold;
    }
</style>
<div id="container-grid">
    <div class="item-grid">1</div>
    <div class="item-grid">2</div>
    <div class="item-grid">3</div>
    <div class="item-grid">4</div>
    <div class="item-grid">5</div>
    <div class="item-grid">6</div>
</div>
```

Listagem 1.32: Uso do *Grid Layout* em HTML/CSS

Recomenda-se ao leitor experimentar variações do código apresentado na Listagem 1.32 para explorar as diferentes possibilidades de uso do *Grid Layout*.

Flexbox e *Grid Layout* são ferramentas poderosas que permitem a criação de layouts responsivos e organizados, cada um com suas vantagens dependendo do tipo de estrutura desejada. Testar diferentes valores das propriedades apresentadas e observar os resultados na prática contribuirão para uma melhor compreensão e assimilação do modelo de layout.

1.14 *Media Query*

A *Media Query* é um recurso do CSS que permite aplicar estilos diferentes dependendo das características do dispositivo, como largura da tela, resolução e orientação. Isso é essencial para a criação de designs responsivos.

Uma *Media Query* segue a estrutura de um bloco de código CSS, onde é especificada uma condição dentro do seletor `@media`. Se a condição for atendida, os estilos definidos dentro do bloco serão aplicados. A condição pode incluir características como largura da tela, altura, orientação e resolução. O objetivo é adaptar o design de uma página para diferentes dispositivos e contextos de uso. A Listagem 1.33 mostra a sintaxe básica de definição de uma *Media Query*.

```
@media (condição) {
    seletor {
        propriedade: valor;
    }
}
```

Listagem 1.33: Sintaxe básica de *Media Query* em CSS

Por exemplo, supondo que o fundo padrão da página seja branco e deseja-se que a cor de fundo seja azul quando a tela for menor que 600 pixels, pode-se usar uma *Media Query* para isso, assim como mostra a Listagem 1.34.

```
@media (max-width: 600px) {
    body {
        background-color: blue;
    }
}
```

Listagem 1.34: Exemplo de uso de *Media Query* em CSS

Uma única folha de estilo pode conter várias *Media Queries*, permitindo ajustes específicos para diferentes tamanhos de tela e dispositivos. Além disso, dentro de cada *Media Query*, é possível definir múltiplos estilos para vários elementos, garantindo que todos os componentes da página sejam ajustados conforme necessário.

As *Media Queries* podem utilizar diferentes condições para definir regras específicas. Algumas das principais opções são:

- `min-width` e `max-width`: Definem regras baseadas na largura da tela. Exemplos:
  - `@media (min-width: 600px) { ... }` - Aplica estilos quando a largura da tela for maior ou igual a 600 pixels.
  - `@media (max-width: 1200px) { ... }` - Aplica estilos quando a largura da tela for menor ou igual a 1200 pixels.

- `min-height` e `max-height`: Definem regras baseadas na altura da tela. Exemplos:
  - `@media (min-height: 500px) { ... }` - Aplica estilos quando a altura da tela for maior ou igual a 500 pixels.
  - `@media (max-height: 800px) { ... }` - Aplica estilos quando a altura da tela for menor ou igual a 800 pixels.
- `orientation`: Definem regras baseadas na orientação da tela (retrato ou paisagem). Exemplos:
  - `@media (orientation: portrait) { ... }` - Aplica estilos quando o dispositivo estiver em modo retrato.
  - `@media (orientation: landscape) { ... }` - Aplica estilos quando o dispositivo estiver em modo paisagem.
- `resolution`: Define estilos baseados na densidade de pixels. Exemplos:
  - `@media (min-resolution: 300dpi) { ... }` - Aplica estilos para dispositivos com resolução mínima de 300dpi.
  - `@media (max-resolution: 150dpi) { ... }` - Aplica estilos para dispositivos com resolução máxima de 150dpi.
- `aspect-ratio`: Define estilos baseados na proporção da tela. Exemplos:
  - `@media (aspect-ratio: 16/9) { ... }` - Aplica estilos para dispositivos com uma proporção de tela de 16:9.
  - `@media (aspect-ratio: 4/3) { ... }` - Aplica estilos para dispositivos com uma proporção de tela de 4:3.
- `screen`: Aplica estilos para telas de dispositivos (computadores, celulares, etc.). Exemplo:
  - `@media screen { ... }`
- `print`: Aplica estilos para impressão. Exemplo:
  - `@media print { ... }`

Ainda, é possível combinar diferentes condições em *Media Queries* utilizando operadores lógicos, como `and`, `or` e `not`. Isso permite uma maior flexibilidade na aplicação de estilos para cenários mais complexos. O operadores são os seguintes:

- `and`: é usado para combinar duas ou mais condições que devem ser atendidas simultaneamente.
- `or`: permite que qualquer uma das condições seja atendida.

- `not`: inverte uma condição, aplicando os estilos quando a condição não é atendida.

O uso dos operadores facilita a personalização da interface para diferentes tamanhos de tela, orientações, resoluções, entre outros. O código da Listagem 1.35 apresenta exemplos de uso desses operadores para gerar condições mais elaboradas. No código, a primeira regra aplica estilos para telas com largura mínima de 600 pixels e resolução maior que 150dpi, a segunda regra é aplicada quando a largura é menor ou igual a 800 pixels ou quando o dispositivo está em modo retrato, e a última regra é ativada quando a largura da tela não ultrapassa 500 pixels.

```
@media (min-width: 600px) and (min-resolution: 150dpi) {
    body {
        background-color: lightblue;
    }
}

@media (max-width: 800px) or (orientation: portrait) {
    body {
        font-size: 14px;
    }
}

@media not (min-width: 500px) {
    body {
        background-color: lightgrey;
    }
}
```

Listagem 1.35: Exemplo de combinação de condições de *Media Query* em CSS

1.15 Transformações

As transformações em CSS são aplicadas por meio da propriedade `transform`. Com ela, é possível modificar um ou mais aspectos de um elemento, como sua posição, rotação, escala e distorção, sem alterar o layout ou fluxo normal da página. A sintaxe básica para aplicar transformações é a apresentada na Listagem 1.36, onde se evidencia que se pode aplicar uma ou mais transformações a um mesmo elemento, separando-as por espaços.

```
elemento {
    transform: transformacao-1(valor-1)
               transformacao-2(valor-2)
               transformacao-3(valor-3)
               ...;
}
```

Listagem 1.36: Sintaxe básica de transformações em CSS

As transformações podem ser 2D ou 3D, e é possível combinar diferentes tipos de transformações para criar efeitos visualmente interessantes. A propriedade `transform` não afeta o fluxo do documento, ou seja, o elemento continua ocupando o mesmo espaço que teria ocupado sem a transformação, mas sua aparência é alterada. As principais propriedades de transformações são:

- `translate`: Move o elemento ao longo dos eixos X e Y. Exemplos:
  - `transform: translate(50px, 100px);` (move o elemento 50px para a direita e 100px para baixo)
  - `transform: translate(-20px, 0);` (move o elemento 20px para a esquerda)
- `scale`: Altera o tamanho do elemento. Exemplos:
  - `transform: scale(2, 2);` (dobra o tamanho do elemento)
  - `transform: scale(0.5, 1);` (reduz a largura pela metade, mantendo a altura)
- `rotate`: Gira o elemento ao redor de um ponto fixo. Exemplos:
  - `transform: rotate(45deg);` (gira o elemento 45 graus no sentido horário)
  - `transform: rotate(-90deg);` (gira o elemento 90 graus no sentido anti-horário)
- `skew`: Inclina o elemento nos eixos X e Y. Exemplos:
  - `transform: skew(30deg, 0);` (inclina o elemento 30 graus no eixo X)
  - `transform: skew(0, 20deg);` (inclina o elemento 20 graus no eixo Y)
- `perspective`: Adiciona um efeito de profundidade 3D. Exemplos:
  - `perspective: 500px;` (define a profundidade em 500 pixels)
  - `perspective: 100px;` (efeito de profundidade mais acentuado)
- `rotateX, rotateY, rotateZ`: Gira o elemento em 3D. Exemplos:
  - `transform: rotateX(45deg);` (gira o elemento 45 graus ao redor do eixo X)
  - `transform: rotateY(90deg);` (gira o elemento 90 graus ao redor do eixo Y)
  - `transform: rotateZ(30deg);` (gira o elemento 30 graus ao redor do eixo Z)
- `translateZ`: Move o elemento ao longo do eixo Z. Exemplos:

  - `transform: translateZ(100px);` (move o elemento 100px para frente)
  - `transform: translateZ(-50px);` (move o elemento 50px para trás)

- `matrix`: Aplica uma transformação 2D complexa. Exemplos:
  - `transform: matrix(1, 0, 0, 1, 50, 100);` (equivalente a `translate(50px, 100px)`)
  - `transform: matrix(0.5, 0, 0, 0.5, 0, 0);` (reduz o elemento pela metade)

- `matrix3d`: Aplica uma transformação 3D complexa. Exemplos:
  - `transform: matrix3d(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 100, 200, 300, 1);` (move o elemento para `x = 100`, `y = 200`, `z = 300`)
  - `transform: matrix3d(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1);` (não aplica nenhuma transformação, identidade)

Essas transformações podem ser combinadas entre si, permitindo criar efeitos complexos e dinâmicos na interface do usuário. Além disso, é possível aplicar transições para suavizar os efeitos, como mostra a Listagem 1.37. No código, a classe `.box` define um quadrado azul de 100 pixels por 100 pixels que, quando o usuário passa o mouse sobre ele (`:hover`), ele aumenta de tamanho em 20% (`scale(1.2)`) e gira 15 graus (`rotate(15deg)`), criando um efeito dinâmico e interativo.

```
.box {
    width: 100px;
    height: 100px;
    background-color: blue;
    transition: transform 0.3s;
}

.box:hover {
    transform: scale(1.2) rotate(15deg);
}
```

Listagem 1.37: Exemplo de uso de transformações em CSS

1.16 Animações

As animações em CSS permitem criar efeitos dinâmicos e personalizados. Elas são definidas por meio da propriedade `@keyframes`, que especifica as etapas da animação, e da propriedade `animation`, que aplica a animação a um elemento.

Para criar uma animação, primeiro definem-se os estados intermediários usando `@keyframes`, como se pode visualizar na Listagem 1.38. No exemplo, uma animação nomeada como "exemplo" é criada e atribuída aos elementos `<div>`. O código define a seguinte animação: o elemento se move 50 pixels para a direita e retorna à posição inicial em um ciclo de 2 segundos repetindo infinitamente.

```
@keyframes exemplo {
    0% { transform: translateX(0px); }
    50% { transform: translateX(50px); }
    100% { transform: translateX(0px); }
}

div {
    animation: exemplo 2s infinite;
}
```

Listagem 1.38: Exemplo de definição de animação em CSS

As animações, por meio dos `keyframes`, podem ser definidas de duas formas: usando porcentagens para definir etapas intermediárias da animação ou usando as palavras "`from`" e "`to`", que representam 0% e 100% respectivamente. A Listagem 1.39 apresenta dois exemplos de definição de animação: a animação nomeada "fadeIn" usa os valores "`from`" e "`to`" para criar um efeito de transição entre a transparência total e a opacidade total, enquanto a animação nomeada "colorChange" usa porcentagens para definir estados intermediários, criando um efeito de transição de cores.

```
@keyframes fadeIn {
    from { opacity: 0; }
    to { opacity: 1; }
}
@keyframes colorChange {
    0% { background-color: red; }
    33% { background-color: yellow; }
    66% { background-color: blue; }
    100% { background-color: red; }
}
```

Listagem 1.39: Exemplos de duas formas diferentes de definir animação em CSS

Como já visto no código-exemplo da Listagem 1.38, uma animação pode ser aplicada a um elemento usando a propriedade `animation` que pode receber vários parâmetros para diferentes variáveis acerca da animação, como duração, nome da animação, *delay*, etc. Porém, é possível realizar a mesma definição em partes com propriedades mais específicas referentes a animação. As propriedades de animação são as seguintes:

- `animation-name`: Nome da animação definida em `@keyframes`.
- `animation-duration`: Duração total da animação.

- `animation-timing-function`: Define a curva de aceleração (ex: `ease`, `linear`, `ease-in`, etc.).
- `animation-delay`: Tempo antes da animação começar.
- `animation-iteration-count`: Número de repetições (ex: `infinite` para repetição contínua).
- `animation-direction`: Define a direção da animação (ex: `normal`, `reverse`, `alternate`).

1.17 Exemplo Prático Básico

Nesta seção, será desenvolvido um projeto prático que demonstra a aplicação de alguns conceitos abordados no capítulo. O projeto consiste em uma página web simples contendo um cabeçalho, uma seção de conteúdo principal e um rodapé. Serão utilizadas *tags* HTML essenciais, propriedades de CSS para estilização, *Flexbox* e *Grid Layout* para organização do layout, pseudo-classes para interação com o usuário e animações para melhorar a experiência visual.

Primeiramente, cria-se um arquivo nomeado "index.html", responsável por conter a estrutura da página. Esse documento é construído seguindo a marcação apresentada na Listagem 1.40.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale
        =1.0">
    <title>Projeto Exemplo</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <header>
        <h1>Bem-vindo ao Projeto Exemplo</h1>
    </header>
    <main>
        <section class="container">
            <article class="card">Conteúdo 1</article>
            <article class="card">Conteúdo 2</article>
            <article class="card">Conteúdo 3</article>
        </section>
    </main>
    <footer>
        <p>&copy; 2025 Projeto Exemplo</p>
    </footer>
</body>
</html>
```

Listagem 1.40: Código HTML da página do projeto-exemplo

Para a estilização da página, cria-se um arquivo chamado “style.css” (conforme especificado na *tag* de `<link>` presente no cabeçalho do arquivo HTML), no qual se definem os estilos CSS. O arquivo deve estar localizado no mesmo diretório do arquivo “index.html”.

A fim de remover margens e preenchimentos padrão dos navegadores, pode-se aplicar um *reset* de estilos, como apresentado na Listagem 1.41, o que garante um comportamento consistente dos elementos.

```
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}
```

Listagem 1.41: Código CSS da página do projeto-exemplo (Parte 1)

Para estilizar o corpo da página e melhorar a leitura, usa-se um fundo neutro e um espaçamento adequado entre as linhas de texto, como mostrado no código da Listagem 1.42.

```
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
    color: #333;
    line-height: 1.6;
}
```

Listagem 1.42: Código CSS da página do projeto-exemplo (Parte 2)

De acordo com a Listagem 1.43, o cabeçalho terá um fundo azul, texto branco e um menu de navegação com os itens dispostos horizontalmente no centro da tela. O efeito visual é de um menu limpo e organizado.

```css
header {
    background-color: #007bff;
    color: white;
    text-align: center;
    padding: 20px;
}

nav ul {
    list-style: none;
    display: flex;
    justify-content: center;
    gap: 20px;
}

nav a {
    text-decoration: none;
    color: white;
    font-weight: bold;
    transition: color 0.3s;
}

nav a:hover {
    color: yellow;
}
```

Listagem 1.43: Código CSS da página do projeto-exemplo (Parte 3)

O código da Listagem 1.44 cria a estilização de um banner de destaque, que aparece suavemente com uma animação ao carregar a página. Ele se posiciona no centro e se destaca do restante do conteúdo.

```css
.banner {
    background-color: #ff9800;
    color: white;
    text-align: center;
    padding: 50px;
    animation: fadeIn 2s ease-in-out;
}

@keyframes fadeIn {
    from {
        opacity: 0;
        transform: translateY(-20px);
    }
    to {
        opacity: 1;
        transform: translateY(0);
    }
}
```

Listagem 1.44: Código CSS da página do projeto-exemplo (Parte 4)

Para exibir múltiplos elementos de forma responsiva, use-se, na Listagem 1.45, um layout de grade. Os itens se ajustam automaticamente ao tamanho da tela, garantindo um alinhamento fluido.

```
.container {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
    gap: 20px;
    padding: 20px;
    max-width: 800px;
    margin: auto;
}
```

Listagem 1.45: Código CSS da página do projeto-exemplo (Parte 5)

Os cartões dentro da grade têm um fundo branco, sombras suaves e um efeito de ampliação quando o usuário passa o mouse, criando uma sensação interativa. O código que apresenta essa definição de estilo é apresentado na Listagem 1.46.

```
.card {
    display: flex;
    align-items: center;
    justify-content: center;
    background-color: white;
    padding: 20px;
    border-radius: 10px;
    box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
    text-align: center;
    transition: transform 0.3s;
    animation: slideIn 1s ease-in-out;
}

.card:hover {
    transform: scale(1.05);
}
```

Listagem 1.46: Código CSS da página do projeto-exemplo (Parte 6)

A estilização do rodapé da página é apresentada na Listagem 1.47. O rodapé da página recebe um fundo escuro com texto branco, garantindo um bom contraste e legibilidade. O conteúdo do rodapé é centralizado horizontalmente e o espaçamento interno (*padding*) fornece um espaço confortável para o texto, evitando que ele fique colado às bordas, enquanto a margem superior separa o rodapé do conteúdo anterior.

```css
footer {
    background-color: #333;
    color: white;
    text-align: center;
    padding: 10px;
    margin-top: 20px;
}
```

Listagem 1.47: Código CSS da página do projeto-exemplo (Parte 7)

A Figura 1.5 apresenta o visual da página do projeto exemplo prático desenvolvido nessa seção. Este exemplo simples permite visualizar na prática o uso de conceitos essenciais de HTML e CSS, fornecendo uma base para projetos mais avançados.

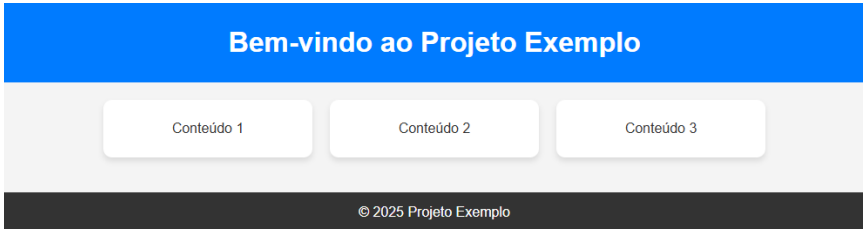


Figura 1.5: Visual da página do projeto exemplo prático desenvolvido na seção.

Capítulo 2

JavaScript e TypeScript

Jadson Faustino Mororó (UFC)

Erick Aguiar Donato (UFC)

Iális Cavalcante de Paula Júnior (UFC)

Fischer Ferreira (Unifei)

Este capítulo apresenta uma introdução prática e objetiva às linguagens *JavaScript* (JS) e *TypeScript* (TS), destacando suas principais características e como elas são aplicadas no desenvolvimento web. O conteúdo é dividido em três partes para facilitar a compreensão:

- **Visão Geral:** Aqui, você conhecerá os fundamentos de *JavaScript* e *TypeScript*, suas diferenças e vantagens. Esses conceitos são abordados nas Seções 2.1 e 2.2.
- **Sintaxe e Funcionalidades:** Esta parte aborda a estrutura das linguagens de forma prática. Serão abordados tipos de dados, variáveis, operações matemáticas e manipulação de *strings* (Seções 2.3 a 2.7). Em seguida, exploramos estruturas essenciais como condicionais (`if/else`, `switch/case`) e laços de repetição (`for`, `while`) de forma integrada (Seções 2.8 e 2.9). Também veremos funções e suas variações, como *arrow functions* (Seção 2.10), além do tratamento de exceções e a programação assíncrona com *callbacks*, *Promises* e `async/await` (Seções 2.11, 2.12 e 2.13). Para organização de dados, abordamos *arrays* e coleções chaveadas (Seções 2.14 e 2.15). Por fim, introduzimos classes, herança e interfaces (Seções 2.16 e 2.17).
- **Aplicação Prática:** Para consolidar os conceitos, a Seção 2.18 apresenta um exemplo completo: a implementação de um jogo *Flappy Bird* em *JavaScript* e *TypeScript*.

2.1 Visão Geral sobre *JavaScript*

O *JavaScript* é uma linguagem de programação de alto nível e multiparadigma, suportando programação orientada a objetos, funcional e imperativa incluído orientada a objetos, amplamente utilizada no desenvolvimento de aplicações web. Originalmente criada para fornecer interatividade aos sites, permitindo a manipulação dinâmica do conteúdo da página e a resposta a eventos do usuário, o *JavaScript* evoluiu ao longo dos anos e se tornou uma linguagem versátil, com suporte a programação assíncrona e desenvolvimento tanto no cliente quanto no servidor (NETWORK, 2023a).

A história do *JavaScript* remonta à década de 1990, quando foi criado por Brendan Eich na Netscape Communications Corporation. Inicialmente chamada de *LiveScript*, a linguagem foi renomeada para *JavaScript* para capitalizar a popularidade da linguagem *Java* na época. Em 1997, a Ecma International lançou a especificação ECMAScript, que é a padronização oficial da linguagem. Desde então, várias versões do ECMAScript foram lançadas, introduzindo melhorias e novos recursos para o *JavaScript* (W3SCHOOLS, 2023a; CONTRIBUTORS, 2021).

O *JavaScript* é uma linguagem interpretada, o que significa que o código-fonte é executado diretamente por um interpretador em tempo de execução, sem a necessidade de compilação prévia. Os navegadores modernos incorporam motores



Figura 2.1: Código JavaScript sendo interpretado pelo navegador Web

JavaScript, como o V8, *SpiderMonkey* e *JavaScriptCore*, para executar o código *JavaScript* (CONTRIBUTORS, 2021). Na Figura 2.1 é possível visualizar um exemplo de um código JavaScript implementado a uma página HTML sendo interpretado por um navegador web.

Uma característica importante do *JavaScript* é a tipagem dinâmica, que dispensa a declaração explícita dos tipos de variáveis. Variáveis em *JavaScript* podem armazenar valores de diferentes tipos, como números, strings, objetos e funções. Além disso, o *JavaScript* suporta programação assíncrona por meio de recursos como chamadas de retorno (*callbacks*), *Promises* e a adição mais recente do async/await, que permitem a execução de tarefas demoradas em segundo plano, sem bloquear a execução do programa (DEVPLENO, 2017; NETWORK, 2023b; NETWORK, 2023c).

O *JavaScript* desempenha um papel fundamental no desenvolvimento de páginas da web e tornou-se uma linguagem padrão. Sua importância reside no fato de que o *JavaScript* é responsável por tornar as páginas web dinâmicas, interativas

e funcionais. Assim, é possível adicionar recursos como validação de formulários, animações, manipulação do *DOM* (Modelo de Objeto de Documento) e interações em tempo real com o usuário (CONTENT, 2019).

Além do desenvolvimento *front-end* de sites e aplicações web, o *JavaScript* é amplamente adotado em *frameworks* e bibliotecas populares, como *React.js*, *Angular.js*, *Vue.js* e *Node.js*. Esses frameworks possibilitam a construção de aplicativos web complexos, tanto no lado do cliente quanto no servidor, além de fornecerem uma estrutura robusta para o desenvolvimento de interfaces de usuário sofisticadas e responsivas. O *JavaScript* também é amplamente utilizado no desenvolvimento de jogos, aplicativos móveis híbridos e até no desenvolvimento de servidores com o *Node.js* (CONTRIBUTORS, 2021).

O *Node.js* é uma versão do *JavaScript* que se destaca por sua aplicação no desenvolvimento *back-end*. Enquanto o *JavaScript* é amplamente reconhecido como a linguagem de programação padrão para páginas web, responsável por torná-las dinâmicas e interativas, o *Node.js* estende o uso dessa linguagem para o lado do servidor. Essa expansão permitiu que os desenvolvedores utilizassem o *JavaScript* em todas as camadas de uma aplicação, unificando a linguagem e facilitando o compartilhamento de código entre o *front-end* e o *back-end*.

2.1.1 Tipagem Dinâmica

Uma das características mais distintivas do *JavaScript* é a tipagem dinâmica. Diferentemente de linguagens com tipagem estática, como *C++* ou *Java*, em que os tipos de variáveis devem ser declarados explicitamente, o *JavaScript* permite que as variáveis armazenem valores de diferentes tipos em momentos distintos durante a execução do programa. Essa flexibilidade da tipagem dinâmica pode ser uma vantagem, pois simplifica o desenvolvimento, permitindo que variáveis sejam reutilizadas para diferentes propósitos sem a necessidade de redeclaração ou conversões explícitas de tipo. Por exemplo, uma variável pode armazenar um número em um momento e, em seguida, armazenar uma string ou um objeto posteriormente.

Em contrapartida, a tipagem dinâmica também pode apresentar desafios, especialmente em projetos grandes e complexos, nos quais o controle estrito dos tipos pode ser desejado para evitar erros. É importante ter cuidado ao lidar com tipos em *JavaScript* para garantir que as operações sejam realizadas corretamente e que os valores sejam coerentes com o esperado. E como solução a esta problemática com a linguagem, surgiu o *TypeScript*.

2.1.2 Programação Assíncrona

Outra característica importante do *JavaScript* é o suporte à programação assíncrona. Isso permite que o código seja executado de forma não sequencial, de modo que tarefas demoradas ou bloqueantes possam ser executadas em segundo plano, enquanto outras operações continuam. Antes dos recursos assíncronos do

JavaScript, as chamadas de função que envolviam operações demoradas, como solicitações de rede, poderiam bloquear a execução do programa até que a operação fosse concluída. Isso poderia levar a uma experiência ruim para o usuário, pois a interface do usuário poderia parecer congelada até que a operação fosse concluída.

No entanto, com a introdução de recursos como chamadas de retorno (*callbacks*), *Promises* e async/await, o *JavaScript* oferece mecanismos poderosos para lidar com operações assíncronas. As chamadas de retorno permitem que uma função seja passada como argumento para outra função, a ser executada quando uma determinada operação for concluída. As *Promises* são objetos que representam o resultado futuro de uma operação assíncrona, permitindo que o código se organize de forma mais legível e evitando a ocorrência de "*callback hell*" (aninhamento excessivo de chamadas de retorno). O *async/await* é uma adição mais recente à linguagem que simplifica ainda mais a programação assíncrona, permitindo que o código assíncrono seja escrito de forma mais semelhante à programação síncrona, o que facilita a compreensão e a manutenção do código.

Esses recursos de programação assíncrona são especialmente úteis no contexto de aplicações web, já que muitas vezes é necessário realizar operações demoradas, como fazer solicitações a APIs externas, buscar dados em bancos de dados ou processar grandes volumes de dados. O *JavaScript* oferece ferramentas poderosas para lidar com essas situações, permitindo que as aplicações sejam responsivas e mantenham uma boa experiência para o usuário.

2.1.3 Ecossistema do *JavaScript*

O *JavaScript* possui um ecossistema vibrante, com uma grande quantidade de bibliotecas, *frameworks* e ferramentas disponíveis para facilitar o desenvolvimento de aplicações web. Além disso, o *JavaScript* é suportado por todos os principais navegadores modernos, tornando-se uma linguagem universalmente aceita para o desenvolvimento *front-end*. Existem muitos *frameworks* populares construídos com *JavaScript*, que fornecem estruturas e padrões para o desenvolvimento web. Alguns dos mais conhecidos são:

- ***React.js***[1]: Um *framework JavaScript* mantido pelo Facebook, que é amplamente utilizado para construir interfaces de usuário interativas. O *React.js* utiliza um modelo de componentes reutilizáveis, que simplifica o desenvolvimento e a manutenção de interfaces complexas.

- ***Angular.js***[2]: Um *framework JavaScript* mantido pelo Google, que permite a criação de aplicações web escaláveis e de alto desempenho. O *Angular.js* utiliza o conceito de *data binding*, no qual as alterações nos dados são automaticamente refletidas na interface do usuário, simplificando a sincronização entre a lógica e a apresentação.

[1] **React.js:** https://react.dev/
[2] **Angular.js:** https://angular.io/

- ***Vue.js***[3]: Um *framework JavaScript* progressivo e de fácil aprendizado, que permite a construção de interfaces de usuário interativas e reativas. O *Vue.js* oferece uma sintaxe intuitiva e flexível, permitindo que os desenvolvedores criem componentes reutilizáveis e construam aplicações de forma incremental.

- ***Node.js***[4]: Uma plataforma de tempo de execução *JavaScript* construída sobre o mecanismo V8 do Google Chrome. O *Node.js* permite a execução de código *JavaScript* no lado do servidor, o que abre um mundo de possibilidades para o desenvolvimento web. Com o *Node.js*, é possível criar servidores web, APIs RESTful, microsserviços e muito mais.

Esses são apenas alguns exemplos do vasto ecossistema *JavaScript*. Existem inúmeras outras bibliotecas e ferramentas disponíveis, cada uma com suas características e casos de uso específicos. O *JavaScript* continua evoluindo rapidamente, com novas versões do ECMAScript sendo lançadas regularmente, trazendo melhorias e recursos adicionais para a linguagem.

2.2 Visão Geral sobre *TypeScript*

O *TypeScript* é uma linguagem de programação desenvolvida pela Microsoft que se baseia no *JavaScript* padrão, adicionando recursos de tipagem estática e outros recursos avançados. Foi projetada para solucionar algumas das limitações do *JavaScript*, oferecendo aos desenvolvedores uma experiência de desenvolvimento mais robusta e segura. Assim como o *JavaScript*, o *TypeScript* é essencial para o desenvolvimento web, cada um com vantagens únicas para diferentes cenários, tanto no *front-end* quanto no *back-end*. Ele compartilha a mesma sintaxe básica e a mesma estrutura de programação do *JavaScript*, mas introduz tipos estáticos opcionais. Essa adição permite que os desenvolvedores especifiquem o tipo de cada variável, parâmetro e retorno de função, trazendo benefícios em termos de detecção de erros em tempo de compilação e ferramentas de autocompletar mais poderosas.

O *TypeScript* foi lançado como um projeto de código aberto em 2012, com base na especificação *ECMAScript*, a mesma base do *JavaScript*. Ele transpila para *JavaScript*, o que significa que o código escrito em *TypeScript* é traduzido para *JavaScript* e executado em qualquer ambiente compatível com *JavaScript*. O *TypeScript* é amplamente adotado em projetos de grande escala, nos quais a manutenção e a escalabilidade são essenciais. *Frameworks* populares, como *Angular.js*, adotam o *TypeScript* como sua linguagem principal, aproveitando os recursos adicionais fornecidos pela linguagem para facilitar o desenvolvimento de aplicativos web robustos.

A Figura 2.2 ilustra um exemplo de transpilação de código *TypeScript* para código *JavaScript*. No código do arquivo `exemplo.ts` é possível observar a declaração

[3]**Vue.js:** https://vuejs.org/
[4]**Node.js:** https://nodejs.org/en

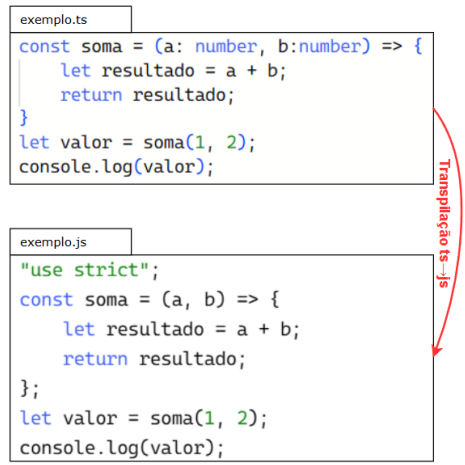


Figura 2.2: Exemplo de Compilação de JavaScript para TypeScript

de uma função `somar`, que especifica os tipos dos parâmetros da função, sendo eles dois valores do tipo `number`, e a função, por sua vez, retorna a soma desses valores. Já no código do arquivo `exemplo.js`, resultado da traspilaço do *TypeScript*, apresenta o mesmo código, contudo, sem os tipos estáticos definidos. Além disso, o compilador acrescenta na primeira linha de código o comando "`use strict`", que impede que variáveis sejam usadas sem serem declaradas previamente em *JavaScript* (W3SCHOOLS, 2023b).

Uma das principais vantagens do *TypeScript* é a detecção de erros em tempo de transpilação. Ao adicionar tipos estáticos ao código, o compilador do *TypeScript* pode identificar erros, como atribuições de tipos incorretas, chamadas de função inválidas e propriedades inexistentes. Essa verificação estática ajuda a reduzir erros comuns e permite que os desenvolvedores identifiquem problemas antes mesmo de executarem o código. Além da tipagem estática, o *TypeScript* oferece recursos avançados, como interfaces, herança de classe, classes abstratas, módulos e uma variedade de recursos adicionais para facilitar o desenvolvimento de aplicativos complexos. Ele também suporta as últimas atualizações do ECMAScript, permitindo que os desenvolvedores utilizem as funcionalidades mais recentes do *JavaScript*.

2.2.1 Benefícios da Tipagem Estática

A adição de tipos estáticos ao *JavaScript* no *TypeScript* traz diversos benefícios para o desenvolvimento de software. Alguns dos principais benefícios incluem:

- **Detecção de erros em tempo de transpilação**: Com a tipagem estática, o compilador do *TypeScript* é capaz de identificar erros no código antes mesmo de ser executado. Isso ajuda a reduzir erros comuns, como atribuições

incorretas de tipos e chamadas de função inválidas, proporcionando um código mais confiável e menos propenso a erros em tempo de execução.

- **Melhor autocompletar e recursos de IDE**: A especificação de tipos no *TypeScript* permite que as IDEs forneçam recursos de autocompletar mais poderosos e informações sobre os tipos esperados. Isso aumenta a produtividade do desenvolvedor, reduzindo o tempo gasto com pesquisas de documentação e evitando erros de digitação.

- **Refatoração segura**: Com a tipagem estática, é possível refatorar código com mais segurança. O compilador pode identificar e alertar sobre alterações que possam afetar outras partes do código, facilitando a manutenção e a evolução do projeto.

- **Documentação mais clara**: A especificação dos tipos no código-fonte torna a documentação do código mais clara e autoexplicativa. Os tipos fornecem informações valiosas sobre os parâmetros e os retornos das funções, tornando o código mais legível e compreensível para outros desenvolvedores.

- **Colaboração facilitada**: A tipagem estática facilita a colaboração entre os membros da equipe, especialmente em projetos de grande porte. Com os tipos definidos explicitamente, fica mais fácil entender as intenções do código e evitar conflitos entre diferentes partes do projeto.

- **Melhor escalabilidade**: O *TypeScript* facilita o desenvolvimento de projetos em larga escala de forma mais eficiente. Com a tipagem estática, é possível ter uma visão mais clara das dependências e das interações entre os diferentes componentes do sistema, facilitando a manutenção e a evolução do código à medida que o projeto cresce.

Desde 2012, o *TypeScript* tem sido amplamente adotado por grandes corporações devido à sua tipagem estática e compatibilidade com JavaScript, além de ser manutenível e escalável. Esses benefícios são alcançados graças às características da linguagem, como a tipagem estática e os recursos de orientação a objetos, que permitem aos desenvolvedores detectar erros de forma antecipada, melhorar a clareza e legibilidade do código e facilitar sua manutenção ao longo do tempo. Além disso, o *TypeScript* oferece suporte a ferramentas de desenvolvimento poderosas, como autocompletar e refatoração, que ajudam a acelerar o processo de desenvolvimento e garantir uma base de código mais robusta. Com essas vantagens, o *TypeScript* se tornou a escolha preferida de muitas equipes de desenvolvimento que buscam construir sistemas mais confiáveis e escaláveis.

2.2.2 Ecossistema *TypeScript*

Assim como o *JavaScript*, o *TypeScript* também possui um ecossistema ativo e diversificado, com uma ampla gama de bibliotecas e *frameworks* disponíveis para

facilitar o desenvolvimento de aplicações web. Muitas bibliotecas e *frameworks* populares do *JavaScript* são compatíveis com o *TypeScript*, permitindo que os desenvolvedores aproveitem recursos adicionais de tipagem estática. Além disso, o *TypeScript* é suportado por uma série de ferramentas de desenvolvimento, como IDEs (Ambientes de Desenvolvimento Integrado) e editores de código, que oferecem recursos avançados para ajudar os desenvolvedores a escrever, depurar e refatorar código *TypeScript*. Algumas das ferramentas populares para desenvolvimento *TypeScript* incluem:

- ***Visual Studio Code***[5]: Um editor de código gratuito e de código aberto desenvolvido pela Microsoft, com suporte nativo ao *TypeScript*. O *Visual Studio Code* oferece recursos avançados, como autocompletar, depuração integrada, integração com controle de versão e extensibilidade por meio de plugins.
- ***WebStorm***[6]: Uma IDE poderosa para o desenvolvimento web, que oferece suporte ao *TypeScript* e a outras tecnologias web. O *WebStorm* fornece recursos avançados de refatoração, depuração e análise de código, ajudando os desenvolvedores a escrever código *TypeScript* de forma eficiente.
- ***TypeScript Playground***[7]: Uma ferramenta online fornecida pela equipe do *TypeScript*, que permite experimentar e testar código *TypeScript* diretamente no navegador. O *TypeScript Playground* oferece recursos de autocompletar, verificação de erros em tempo real e visualização do *JavaScript* gerado pelo código *TypeScript*.
- ***Deno***[8]: Um runtime *JavaScript*/*TypeScript* seguro e moderno, construído com base no mecanismo V8 do Google Chrome. O Deno permite executar código *TypeScript* diretamente, sem a necessidade de compilação prévia, e oferece recursos avançados, como módulos nativos, controle de acesso a recursos do sistema e suporte a importação de módulos diretamente do navegador.

Esses são apenas alguns exemplos do vasto ecossistema *TypeScript*. Existem inúmeras outras bibliotecas e ferramentas disponíveis, cada uma com suas características e casos de uso específicos. O *TypeScript* continua evoluindo rapidamente, com novas versões lançadas regularmente, trazendo melhorias e recursos adicionais à linguagem.

2.2.3 Instalando e Usando *TypeScript*

Para usufruir da tipagem estática e outros recursos avançados do *TypeScript* no desenvolvimento do projeto é necessário instalá-lo, utilizando o `npm` (*Node Package*

[5]https://code.visualstudio.com/
[6]https://www.jetbrains.com/pt-br/webstorm/
[7]https://www.typescriptlang.org/play
[8]https://deno.com/runtime

Manager), que é o gerenciador de pacotes do Node.js. Para usar o `npm` basta baixar e instalar o Node.js, pois ele já vem incluído.

Para instalar o *TypeScript* usando o `npm`, usa-se os comandos apresentados na Listagem 2.1. O primeiro comando é utilizado para instalá-lo globalmente, já o segundo para a adicioná-lo ao projeto como dependência de desenvolvimento.

```
1 npm install -g typescript
2 npm install --save-dev typescript
```

Listagem 2.1: Comandos de instalação do *TypeScript*.

Para iniciar um projeto *TypeScript* usa-se o comando apresentado da Listagem 2.2. Esse comando gera um arquivo chamado `tsconfig.json`, onde pode-se definir configurações como o diretório de saída, nível de rigor e suporte a módulos.

```
1 npx tsc --init
```

Listagem 2.2: Comando de inicialização de projeto *TypeScript*.

Para transpilar arquivos *JavaScript* (.js) para *TypeScript* (.ts) utiliza-se o comando mostrado na Listagem 2.3, onde `arquivo.ts` é o caminho do arquivo. Se o objetivo for transpilar todo o projeto basta omitir o caminho do arquivo.

```
1 npx tsc arquivo.ts
```

Listagem 2.3: Comando de transpilação de arquivos *TypeScript* (.ts) para *JavaScript* (.js).

Utilizando Node.js, é possível executar arquivos *TypeScript* sem precisar transpilá-los manualmente. Basta instalar o `ts-node` globalmente, ou localmente no projeto, como já apresentado na Listagem 2.1, porém, ao invés de `typescript` é `node-ts`. Uma vez feito isso, pode-se usar o comando da Listagem 2.4 para executar o arquivo em *TypeScript*.

```
1 ts-node arquivo.ts
```

Listagem 2.4: Execução direta de arquivo *TypeScript*.

2.3 Funcionamento

O *JavaScript* (JS) é uma linguagem interpretada, o que significa que os programas codificados nessa linguagem são executados por um interpretador em tempo de execução, sem a necessidade de compilação prévia para linguagem de máquina. Por outro lado, o *TypeScript* é uma linguagem compilada, mas não para linguagem

de máquina. Os arquivos *TypeScript*, de extensão ".ts", passam por um processo de compilação para *JavaScript*, gerando arquivos de extensão ".js", derivados dos arquivos *TypeScript*. Somente então esses arquivos *JavaScript* podem ser executados (Cubos Academy, 2022; STX Next, 2022).

O *TypeScript* (TS) utiliza um compilador (tsc) para transformar o código *TypeScript* em um código *JavaScript* equivalente. Durante o processo de compilação, o compilador verifica a sintaxe do código TS, realiza análise estática de tipos e converte o código em *JavaScript* válido. Esse código JS resultante pode ser executado em qualquer ambiente compatível com *JavaScript*, como navegadores web, servidores *Node.js* e outros ambientes de tempo de execução (CHERNY, 2019).

2.4 Tipos de Dados e Variáveis

Os tipos de dados referem-se ao tipo de informações que uma linguagem consegue operar primitivamente, ou seja, ao que é representável computacionalmente no contexto da linguagem de programação (Embarcados, 2022; Wikipédia, 2022a). Já as variáveis são espaços de memória alocados para armazenar esses tipos de dados. É por meio das variáveis que se manipulam os dados na programação (Gaea, 2022; Wikipédia, 2022b).

2.4.1 Declaração

Para o programa reconhecer uma variável, precisa-se declará-la, ou seja, dar o comando de alocar um espaço da memória do computador e nomeá-lo para que se possa armazenar e consultar dados nesse espaço de memória no decorrer do programa. O nome dado a esse trecho de memória é o nome da variável, também chamado de identificador (Wikipédia, 2022b; Gaea, 2022).

Para declarar uma variável em *JavaScript* e *TypeScript* é bem simples, como mostra a Listagem 2.5. Observa-se que o comando inicia com a palavra reservada `let`, que indica a declaração de uma variável, e logo em seguida, consta o nome da variável (não pode conter caracteres especiais). No caso do TS, em seguida, é possível inserir dois pontos e especificar o tipo de dado da variável declarada. De forma geral, pode-se substituir "`minhaVariavel`" por qualquer nome de variável (contanto que não contenha caracteres especiais nem comece com um número) e "`number`" por qualquer tipo de dado disponível na linguagem.

```
1 let minhaVariavel;          // Em JavaScript
2 let minhaVariavel: number;  // Em TypeScript
```

Listagem 2.5: Exemplo de Declaração de Variável em JS e TS.

2.4.2 Constantes e Variáveis

A distinção entre constantes e variáveis é fundamental na programação. As constantes são valores imutáveis que não sofrem alterações ao longo do programa, o que proporciona consistência e confiabilidade ao código. Essas constantes são frequentemente utilizadas para representar informações fixas, como números específicos, configurações ou valores que não devem ser modificados. Por outro lado, as variáveis são utilizadas para armazenar informações que podem ser alteradas durante a execução do programa. Elas permitem a flexibilidade de manipular e atualizar valores à medida que o programa avança. As variáveis são úteis quando se precisa lidar com dados que podem ser modificados, como contadores, resultados de cálculos ou entradas fornecidas pelo usuário. A distinção entre constantes e variáveis permite aos programadores controlar o comportamento do código de acordo com as necessidades do programa, garantindo assim um melhor gerenciamento dos dados e uma maior flexibilidade na manipulação dos valores.

A sintaxe geral para a declaração de constantes é ilustrada na Listagem 2.6. No caso do *JavaScript*, a declaração simplesmente consiste em utilizar a palavra `const` seguida do nome da constante. Já no *TypeScript*, é possível também definir o tipo da constante por meio da notação "`<tipo>`", onde `<tipo>` representa o tipo de dado que a constante armazenará. Essa adição de tipos é uma das vantagens do *TypeScript*, pois traz maior segurança e ajuda a evitar erros de tipo durante o desenvolvimento.

```
1 const <nomeDaConstante>          // em JavaScript
2 const <nomeDaConstante>: <tipo>  // em TypeScript
```

Listagem 2.6: Sintaxe Generalizada de Declaração de Constante em JS e TS.

Outra maneira de declarar uma variável, como se pode observar na Listagem 2.7, é utilizar a palavra-chave `var`. A principal diferença entre o `let` e o `var` é que o `var` tem escopo de função ou escopo global, é içada (*hoisted*) para o topo do escopo e permite reatribuição, enquanto o `let`, usado na Listagem 2.5, tem escopo de bloco, não é içada (*hoisted*) para o topo do escopo e também permite reatribuição, mas apenas dentro do mesmo bloco em que foi declarada (Mozilla Developer Network, 2023c; Mozilla Developer Network, 2023b). A preferência atual é usar `let` em vez de `var` na maioria dos casos, pois o escopo de bloco do `let` oferece maior granularidade de controle e ajuda a evitar problemas de escopo.

```
1 var <nomeDaVariavel>          // Em JavaScript
2 var <nomeDaVariavel>: <tipo>  // Em TypeScript
```

Listagem 2.7: Sintaxe Generalizada de Declaração de Variável Global em JS.

2.4.3 Estruturas de Dados

As variáveis em um algoritmo podem assumir diferentes tipos de dados, dependendo do papel que desempenham. O tipo de dados atribuído a uma variável define o conjunto de valores que ela pode armazenar e as operações que podem ser realizadas sobre ela (Wikipédia, 2022a). Por exemplo, ao calcular a área de uma circunferência, é apropriado utilizar uma variável do tipo numérico para armazenar o raio, permitindo realizar operações matemáticas. Por outro lado, ao armazenar o nome de um usuário, é mais adequado utilizar uma variável do tipo textual para lidar com caracteres e strings. Essa seleção cuidadosa dos tipos de dados garante que as variáveis sejam usadas corretamente e de forma consistente no contexto do algoritmo.

Além disso, nas linguagens de programação, existem os tipos de dados primitivos, que são os tipos mais básicos e fundamentais da linguagem. Esses tipos primitivos servem de base para a definição de outros tipos de dados mais complexos. Em geral, os tipos primitivos incluem inteiros, números de ponto flutuante, caracteres individuais, valores booleanos (verdadeiro/falso) e outros tipos simples. Eles possuem representação direta na memória e são processados de forma eficiente pelos sistemas computacionais. Os tipos primitivos fornecem os blocos de construção essenciais para a criação de estruturas de dados mais complexas e para o desenvolvimento de algoritmos. Portanto, compreender e utilizar corretamente os tipos de dados primitivos é fundamental para a programação eficiente e eficaz (DevMedia, 2006; Wikipédia, 2023).

Os tipos de dados existentes em *JavaScript* também são válidos em *TypeScript*, mas vale lembrar que, diferentemente da linguagem *JavaScript*, *TypeScript* é uma linguagem fortemente tipada e possui alguns tipos adicionais, como o `any` e o `never` (STACK, 2021; TypeScript, 2023; CONTRIBUTORS, 2021). Na Tabela 2.1 é possível relacionar os tipos e seus significados, por exemplo: o tipo `Boolean` representa um valor lógico verdadeiro ou falso; o tipo `Null` representa a ausência intencional de qualquer objeto ou valor; e assim por diante.

Em *JavaScript* e *TypeScript*, há duas formas de indicar a ausência de valor: `null` e `undefined`. Embora sejam semelhantes em alguns aspectos, há diferenças importantes entre eles. O `null` é um valor atribuído explicitamente a uma variável para indicar a ausência intencional de qualquer objeto ou valor. Em outras palavras, quando uma variável é definida como `null`, isso indica que não há valor ou objeto associado a ela. Por outro lado, `undefined` é um valor atribuído automaticamente a uma variável declarada, mas que não recebeu um valor. Em outras palavras, uma variável é definida como `undefined` quando não foi inicializada com um valor válido.

O tipo `any` em *TypeScript* é usado quando não se tem certeza do tipo de uma variável ou quando a variável pode assumir diferentes tipos ao longo do programa. Essencialmente, o tipo `any` desabilita o sistema de tipos do *TypeScript* para uma variável específica, permitindo que ela aceite qualquer valor. Isso pode ser útil em situações em que você está migrando código *JavaScript* existente para o *TypeScript*

Tabela 2.1: Tipos de dados em *JavaScript* (JS) e *TypeScript* (TS)

Tipo	**Descrição**
`Boolean`	Representa um valor lógico verdadeiro ou falso. Pode ser `true` (verdadeiro) ou `false` (falso).
`Null`	Representa a ausência intencional de qualquer objeto ou valor. O `null` é um valor especial que indica a ausência de um valor ou objeto.
`Undefined`	Representa uma variável declarada, mas sem valor atribuído. Quando uma variável é declarada, mas não inicializada, ela terá o valor `undefined`.
`Number`	Representa valores numéricos, como inteiros e números de ponto flutuante.
`BigInt`	Representa valores inteiros maiores do que o limite superior do tipo `number`.
`String`	Representa uma sequência de caracteres.
`Symbol`	Representa um valor único e imutável que pode ser usado como chave de uma propriedade de um objeto.
`Object`	Representa uma coleção de pares chave/valor.

e ainda não tem informações sobre o tipo adequado. No entanto, o uso excessivo do tipo `any` pode anular os benefícios de segurança de tipos que o *TypeScript* oferece.

Por outro lado, em contraste com o tipo `any`, o tipo `never` em *TypeScript* representa o conjunto de valores que nunca ocorrem. Isso significa que uma função que retorna `never` nunca retorna ou sempre lança uma exceção. Esse tipo é útil em cenários em que você quer garantir que uma função nunca conclua com sucesso, como funções que lançam exceções ou em loops infinitos. No *JavaScript*, não há um tipo diretamente equivalente a `never`. No entanto, você pode obter um comportamento semelhante usando exceções ou loops infinitos.

2.5 Números

Os números desempenham um papel fundamental na programação web, pois são essenciais para realizar cálculos, representar quantidades, armazenar informações e controlar o comportamento de aplicativos e sites. Por meio da manipulação de números, é possível realizar operações matemáticas complexas, como cálculos financeiros, estatísticos e de geometria, além de gerenciar datas e horários. Além disso, os números são cruciais para criar interações dinâmicas e responsivas, permitindo a validação de formulários, a geração de números aleatórios, a animação de elementos e muito mais. A habilidade de manipular números com precisão e eficiência é essencial para desenvolver aplicativos web funcionais, confiáveis e capazes de oferecer uma experiência fluida aos usuários. Ambas as linguagens JavaScript (JS) e TypeScript

(TS) oferecem recursos poderosos para manipular e trabalhar com números de forma eficiente.

Números Decimais: JavaScript e *TypeScript* suportam números decimais, representados pelo tipo `number`. Ambas as linguagens permitem realizar operações aritméticas com números decimais, como adição, subtração, multiplicação e divisão. O exemplo na Listagem 2.8 mostra as operações de adição, subtração, multiplicação e divisão entre os números `a` e `b`, respectivamente, nas linhas 3, 4, 5 e 6.

```
1 let a = 5.0;
2 let b = 2.5;
3 let sum = a + b;      // Resultado: 7.5
4 let dif = a - b;      // Resultado: 2.5
5 let prod = a * b;     // Resultado: 12.5
6 let div = a / b;      // Resultado: 2.0
```

Listagem 2.8: Operações com números decimais em *JavaScript* e *TypeScript*

Números Binários: Enquanto *JavaScript* não possui um literal específico para números binários, o *TypeScript* introduziu a sintaxe de literais binários com o prefixo `0b`. Isso permite a representação direta de números binários, como `0b1010` para o número decimal 10, conforme se observa na primeira linha da Listagem 2.9.

Números Octais: Assim como os números binários, o *JavaScript* não possui um literal para números octais. No entanto, o *TypeScript* adicionou suporte a literais octais com o prefixo `0o`. Por exemplo, `0o16` representa o número decimal 14, como mostra a segunda linha da Listagem 2.9.

Números Hexadecimais: Ambas as linguagens *JavaScript* e *TypeScript* suportam números hexadecimais por meio do prefixo `0x`. Na terceira linha da Listagem 2.9, é apresentado um exemplo em que `0xFF` representa o número decimal 255.

```
1 let numBinario = 0b1010;    // Numero decimal 10
2 let numOctal = 0o16;        // Numero decimal 14
3 let numHexadecimal = 0xFF; // Numero decimal 255
```

Listagem 2.9: Números binários em *TypeScript*

Exponenciação/Notação Científica: JavaScript e *TypeScript* suportam a notação científica para representar números muito grandes ou muito pequenos. Por exemplo, `1.23e+6` representa o número 1230000 e `1.23e-6` representa o número 0,00000123, como mostra o exemplo na Listagem 2.10.

```
1 let bigNum = 1.23e+6;    // Numero 1230000
2 let smallNum = 1.23e-6; // Numero 0.00000123
```

Listagem 2.10: Notação científica em *JavaScript* e *TypeScript*

Tabela 2.2: Métodos e Atributos do Objeto Number em JS e TS

Método/Atributo	**Descrição**
toExponential()	String do número em notação exponencial.
toFixed(n)	String do número com n casas decimais.
toPrecision(n)	String do número com a precisão especificada n.
toString(n)	String do número na base n.
valueOf()	Valor numérico primitivo do objeto Number.
MAX_VALUE	O maior número possível da linguagem.
MIN_VALUE	O menor número possível da linguagem.
POSITIVE_INFINITY	Representa o infinito positivo.
NEGATIVE_INFINITY	Representa o infinito negativo.
NaN	Representa um valor que não é um número.

2.5.1 Objeto Number

JavaScript fornece o objeto `Number` embutido, que oferece uma variedade de métodos úteis para trabalhar com números e também é válido em *TypeScript*. Por exemplo, o método `toFixed()` retorna uma representação formatada do número com um número específico de casas decimais. Na Listagem 2.11, demonstra-se o uso do objeto `Number` para formatar o número 42 em uma string com duas casas decimais, resultando em "`42.00`".

```
let num = 42;
let fixedDecimal = num.toFixed(2);
// Numero formatado com 2 casas decimais: "42.00"
```

Listagem 2.11: Utilizando o objeto Number em *JavaScript* e *TypeScript*

O uso do objeto `Number` permite realizar operações mais avançadas, como conversões entre bases numéricas e manipulação precisa de valores numéricos. Ele oferece uma série de métodos e propriedades, como `toString()`, que retorna a string do número em uma base numérica especificada como argumento, entre outros, que podem ser observados na Tabela 2.2, que apresenta alguns dos principais métodos e atributos que podem ser utilizados com o objeto `Number`. Esses recursos fornecem funcionalidades adicionais para a manipulação e formatação de números em *JavaScript* e *TypeScript*, permitindo realizar operações mais avançadas e precisas (Mozilla Developer Network, 2023a).

2.5.2 NaN (Not a Number)

Em *JavaScript* e *TypeScript*, o valor especial `NaN` (*Not a Number*) é usado para representar um resultado numérico inválido ou indefinido. Ele é retornado quando ocorre uma operação matemática que não pode produzir um resultado numérico

válido. O `NaN` é um valor do tipo número, mas não é igual a nenhum número real ou infinito. Ele é considerado um valor falso quando usado em um contexto booleano. Por exemplo, a expressão `NaN === NaN` retorna `false`, o que indica que `NaN` não é igual a si mesmo. Alguns exemplos de casos em que o `NaN` pode ser gerado são:

- Divisão por zero: Quando um número é dividido por zero, o resultado é indefinido e é representado como `NaN`.
- Operações matemáticas inválidas: Se uma operação matemática sem sentido for executada, como tentar calcular a raiz quadrada de um número negativo, o resultado será `NaN`.
- Conversões numéricas inválidas: Ao tentar converter uma string que não representa um número válido em *JavaScript* ou *TypeScript*, o resultado será `NaN`.

O `NaN` tem algumas propriedades peculiares quando usado em cálculos, pois o `NaN` é contagioso. Qualquer operação aritmética com `NaN` resultará em `NaN`, como por exemplo: `NaN + 5` ou `NaN * 10`. Por isso, ao lidar com valores numéricos em *JavaScript* e *TypeScript*, é importante verificar se um valor é `NaN` antes de usá-lo. Pode-se usar a função `isNaN()` para verificar se um valor é ou não `NaN`. Essa função retorna `true` se o valor fornecido for `NaN` e `false` caso contrário.

Embora o comportamento do `NaN` seja semelhante em *JavaScript* e *TypeScript*, o *TypeScript* oferece suporte adicional aos tipos de dados estáticos, o que pode ajudar a evitar erros relacionados ao uso de `NaN` em operações matemáticas. Com o *TypeScript*, é possível definir tipos específicos para variáveis e parâmetros de função, o que pode ajudar a identificar erros de tipo em tempo de compilação e evitar situações em que `NaN` é usado incorretamente.

2.6 Strings

As strings desempenham um papel crucial na programação web, sendo fundamentais para manipular e exibir texto e outros conteúdos em aplicativos e sites. Elas permitem a representação e o armazenamento de informações, como nomes, mensagens, endereços e muitos outros dados textuais. Além disso, a manipulação de strings é essencial para realizar operações como concatenação, formatação, busca, substituição e extração de partes específicas do texto. Isso permite criar recursos como formulários interativos, validação de entrada de dados, filtragem e ordenação de listas, geração dinâmica de conteúdo e muito mais. A habilidade de trabalhar com strings de forma eficiente e precisa é essencial para o desenvolvimento de aplicativos web robustos, funcionais e com uma interface de usuário envolvente e amigável. Ambas as linguagens *JavaScript* e *TypeScript* oferecem recursos poderosos para trabalhar com strings.

Declaração de Strings: Em *JavaScript* e *TypeScript*, as strings podem ser declaradas com aspas simples (‘ ’) ou aspas duplas (“ ”). Ambas as formas são

igualmente válidas e produzem a mesma saída. Pode-se optar por usar aspas simples ou duplas, com base em preferências pessoais ou em convenções de estilo de código adotadas em um projeto específico. A Listagem 2.12 apresenta um exemplo onde se declara uma variável `mensagem` com o valor 'Ola, mundo!', utilizando-se aspas simples, e outra variável `nome` com o valor "Maria", declarada com aspas duplas.

```
let mensagem = 'Ola, mundo!';
let nome = "Maria";
```

Listagem 2.12: Declaração de String em JS e TS.

Concatenação de Strings: A concatenação de strings é uma operação comum ao lidar com texto. Em *JavaScript* e *TypeScript*, pode-se concatenar strings utilizando o operador de adição (+) ou o método `concat()`. Para exemplificar, na Listagem 2.13, declara-se uma variável `saudacao` com o valor "Ola, " e outras variáveis `nome1` e `nome2` com os valores "Maria" e "Jose", respectivamente. Em seguida, utiliza-se o operador de adição (+) para concatenar as strings `saudacao` e `nome1`, armazenando o resultado na variável `mensagem`, que resultou em "Ola, Maria". Para realizar outra concatenação, foi utilizado o método `concat()` entre as strings `saudacao` e `nome2`, resultando em "Olá, Jose".

```
let saudacao = "Ola, ";
let nome1 = "Maria";
let nome2 = "Jose";
let mensagem1 = saudacao + nome1;       // "Ola, Maria"
let mensagem2 = saudacao.concat(nome2); // "Ola, Jose"
```

Listagem 2.13: Concatenção de String em JS e TS.

Acessando Caracteres: Caracteres individuais de uma string podem ser acessados por meio da notação de colchetes ([]). Os índices começam em zero, ou seja, o primeiro caractere tem o índice 0. No exemplo da Listagem 2.14, a variável `nome` é declarada com o valor "Alice". Em seguida, foi utilizada a notação de colchetes com o índice zero para acessar o primeiro caractere da string, que é "A", e ainda, foi utilizado o atributo `length` da string para pegar seu tamanho e, posteriormente, subtraído em uma unidade para indicar o índice do último caractere, que é "e".

```
let nome = "Alice";
let primeiroCaractere = nome[0];              // "A"
let ultimoCaractere = nome[nome.length - 1]; // "e"
```

Listagem 2.14: Acessando caracteres de strings em JS e TS.

Métodos de Manipulação de Strings: *JavaScript* e *TypeScript* fornecem uma variedade de métodos embutidos para manipulação de strings. A Listagem 2.15

ilustra o uso do método `toUpperCase` para converter todos os caracteres da string "`julia`" para maiúsculas. Alguns dos métodos de manipulação de string em JS e TS mais comuns incluem:

- `length`: Retorna o comprimento da string.
- `toUpperCase()`: Converte a string para letras maiúsculas.
- `toLowerCase()`: Converte a string para letras minúsculas.
- `substring(startIndex, endIndex)`: Retorna uma parte da string, começando do índice `startIndex` até o índice `endIndex` (não incluído).
- `split(separator)`: Divide a string em um array de substrings com base no separador especificado.
- `trim()`: Remove espaços em branco do início e do final da string.

```
let nome = "julia";
let nomeMaiusculo = nome.toUpperCase() // "JULIA"
```

Listagem 2.15: Acessando caracteres de strings em JS e TS.

2.7 Operações Básicas

As operações básicas são realizadas pelos operadores nativos da linguagem, amplamente utilizados; são como comandos atômicos. Em *JavaScript* (JS) e *TypeScript* (TS), dentre essas operações estão: operadores aritméticos, operadores de atribuição e operadores de comparação.

2.7.1 Atribuição

Atribuição é uma operação que armazena um valor em uma variável. O operador utilizado para essa tarefa em JS e TS é o igual (=). Como mostra a Listagem 2.16, há a declaração de uma variável numérica e em seguida é-lhe atribuído o valor 5, na primeira e segunda linhas, respectivamente. Também é possível fazer o mesmo processo de declaração e atribuição de uma forma mais sucinta, como é expresso na terceira linha.

```
let minhaVariavel;
minhaVariavel = 5;
let minhaOutraVariavel = 5;
```

Listagem 2.16: Exemplo de Atribuição em JS e TS.

A *inicialização* é a primeira operação de atribuição que uma variável recebe e é fundamental para garantir que ela não fique com um valor vazio ou indefinido. No *TypeScript*, ao inicializar uma variável com um valor, a linguagem realiza a *inferência de tipo*, determinando automaticamente o tipo da variável com base nessa atribuição. Se uma variável for declarada e receber um valor na mesma linha, o compilador é capaz de inferir seu tipo, dispensando a necessidade de declarar explicitamente o tipo da variável. No entanto, se uma variável for declarada sem especificar seu tipo e não receber um valor na mesma linha, seu tipo será assumido como `any` no código *TypeScript*.

2.7.2 Comentários

Os comentários, no contexto da programação, são trechos do código que não interferem no programa; são apenas anotações no decorrer do código. Para iniciar um comentário, tanto em JS como em TS, basta inserir duas barras (//) no código e todo o texto que estiver em sequência na mesma linha será considerado como um comentário. Caso seja necessário escrever um comentário que ocupe mais de uma linha, pode-se iniciá-lo com uma barra e um asterisco (/*) e finalizá-lo com um asterisco e uma barra (*/). Como mostra o exemplo na Listagem 2.17, na primeira linha de código há um comando de declaração e inicialização de uma variável, e após esse comando, na mesma linha, um comentário "Comentario em uma linha", e após, nas linhas seguintes, um outro comentário "Este comentario pode tomar varias linhas".

```
1 let minhaVariavel = 2; // Comentario em uma linha
2 /* Este comentario
3    pode tomar
4    varias linhas */
```

Listagem 2.17: Exemplo de Comentário em JS e TS.

A importância dos comentários reside no fato de que eles facilitam a compreensão do código para os desenvolvedores, tanto para o próprio autor quanto para outros que possam trabalhar ou dar manutenção no código no futuro. Comentários bem escritos ajudam a descrever a lógica por trás de uma seção de código, a fornecer contexto importante e a documentar decisões de design. Os comentários também são úteis para adicionar lembretes, fazer anotações ou desativar temporariamente partes do código durante o desenvolvimento. Além disso, eles podem ajudar na depuração, permitindo que os desenvolvedores expliquem o propósito de certas linhas de código ou identifiquem possíveis problemas. Ao utilizar comentários, é importante seguir algumas boas práticas, que incluem: manter os comentários atualizados à medida que o código evolui, escrever comentários claros e concisos, evitar comentários óbvios ou redundantes e evitar comentários em excesso que possam poluir o código.

2.7.3 Operações Aritméticas

Algumas operações aritméticas básicas e recorrentemente utilizadas na programação são: adição, subtração, multiplicação, divisão e módulo (resto da divisão). Essas operações são realizadas em *JavaScript* e *TypeScript* de maneira semelhante à matemática convencional, e seus resultados podem ser armazenados em variáveis para uso posterior. É importante mencionar que, assim como na matemática, os parênteses podem ser utilizados para definir a ordem de prioridade das operações.

A Listagem 2.18 mostra alguns exemplos de operações aritméticas básicas, dentre elas: na primeira linha, a adição dos valores 2 e 5, em que o resultado é atribuído à variável `x`, que terá o valor 7; na segunda linha, a subtração do valor 5 do valor 15, em que resultado é atribuído à variável `y`, que terá o valor 10; na terceira linha, a multiplicação dos valores 3 e 5, em que o resultado é atribuído à variável `z`, que terá o valor 15; na quarta linha, a divisão de 15 por 5, em que o resultado é atribuído à variável `w`, que terá o valor 3; e na quinta linha, a operação de resto da divisão, dividindo 7 por 5, em que o resultado é atribuído à variável `k`, que terá o valor 2.

```
let x = 2 + 5; // x vale 7
let y = 15 - 5; // y vale 10
let z = 3 * 5; // z vale 15
let w = 15 / 5; // w vale 3
let k = 7 % 5; // k vale 2
```

Listagem 2.18: Exemplos de Operações Aritméticas em *JavaScript* e *TypeScript*.

Essas operações aritméticas são úteis em uma variedade de cenários de programação, desde cálculos simples até operações mais complexas envolvendo variáveis e expressões. É importante ter em mente a ordem de precedência dos operadores aritméticos, bem como o uso adequado de parênteses quando necessário, para obter os resultados desejados.

2.7.4 Objeto Math

O objeto `Math` em *JavaScript* e *TypeScript* fornece um conjunto de propriedades e métodos para realizar operações matemáticas avançadas. Essas operações abrangem desde funções trigonométricas e exponenciais até operações de arredondamento e geração de números aleatórios. Nessa seção, são apresentados alguns dos principais métodos e propriedades disponíveis no objeto `Math`.

Métodos Trigonométricos: O objeto `Math` oferece métodos para calcular funções trigonométricas como seno, cosseno e tangente. Na Listagem 2.19, usa-se o método `Math.sin()` para calcular o seno de um ângulo. No exemplo, o ângulo é definido como $\frac{\pi}{4}$, que é equivalente a 45 graus em radianos. O resultado é armazenado na variável `sineValue` e, em seguida, é impresso no console. Além do

`Math.sin()`, o objeto `Math` também oferece os métodos `Math.cos()` e `Math.tan()` para calcular o cosseno e a tangente de um ângulo, respectivamente.

```
// Angulo de 45 graus em radianos
const angle = Math.PI / 4;
// Calcula o seno do angulo
const sineValue = Math.sin(angle);
```

Listagem 2.19: Exemplo de uso do método `Math.sin()`.

Métodos Exponenciais: O objeto `Math` possui métodos para cálculos exponenciais, como elevar um número a uma potência e calcular o logaritmo. Na Listagem 2.20, o método `Math.pow()` é utilizado para calcular a potência de um número. No exemplo, eleva-se o número 2 à potência de 3, o que resulta em 8. O resultado é armazenado na variável `result` e é impresso no console. Além do `Math.pow()`, o objeto `Math` também possui o método `Math.sqrt()` para calcular a raiz quadrada de um número e o método `Math.log()` para calcular o logaritmo natural.

```
const base = 2;
const exponent = 3;
// Calcula 2 elevado a 3, que resulta em 8
const result = Math.pow(base, exponent);
```

Listagem 2.20: Exemplo de uso do método `Math.pow()`.

Métodos de Arredondamento: O objeto `Math` fornece métodos para arredondar números para cima, para baixo e para o valor inteiro mais próximo. Na Listagem 2.21, utilizam-se os métodos `Math.ceil()`, `Math.floor()` e `Math.round()` para arredondar um número. No exemplo, o número inicial é 4.8. O método `Math.ceil()` é usado para arredondar para cima, `Math.floor()` para arredondar para baixo e `Math.round()` para arredondar para o valor inteiro mais próximo. Os resultados são armazenados em variáveis diferentes e são impressos no console.

```
const number = 4.8;
// Arredonda para cima: 5
const roundedUp = Math.ceil(number);
// Arredonda para baixo: 4
const roundedDown = Math.floor(number);
// Arredonda para o valor inteiro mais proximo: 5
const rounded = Math.round(number);
```

Listagem 2.21: Exemplo de uso dos métodos de arredondamento em `Math`.

Métodos de Números Aleatórios: O objeto `Math` também possui um método para gerar números aleatórios. Na Listagem 2.22, foi utilizado o método

`Math.random()` para gerar um número aleatório entre 0 e 1. O resultado é armazenado na variável `random` e é impresso no console.

```
// Gera um numero aleatorio entre 0 e 1
const random = Math.random();
```

Listagem 2.22: Exemplo de uso do método `Math.random()`.

2.7.5 Operações Relacionais

As operações relacionais são utilizadas para comparar valores. Essas operações são comumente usadas em estruturas de decisão, como os comandos `if` e `switch` (que serão apresentados posteriormente), para verificar se uma determinada condição é verdadeira ou falsa. Existem seis operações relacionais em *JavaScript* e *TypeScript*: igual (==), diferente (`!=`), maior que (`>`), maior ou igual (`>=`), menor que (`<`) e menor ou igual (`<=`). Quando se utiliza uma dessas operações para comparar dois valores, o resultado será um valor booleano (verdadeiro/*true* ou falso/*false*).

Na Listagem 2.23 há alguns exemplos de uso das operações relacionais entre as variáveis `a` e `b`, dentre elas: na terceira linha, a operação de igualdade (==), que resultará em *false*, que é atribuído à variável `x`; na quarta linha, a operação de diferença, o resultado será atribuído à variável `y`, que terá o valor booleano *true*; na quinta linha, a operação de maior que, que resultará em *true* e será atribuído à variável `z`; na sexta linha, a operação de maior ou igual, que resultará em *true* e será atribuído à variável `w`; na sétima linha, a operação de menor que, que resultará em *false* e será atribuído à variável `k`; e por fim, na oitava linha, a operação de menor ou igual, atribuída à variável `m`, que valerá *false*.

```
const a = 5;
const b = 4;
const x = a == b; // x = false
const y = a != b; // y = true
const z = a > b;  // z = true
const w = a >= b; // w = true
const k = a < b;  // k = false
const m = a <= b; // m = false
```

Listagem 2.23: Exemplo de Operações Relacionais em *JavaScript* e *TypeScript*.

Além desses operadores, também existem os operadores estritamente igual, os quais são definidos sintaticamente como três iguais juntos (`===`), e o estritamente diferente, com sintaxe de um ponto de exclamação seguido de dois sinais de igual (`!==`). Esses operadores diferem dos operadores de igualdade (`==`) e diferença (`!=`) por considerarem também os tipos dos valores sendo comparados. Eles exigem que os valores sejam iguais em valor e tipo para serem considerados verdadeiros. Esses

operadores são úteis quando se deseja garantir que os valores comparados sejam idênticos quanto a valor e tipo.

Recomenda-se utilizar os operadores estritamente igual e estritamente diferente sempre que possível, pois oferecem uma comparação mais precisa e ajudam a evitar resultados inesperados decorrentes da coerção de tipo. Além disso, é importante notar que, ao realizar comparações em *JavaScript* e *TypeScript*, é necessário considerar as regras de coerção de tipo (*type coercion*) que podem ocorrer durante as operações. Por exemplo, ao comparar um número com uma string, o *JavaScript* tentará converter a string em um número antes de realizar a comparação.

2.7.6 Entrada e Saída

Os comandos básicos de entrada e saída são utilizados para interagir com o usuário ou com dispositivos externos, permitindo que o programa receba informações e exiba resultados. Em *JavaScript* e *TypeScript* os comandos de entrada e saída são os mesmos. Para ler os dados do usuário, utiliza-se o comando `prompt`, que exibe uma janela de diálogo na qual o usuário pode inserir um valor. Já para exibir informações ao usuário, utiliza-se o comando `console.log`, que imprime uma mensagem no console do navegador ou no console do *Node.js*. A Listagem 2.24 ilustra a utilização dos comandos, em que na segunda linha é solicitado o nome do usuário com a função `prompt` e depois é impresso no console o nome digitado com o comando `console.log`.

```
1 // solicita uma entrada do usuario
2 let nome = prompt("Digite seu nome:");
3 // imprime no console a entrada do usuario
4 console.log(nome);
```

Listagem 2.24: Comando `prompt` e `console.log` em JS e TS.

Além do comando `console.log` que imprime mensagens no console, existe também o comando `alert`, que exibe uma janela de alerta no navegador com uma mensagem para o usuário, como mostrado na Figura 2.1. O `alert` é um comando de saída muito utilizado para exibir informações importantes ou notificações ao usuário. Esse comando é particularmente útil em contextos em que é necessário chamar a atenção do usuário para alguma informação importante, como validações de formulários, confirmações de ações ou exibição de erros. No entanto, é importante usar o `alert` com moderação, pois janelas de alerta excessivas podem ser intrusivas e perturbar a experiência do usuário.

Em *JavaScript* e *TypeScript*, o uso do `alert` é semelhante ao de `console.log`. A Listagem 2.25, baseado na Listagem 2.24, ilustra o uso do `alert` para exibir o nome digitado pelo usuário em uma janela de alerta. Nesse exemplo, após solicitar o nome do usuário com o comando `prompt`, o comando `alert` é utilizado para exibir o nome em uma janela de alerta. Assim, o usuário verá o valor digitado no alerta do navegador.

```
// solicita uma entrada do usuario
let nome = prompt("Digite seu nome:");
// exibe o nome em uma janela de alerta
alert(nome);
```

Listagem 2.25: Exemplo de Utilização do Comando Alert em JS e TS.

2.8 Condicionais

As condicionais são estruturas de controle que permitem ao programa executar diferentes blocos de código com base em condições específicas. Elas são amplamente utilizadas em *JavaScript* e *TypeScript* para controlar o fluxo do programa com base em valores e expressões. As condicionais mais comuns nessas linguagens são `if/else`, mas também existe a estrutura `switch/case`, que permite controlar o fluxo do código com base em diferentes valores de uma variável.

2.8.1 `if/else`

O comando `if` é usado para testar uma condição e executar um bloco de código se ela for verdadeira. A Listagem 2.26 exemplifica a utilização da estrutura, onde o programa verifica se a variável `idade` é maior ou igual a 18. Se essa condição for verdadeira, o bloco de código dentro do `if` é executado e exibe a mensagem "Voce eh maior de idade" no console.

```
if (idade >= 18) {
    console.log("Voce eh maior de idade");
}
```

Listagem 2.26: Exemplo de condicional `if` em JS e TS.

O comando `else` é usado em conjunto com o `if` e permite executar um bloco de código caso a condição no `if` seja falsa. Como é possível observar na Listagem 2.27, se a condição no `if` for falsa, o bloco de código dentro do `else` é executado e exibe a mensagem "Voce eh menor de idade" no console.

```
if (idade >= 18) {
    console.log("Voce eh maior de idade");
} else {
    console.log("Voce eh menor de idade");
}
```

Listagem 2.27: Exemplo de condicional `if/else` em JS e TS.

A estrutura `else if` é usada para testar condições adicionais, caso a condição no `if` seja falsa. Na Listagem 2.28, se a condição no `if` for falsa, o programa testa a condição no `else if` e, se essa condição for verdadeira, o bloco de código correspondente é executado e exibe a mensagem "Voce tem idade suficiente para dirigir", caso contrário, o bloco de código no `else` é executado e exibe a mensagem "Voce eh menor de idade".

```
if (idade >= 18) {
    console.log("Voce eh maior de idade.");
} else if (idade >= 16) {
    console.log("Voce tem idade suficiente para dirigir.");
} else {
    console.log("Voce eh menor de idade.");
}
```

Listagem 2.28: Exemplo de condicional utlizando `if/else` e `else if` em JS e TS.

2.8.2 `switch/case`

A estrutura `switch/case` é útil quando há várias condições a serem testadas e se quer evitar usar várias estruturas `if/else`. O comando `switch` inicia a estrutura, seguido pela variável que será testada nos diferentes casos. Como se pode observar na Listagem 2.29, o programa verifica a variável `diaDaSemana` e executa o bloco de código correspondente a cada caso. Se o valor for "Sabado" ou "Domingo", a mensagem "Hoje eh fim de semana" será exibida no console. Caso contrário, o bloco de código no caso `default` é executado e exibe a mensagem "Hoje eh dia de trabalho".

```
switch (diaDaSemana) {
    case "Sabado":
    case "Domingo":
        console.log("Hoje eh fim de semana");
        break;
    default:
        console.log("Hoje eh dia de trabalho");
        break;
}
```

Listagem 2.29: Exemplo de condicional `switch/case` em JS e TS.

O comando `break` é uma instrução utilizada em JavaScript e TypeScript para interromper a execução de um laço de repetição ou de um bloco de código dentro de uma estrutura condicional. Quando o comando `break` é executado, o controle do programa é transferido imediatamente para fora do laço ou bloco de código, ignorando as instruções restantes. É importante notar que cada bloco de código

dentro de `case` deve terminar com o comando `break`. Isso impede que o programa execute os casos seguintes, mesmo que a condição seja verdadeira.

2.9 Laços de Repetição

Os laços de repetição são estruturas que permitem que o programa execute um bloco de código várias vezes, com base em uma condição específica. Eles são amplamente utilizados em *JavaScript* e *TypeScript* para automatizar tarefas repetitivas e percorrer listas de elementos. Existem dois tipos principais de laços de repetição nessas linguagens: `for` e `while`. Cada um tem sua utilidade, dependendo da situação.

2.9.1 For

O laço `for` é frequentemente usado quando se conhece o número exato de repetições desejadas do bloco de código. Ele possui três partes: a inicialização, a condição e o incremento. A inicialização é a etapa em que a variável do laço é definida e inicializada. A condição é a expressão que deve ser verdadeira para que a execução do bloco de código continue. O incremento define como a variável do laço é atualizada a cada iteração.

```
1 for (let i = 0; i < 10; i++) {
2     console.log(i);
3 }
```

Listagem 2.30: Exemplo do laço de repetição `for` em JS e TS.

Na Listagem 2.30, o programa executa o bloco de código dentro do laço de repetição dez vezes. Ele exibe os números de 0 a 9 no console, pois o que acontece é o seguinte: na primeira repetição, o valor de `i` assume o valor da sua inicialização "`let i = 0`", e executa o bloco de código dentro do loop; após finalizar a execução do bloco, o incremento "`i++`" é realizado, e `i` recebe seu valor anterior somado de um; depois do incremente a verificação da condição "`i < 10`" é realizada e, se for verdadeira, o bloco de código do `for` é repetido, caso não, o *loop* termina. Esse processo de incremento e verificação de condição é realizado em cada iteração.

2.9.2 While

O laço `while` é comumente usado quando não se sabe antecipadamente quantas vezes é necessário repetir o bloco de código. Ele repete o bloco de código enquanto a condição especificada for verdadeira. Como mostra a Listagem 2.31, o programa executa o bloco de código no laço de repetição enquanto a variável `i` for menor que 10. Ele exibe os números de 0 a 9 no console, incrementando a variável `i` a cada iteração.

```
let i = 0;
while (i < 10) {
    console.log(i);
    i++;
}
```

Listagem 2.31: Exemplo do laço de repetição `while` em JS e TS.

É importante garantir que a condição do laço de repetição eventualmente se torne falsa, caso contrário, o laço será executado infinitamente, causando um loop infinito no programa. Portanto, é necessário cuidado ao definir a condição para evitar esse problema.

2.10 Funções

Funções são blocos de código que podem ser definidos em um programa para realizar uma tarefa específica. Em *TypeScript*, as funções podem ser definidas e usadas de maneira semelhante ao *JavaScript*, mas com a adição de tipos de dados estáticos. Isso torna as funções em *TypeScript* mais robustas em termos de verificação de tipos e fornecem uma maneira mais clara de entender os tipos de entrada e saída esperados.

Declaração e Execução de Funções: Para se declarar uma função usa-se a palavra-chave `function`, seguida do nome da função, uma lista de parâmetros entre parênteses e o corpo da função dentre chaves, como mostra a Listagem 2.32. No caso do *TypeScript* pode-se definir ainda o tipo de dado do retorno da função, como elucida o exemplo da Listagem 2.33. Já para executar uma função declarada, basta chamá-la pelo seu nome e, caso necessário, passar os devidos parâmetros, como mostra a quarta linha de ambos os exemplos em JS e TS.

```
function saudacao() {
  console.log("Ola, mundo!");
}
saudacao();  // Log: "Ola, mundo!"
```

Listagem 2.32: Exemplo de Declaração de Função em JS.

```
function saudacao(): void {
  console.log("Ola, mundo!");
}
saudacao();  // Log: "Ola, mundo!"
```

Listagem 2.33: Exemplo de Declaração de Função em TS.

Parâmetros da Função: Ao declarar uma função, pode-se especificar seus parâmetros. Na linguagem *TypeScript* é possível ainda especificar os tipos dos parâmetros, ajudando a melhorar a verificação de tipos. Nos exemplos da Listagem 2.34 e Listagem 2.35, pode-se observar a especificação de duas variáveis parâmetro `a` e `b`, que pelo contexto, espera-se que sejam do tipo numérico. Ou seja, ao chamar a função `somar`, deve-se passar dois valores numéricos para que a função use esses valores para executar.

```
1 function somar(a, b): void {
2   console.log(a + b);
3 }
```

Listagem 2.34: Exemplo de Declaração de Função com Parâmetros em JS.

```
1 function somar(a: number, b: number): void {
2   console.log(a + b);
3 }
```

Listagem 2.35: Exemplo de Declaração de Função com Parâmetros em TS.

O exemplo da Listagem 2.36 mostra alguns casos de execução das funções declaradas na Listagem 2.34 e Listagem 2.35 em JS e TS. Pode-se observar a execução correta das somas de 3 mais 5, que resultam em 8, e da adição de -10 a 10, que resulta em 0. Contudo, as diferenças na utilização das linguagens *JavaScript* e *TypeScript* começam a se tornar mais evidentes a partir deste ponto, pois, caso sejam passados parâmetros de tipos não numéricos para a função `somar`, o verificador de tipos do *TypeScript* alertará sobre a inconsistência no código, o que não ocorre em *JavaScript*.

```
1 somar(3, 5);        // Resultado em JS e TS: 8
2 somar(10, -10);     // Resultado em JS e TS: 0
3 somar("JS", "TS"); // Resultado em JS: "JSTS"
4 /* Resultado em TS:  Error: Argument of type 'string'
5 is not assignable to parameter of type 'number'. */
```

Listagem 2.36: Exemplos de Execução de Função em JS e TS.

Parâmetros Opcionais: Em *TypeScript* é possível também declarar uma função com parâmetro opcional, o que significa que ele pode ser passado ou não para a função. Para isso, usa-se o operador de interrogação (`?`) após o nome do parâmetro, como evidencia o exemplo na Listagem 2.37. Na nona linha são passados os três parâmetros da função e a soma de 3 mais 1 mais 4 ocorre e é mostrado o resultado 8 no console. Já na décima linha, somente dois parâmetros, 5 e 2, são passados para a função `somar`, e a função executa sem erros mostrando o valor 7 no console, pois o terceiro parâmetro é opcional.

```
function somar(a:number, b:number, c?:number):void{
    if (c){
        console.log(a+b+c);
    }
    else{
        console.log(a+b);
    }
}
somar(3, 1, 4); // Log: 8
somar(5, 2);    // Log: 7
```

Listagem 2.37: Exemplos de Parâmetros Opcionais em TS.

Valores Padrão de Parâmetros: Ainda na declaração das funções, é possível definir valores padrões para os parâmetros das funções nas linguagens JS e TS. Os parâmetros assumem os valores padrão caso, na chamada de execução da função, eles não sejam especificados. Isso viabiliza uma maneira de tornar os parâmetros opcionais. A Listagem 2.38 ilustra essa ideia, na quarta linha, em que os dois parâmetros são informados, 1 e 2, resultando em 3. Diferentemente, na quinta linha, somente um dos parâmetros é informado e a função executa normalmente, atribuindo o valor zero (valor padrão) ao segundo parâmetro. Já na sexta linha, nenhum parâmetro é informado e todos os parâmetros, por padrão, recebem o valor zero, fazendo a soma resultar em zero.

```
function somar(a = 0, b = 0){
    console.log(a+b+c);
}
somar(1, 2);    // Log: 3
somar(1);       // Log: 1
somar();        // Log: 0
```

Listagem 2.38: Exemplos de Valores Padrão de Parâmetros em JS.

Retornos da Função: O retorno de uma função é o valor que a finalização de sua execução assume. Em *TypeScript* é possível especificar o tipo de retorno de uma função usando a notação de dois pontos (:) seguida do tipo de dado desejado. Se a função não retornar nenhum valor, pode-se usar o tipo void, como mostrado nos exemplos anteriores. Tomando como base o exemplo de Listagem 2.38, ao rodar a função `somar` com diferentes parâmetros, o resultado da soma é exibido na tela. Contudo, a função não retorna nada, como se evidencia no exemplo de Listagem 2.39. Nota-se, então, que a função `somar` retornou um resultado, mas esse valor só existe durante a execução da função e não pode ser recuperado externamente a ela. O retorno desempenha esse papel por meio do comando `return`, como mostra o exemplo da Listagem 2.40.

```
const a = somar(3, 5); // Log: 8
console.log(a);        // Log: undefined
```

Listagem 2.39: Evidenciando Retorno da Função.

```
function somar(a = 0, b = 0){
    return a+b;
}
somar(3, 5); // nada aparece na tela
const x = somar(3, 5);
console.log(x); // Log: 8
```

Listagem 2.40: Exemplo de Retorno de Função.

2.10.1 Arrow Function

As *arrow functions*, ou funções de seta, têm uma sintaxe mais curta e concisa para criar funções em JavaScript e TypeScript. Em vez de escrever a palavra 'function' e o bloco de código da função, pode-se simplesmente usar a seta `(=>)` após os parâmetros da função para definir o bloco de código. Essa sintaxe simplificada traz benefícios em termos de legibilidade e de escrita de código mais concisa. Como apresentado na Listagem 2.41, há uma função de soma declarada como uma *arrow function*. A função recebe dois parâmetros opcionais, 'a' e 'b', e exibe a soma dos dois valores no console. O uso da *arrow function* torna a declaração da função mais compacta e clara, o que permite que o código seja lido e compreendido de forma mais eficiente.

```
// Arrow function
const soma = (a = 0, b = 0) => {
    console.log(a + b);
}
soma(1, 2); // Log: 3
```

Listagem 2.41: Exemplo de Arrow Function.

Declarar arrow functions como `const` é uma prática recomendada. Ao atribuir uma arrow function a uma constante, garante-se que a referência à função não será reatribuída a outro valor. Isso ajuda a evitar erros acidentais, pois impede que a função seja substituída inadvertidamente por outro valor. Além disso, utilizar `const` em funções arrow ajuda a transmitir a intenção de que a função é imutável e não deve ser modificada posteriormente.

2.11 Tratamento de Exceções

O tratamento de exceções desempenha um papel crucial na robustez e na confiabilidade do software. Ele evita que erros não tratados interrompam abruptamente a execução do programa, garantindo que o código seja capaz de lidar com falhas e tomar ações apropriadas. Além disso, ao capturar e tratar exceções, é possível fornecer mensagens de erro significativas e informativas, facilitando a depuração e a identificação de problemas. O tratamento de exceções pode ser realizado com os blocos `try`, `catch` e `continue` em linguagens de programação como *JavaScript* e *TypeScript*. Essa abordagem permite capturar exceções, tratar erros de forma controlada e continuar a execução do programa mesmo após um erro.

O bloco `try` é usado para envolver o código que pode gerar exceções. Se uma exceção ocorrer dentro desse bloco, o fluxo de execução é imediatamente desviado para o bloco `catch`. Isso permite que você capture a exceção e forneça um tratamento adequado. O bloco `catch` contém o código que será executado quando uma exceção é lançada no bloco `try`. Nele pode-se escrever lógica personalizada para lidar com o erro. É possível fornecer um identificador de exceção após a palavra-chave `catch`, como `catch(erro)`, para acessar informações adicionais sobre a exceção, como a mensagem de erro ou detalhes específicos. A instrução `continue` é usada para pular para a próxima iteração de um loop, ignorando temporariamente o erro. Ela é útil quando se deseja continuar a execução do programa mesmo após um erro, sem interromper completamente o fluxo.

Na Listagem 2.42, tem-se um array de números e um valor de destino que se quer encontrar. No laço de repetição `for`, utiliza-se o bloco `try` para processar cada número do array. Se o valor de destino for encontrado, uma mensagem é exibida e usa-se `continue` para pular para a próxima iteração. Caso ocorra algum erro durante o processamento de um número, o bloco `catch` é acionado e exibe uma mensagem de erro.

```
1  const numbers = [1, 2, 3, 4, 5];
2  const targetValue = 7;
3
4  for (let i = 0; i < numbers.length; i++) {
5    try {
6      if (numbers[i] === targetValue) {
7        console.log("Valor encontrado!");
8        continue;
9      }
10     console.log("Processando numero: " + numbers[i]);
11   } catch (erro) {
12     console.log("Ocorreu um erro: " + erro.message);
13   }
14 }
```

Listagem 2.42: Exemplo de uso de `try`, `catch` e `continue` em JS e TS

Já é sabido que o *TypeScript* possui uma funcionalidade adicional em relação ao *JavaScript*: a verificação de tipos estáticos. Isso significa que, ao utilizar o `try`, `catch` e `continue` no *TypeScript*, pode-se fornecer tipos específicos para as exceções capturadas e também para as variáveis usadas dentro dos blocos `try` e `catch`, garantindo uma maior segurança e clareza no código.

2.12 *Promises*

As *Promises* são um recurso poderoso para lidar com operações assíncronas em *JavaScript* e *TypeScript*. Elas permitem executar tarefas assíncronas e lidar com seus resultados ou erros de forma mais estruturada. As *Promises* foram introduzidas na especificação ES6 (ECMAScript 2015) para tratar assincronicidade de forma mais eficiente em *JavaScript*. Uma *Promise* representa um valor que pode estar disponível imediatamente, no futuro ou nunca. Ela pode estar em um dos três estados: *pending* (pendente), *fulfilled* (cumprida) ou *rejected* (rejeitada).

Na Listagem 2.43, uma nova *Promise* é criada com uma função que recebe dois parâmetros: `resolve` e `reject`. Nessa função, é possível realizar operações assíncronas, como chamadas de API ou acesso a um banco de dados. Ao concluir a operação com sucesso, chama-se `resolve` com o valor resultante. Se ocorrer um erro, chama-se `reject` com o erro correspondente.

```
const promise = new Promise((resolve, reject) => {
    // Logica assincrona aqui
    // Se a operacao for bem-sucedida, chame resolve(valor)
    // Se a operacao falhar, chame reject(erro)
});
```

Listagem 2.43: Criação de uma Promise em JavaScript

O *TypeScript*, sendo um superconjunto do *JavaScript*, oferece suporte completo às *Promises*. Além das funcionalidades disponíveis em *JavaScript*, o *TypeScript* permite especificar tipos para os valores resolvidos e rejeitados. Ao utilizar *Promises* em *TypeScript*, pode-se definir o tipo dos valores de retorno, o que proporciona maior segurança e clareza no código. Na Listagem 2.44, a função `fetchData` retorna uma *Promise* que deve resolver um valor do tipo `string`. Essa especificação de tipo permite ao *TypeScript* fornecer verificação de tipo adicional e auxiliar os desenvolvedores no desenvolvimento.

```
function fetchData(): Promise<string> {
    return new Promise((resolve, reject) => {
        // Logica assincrona aqui
        // Se a operacao for bem-sucedida, chame resolve(valor)
        // Se a operacao falhar, chame reject(erro)
    });
}
```

Listagem 2.44: Criação de uma *Promise* em *TypeScript*

Para encadear operações assíncronas com *Promises* em *JavaScript*, os métodos `then` e `catch` são úteis. O método `then` é chamado quando a *Promise* é cumprida, recebendo o valor resolvido como argumento. O método `catch` é chamado quando a *Promise* é rejeitada, recebendo o erro como argumento. Da mesma forma que em *JavaScript*, pode-se encadear operações assíncronas com *Promises* em *TypeScript* usando os métodos `then` e `catch`. A diferença é que o *TypeScript* pode inferir e aplicar os tipos corretos automaticamente, tornando o código mais legível e evitando erros de tipo.

É possível observar um exemplo na Listagem 2.45, onde a função `fetchData` retorna uma *Promise* que resolve uma string simulando a obtenção de dados assíncronos. Em seguida, o método `then` é encadeado para lidar com o valor resolvido e o método `catch` para tratar qualquer erro que possa ocorrer durante a execução da *Promise*. O *TypeScript* garante que o valor retornado pela *Promise* seja do tipo especificado (`string` neste caso) e também fornece verificação de tipo nos argumentos das funções passadas para `then` e `catch`, evitando erros de tipo durante a compilação.

```
function fetchData(): Promise<string> {
    return new Promise((resolve, reject) => {
        // Simulando uma operacao assincrona
        setTimeout(() => {
            const data = 'Dados obtidos com sucesso!';
            resolve(data);
        }, 2000);
    });
}

fetchData()
.then((data) => {
    console.log(data);
})
.catch((error) => {
    console.error(error);
});
```

Listagem 2.45: Exemplo de uso de *Promises* em *TypeScript*

As *Promises* são uma ferramenta essencial para lidar com operações assíncronas em *JavaScript* e *TypeScript*. O *TypeScript*, com sua tipagem estática e recursos avançados de inferência de tipos, oferece uma experiência de desenvolvimento ainda mais robusta ao trabalhar com *Promises*, permitindo detectar erros de tipo em tempo de compilação e fornecendo melhor documentação e maior compreensão do código.

2.13 *Callback*

Callbacks são um padrão de programação muito comum em *JavaScript* e *TypeScript* para lidar com operações assíncronas e chamadas de retorno. Uma função de *callback* é uma função que é passada como argumento para outra função e é executada posteriormente, geralmente quando uma operação assíncrona é concluída ou um evento ocorre. Os *callbacks* são amplamente utilizados para lidar com operações assíncronas, como chamadas de API, leitura de arquivos e requisições HTTP. Um exemplo comum é o uso do método `setTimeout`, que executa uma função de *callback* após um determinado intervalo de tempo.

Como exemplificado na Listagem 2.46, há a função `fetchData` que simula uma operação assíncrona usando `setTimeout`. Ela recebe uma função de *callback* como argumento e, quando a operação é concluída, chama a função de *callback* passando o resultado como parâmetro. A função `processarDados` é passada como *callback* e é responsável por processar os dados obtidos ou lidar com erros, caso ocorram.

```
function fetchData(callback) {
    // Simulando uma operacao assincrona
    setTimeout(() => {
        const data = 'Dados obtidos com sucesso!';
        // Chamando a funcao de callback
        callback(null, data);
    }, 2000);
}

function processarDados(error, data) {
    if (error) {
        console.error('Ocorreu um erro:', error);
        return;
    }
    console.log('Dados processados:', data);
}
// Passando a funcao de callback como argumento
fetchData(processarDados);
```

Listagem 2.46: Exemplo de uso de *callback* em *JavaScript*

O *TypeScript*, como um *superset* do *JavaScript*, também oferece suporte total aos *callbacks*. No entanto, com a adição de recursos avançados de tipagem estática, o *TypeScript* permite maior segurança e clareza ao trabalhar com *callbacks*. Na Listagem 2.47, adicionam-se as anotações de tipo ao *callback* e aos parâmetros das funções `fetchData` e `processarDados`. O *TypeScript* infere e valida os tipos corretos, proporcionando verificação de tipos adicional durante o desenvolvimento. Além disso, é possível utilizar tipos mais complexos para os parâmetros e o retorno das funções de *callback*, proporcionando uma maior expressividade e segurança ao código.

```
function fetchData(callback: (error: Error
| null, data: string) => void) {
    // Simulando uma operacao assincrona
    setTimeout(() => {
        const data = 'Dados obtidos com sucesso!';
        callback(null, data); // Chamando a funcao de callback
    }, 2000);
}

function processarDados(error: Error | null, data: string) {
    if (error) {
        console.error('Ocorreu um erro:', error);
        return;
    }
    console.log('Dados processados:', data);
}
// Passando a funcao de callback como argumento
fetchData(processarDados);
```

Listagem 2.47: Exemplo de uso de *callback* em *TypeScript*

Callbacks são uma forma poderosa de lidar com operações assíncronas e chamadas de retorno em *JavaScript* e *TypeScript*. Eles permitem que você controle o fluxo de execução e lide com o resultado de uma operação de forma assíncrona. No entanto, o uso excessivo de *callbacks* pode levar a um código complexo e de difícil manutenção, conhecido como *callback hell*. Para evitar o *callback hell* e melhorar a legibilidade do código, outras abordagens, como *Promises* e `async/await`, foram introduzidas nas versões mais recentes do *JavaScript* e do *TypeScript*. Essas abordagens oferecem uma sintaxe mais limpa e uma melhor estrutura para lidar com operações assíncronas.

2.14 Arrays

Arrays são estruturas de dados que permitem armazenar e manipular coleções de elementos. Eles podem ser usados para armazenar uma lista de valores, como

números, strings, objetos e outros tipos de dados. Para inicializar um array em *JavaScript* e *TypeScript*, usa-se colchetes para delimitar a sequência de elementos da estrutura. A Listagem 2.48 apresenta a sintaxe de declaração de um array e exemplifica arrays com diferentes tipos de elementos. Na segunda linha é declarado um *array* de inteiros `numeros`, na quarta linha é declarado um *array* de strings `nomes` e na sexta linha é declarado um *array* vazio `arr_vazio`.

```
1 // array de inteiros
2 let numeros = [1, 2, 3, 4, 5];
3 // array de strings
4 let nomes = ["Pedro", "Maria", "Felipe"];
5 // array vazio
6 let arr_vazio = [];
```

Listagem 2.48: Sintaxe de Inicialização de Array em JS e TS.

Como já mencionado, um array se trata de uma cadeia de elementos, portanto, não é possível acessar um valor específico simplesmente chamando o nome da variável que o armazena. Para acessar um elemento de um array utiliza-se o índice, que é a posição de um elemento dentro de um array. Os índices de um array em *JavaScript* e *TypeScript* começam em zero, ou seja, zero é a primeira posição do array. Na Listagem 2.49 é possível observar essa indexação de um array. No exemplo é definido um array chamado `numeros`, o índice 0 desse array é seu primeiro valor, que é o elemento 1, o índice 2 se refere ao seu terceiro elemento, que é 3, e o índice 4 do array indica seu último índice no exemplo, o quinto elemento, que é o número 5.

```
1 let numeros = [1, 2, 3, 4, 5];
2 console.log(numeros[0]); // imprime 1
3 console.log(numeros[2]); // imprime 3
4 console.log(numeros[4]); // imprime 5
```

Listagem 2.49: Exemplo de Indexação de Array em JS e TS.

2.15 Coleções Chaveadas

As coleções chaveadas são estruturas de dados que permitem armazenar e manipular conjuntos de elementos em *JavaScript* e *TypeScript* de forma eficiente. Duas das principais coleções chaveadas disponíveis nessas linguagens são os Mapas (*Map*) e os Conjuntos (*Set*).

Mapas (*Map*): Os Mapas são estruturas que permitem armazenar pares de chave-valor, onde cada chave é exclusiva e associada a um valor correspondente. Isso significa que os Mapas permitem mapear valores a partir de uma chave específica. Em *JavaScript* e *TypeScript*, a classe `Map` é usada para criar e manipular Mapas.

Para criar um Mapa vazio, pode-se usar o construtor da classe Map. Em seguida, é possível adicionar elementos ao Mapa usando o método `set(chave, valor)`, onde a chave representa a chave do elemento e o valor é o valor associado a essa chave. Na Listagem 2.50, um Mapa é criado e dois elementos são adicionados a ele. A chave "chave1" está associada ao valor "valor1", e a chave "chave2" está associada ao valor "valor2".

```
1 const mapa = new Map();
2 mapa.set('chave1', 'valor1');
3 mapa.set('chave2', 'valor2');
```

Listagem 2.50: Exemplo de utilização de Mapa em JS e TS.

Para acessar um valor específico no Mapa, pode-se usar o método `get(chave)`, passando a chave desejada como parâmetro. Para exemplificar, na Listagem 2.51, é feito o acesso ao valor associado à chave "chave1", resultando na exibição do valor "valor1" no console, ainda como base no exemplo da Listagem 2.50. Além disso, os Mapas também possuem métodos para verificar a presença de uma chave (`has(chave)`), remover um elemento (`delete(chave)`) e obter a quantidade de elementos no Mapa (`size`). Essas funcionalidades tornam os Mapas extremamente úteis para armazenar e recuperar informações com base em chaves únicas.

```
1 console.log(mapa.get('chave1')); // Saida: 'valor1'
```

Listagem 2.51: Acessando um valor específico em um Mapa.

Conjuntos (*Set*): Os Conjuntos são estruturas que permitem armazenar elementos únicos, sem a necessidade de chaves associadas. Em outras palavras, um Conjunto contém apenas valores distintos, eliminando automaticamente duplicatas. Em *JavaScript* e *TypeScript*, a classe `Set` é usada para criar e manipular conjuntos.

Na Listagem 2.52, um Conjunto é criado a partir da classe `Set` com o comando "`new Set()`'' e o Conjunto é atribuído à variável `conjunto`. Dois elementos são adicionados a ele, "elemento1" e "elemento2", com o comando `add(elemento)`. O Conjunto garante que apenas valores únicos sejam armazenados, portanto, se um elemento duplicado for adicionado, ele será ignorado.

```
1 const conjunto = new Set();
2 conjunto.add('elemento1');
3 conjunto.add('elemento2');
```

Listagem 2.52: Exemplo de utilização de Conjunto em JS e TS.

Na Listagem 2.53, o elemento "elemento1" é adicionado duas vezes ao Conjunto. No entanto, como os Conjuntos não permitem elementos duplicados, apenas uma ocorrência desse elemento será mantida no Conjunto, resultando em um tamanho de 2. Ademais, os Conjuntos também possuem métodos úteis para verificar a existência

de um elemento (`has(elemento)`), remover um elemento (`delete(elemento)`) e obter a quantidade de elementos no Conjunto (`size`).

```
const conjunto = new Set();
conjunto.add('elemento1');
conjunto.add('elemento2');
conjunto.add('elemento1');
// Elemento duplicado eh removido automaticamente
console.log(conjunto.size); // Saida: 2
```

Listagem 2.53: Removendo elementos duplicados em um Conjunto.

Em resumo, os Mapas e os Conjuntos são poderosas coleções chaveadas que permitem armazenar e manipular dados de forma eficiente em *JavaScript* e *TypeScript*. Os Mapas são ideais quando é necessário mapear valores a partir de chaves únicas, enquanto os Conjuntos são adequados para armazenar elementos únicos sem a necessidade de associação chave-valor.

2.16 Classes e Objetos

As classes e objetos são conceitos fundamentais na programação orientada a objetos (POO). Em *JavaScript* e *TypeScript*, é possível criar e manipular classes para representar entidades e estruturar o código de forma mais organizada e reutilizável. Uma classe é uma estrutura que define um conjunto de atributos e métodos relacionados. Os atributos representam as características dos objetos da classe, enquanto os métodos são as ações que esses objetos podem executar. Em *JavaScript* e *TypeScript*, os atributos e métodos são definidos dentro da declaração da classe.

Na Listagem 2.54 é possível observar a criação de uma classe `Carro` em *JavaScript* e *TypeScript*. A classe `Carro` possui dois atributos: `cor`, do tipo `string`, e `rodas`, do tipo `number`. Além disso, ela possui um método `informacoes()` que imprime uma mensagem contendo a cor e a quantidade de rodas do carro. Isso quer dizer que a classe `Carro` encapsula as propriedades e comportamentos que serão passados às instâncias (objetos) dessa classe.

```
class Carro {
    cor = "vermelho";
    rodas = 4;
    informacoes() {
        console.log(`Sou um carro ${this.cor}
        e tenho ${this.rodas} rodas.`);
    }
}
```

Listagem 2.54: Exemplo básico de classe em JS e TS.

É importante destacar que o *TypeScript* oferece suporte a tipos de dados estáticos, permitindo que você especifique os tipos dos atributos e parâmetros de métodos. Por exemplo, no *TypeScript*, dessa forma, é garantido que a propriedade `cor` seja sempre uma `string` e a propriedade `rodas` seja sempre do tipo `number`. Essa verificação estática de tipos proporcionada pelo *TypeScript* ajuda a evitar erros e a facilitar a compreensão e manutenção do código.

A instanciação de objetos é o processo de criar uma instância ou um objeto específico de uma classe. Em *JavaScript* e *TypeScript*, isso é feito usando o operador `new`, seguido do nome da classe e de quaisquer argumentos necessários pelo construtor. Como ilustra a Listagem 2.55, é criada uma instância da classe `Carro` usando o operador `new` e essa instância é atribuída à variável `meuCarro`. Em seguida, o método `informacoes()` é chamado na instância `meuCarro` para exibir uma mensagem contendo a cor e a quantidade de rodas do carro.

```
const meuCarro = new Carro();
meuCarro.informacoes();
// Saida: "Sou um carro vermelho e
//tenho 4 rodas."
```

Listagem 2.55: Instanciação de objetos em JS e TS.

A instanciação de objetos permite criar múltiplas instâncias da mesma classe, cada uma com seus próprios valores de atributos. Isso é útil quando se deseja trabalhar com diferentes objetos que compartilham a mesma estrutura e comportamento definidos pela classe. Por isso, ao criar uma classe, a fim de torná-la versátil pode-se definir uma função construtora para ela, que pode depender de parâmetros, ou não. Em *JavaScript* e *TypeScript*, isso pode ser feito usando o método especial `constructor()`. Esse método é executado automaticamente quando um objeto da classe é criado.

Na Listagem 2.56, o método `constructor()` recebe dois parâmetros: cor e rodas. Dentro do inicializador, esses valores são atribuídos aos atributos correspondentes do objeto usando a palavra-chave `this`. Agora, ao criar um objeto Carro, é necessário fornecer os valores para `cor` e `idade` durante a instanciação. A partir disso, cada

instância de `Carro` terá sua própria cor e quantidade de rodas, permitindo a criação e manipulação de vários carros independentes uns dos outros.

```
class Carro {
    constructor(cor, rodas) {
        this.cor = cor;
        this.rodas = rodas;
    }
    informacoes() {
        console.log(`Sou um carro ${this.cor}
        e tenho ${this.rodas} rodas.`);
    }
}
```

Listagem 2.56: Adicionando inicializador de objetos em JS.

O *TypeScript*, sendo uma linguagem de programação baseada em *JavaScript*, adiciona a verificação estática de tipos durante a compilação, garantindo que os valores fornecidos durante a instanciação estejam corretamente tipados de acordo com a definição da classe. a Listagem 2.57 mostra o mesmo exemplo da Listagem 2.56, porém, adaptado para *TypeScript*, explicitando os tipos `string` e `number` para os atributos `cor` e `rodas`, respectivamente, além de definir também o tipo `void` para o retorno do método `informacoes`.

```
class Carro {
    cor: string;
    rodas: number;
    constructor(cor: string, rodas: number) {
        this.cor = cor;
        this.rodas = rodas;
    }
    informacoes(): void {
        console.log(`Sou um carro ${this.cor}
        e tenho ${this.rodas} rodas.`);
    }
}
```

Listagem 2.57: Adicionando inicializador de objetos em TS.

Em *JavaScript*, é comum usar uma função construtora para criar objetos de uma classe. Uma função construtora é uma função regular que é invocada usando o operador new e retorna um objeto. Essa abordagem é útil quando se deseja definir membros privados para a classe. Na Listagem 2.58, a função `Pessoa` serve como função construtora. Dentro dela, define-se os atributos `nome` e `idade` como propriedades do objeto usando `this`. Também é definido o método `falar()` como uma função do objeto. Ao criar um novo objeto `Pessoa` usando o operador `new`, a função construtora é invocada e retorna um objeto com as propriedades e métodos

definidos. Essa maneira também é válida em *TypeScript*, no entanto, o uso da sintaxe de classe com a palavra-chave `class` é recomendado para aproveitar os recursos adicionais de verificação de tipo e recursos avançados de POO oferecidos pelo *TypeScript*.

```
function Pessoa(nome, idade) {
    this.nome = nome;
    this.idade = idade;
    this.falar = function() {
    console.log(`Ola, meu nome eh ${this.nome}
    e tenho ${this.idade} anos.`);
    };
}
const pessoa = new Pessoa("Joao", 25);
pessoa.falar();
// Saida: Ola, meu nome eh Joao e tenho 25 anos.
```

Listagem 2.58: Exemplo de função construtora em JS.

Além dessas formas mencionadas, há também o método `Object.create()`, que é outra forma de criar objetos a partir de uma classe em *JavaScript* e *TypeScript*. Ele permite criar um novo objeto com um protótipo especificado. O protótipo é um objeto existente que será usado como base para o novo objeto. Por exemplo, na Listagem 2.59 é definido o objeto `pessoaProto` que contém o método `falar()`. Ao chamar `Object.create(pessoaProto)`, cria-se um novo objeto pessoa que tem `pessoaProto` como protótipo. Em seguida, os atributos nome e idade no objeto pessoa são definidos. Dessa forma, o objeto pessoa herda o método `falar()` do protótipo.

```
const pessoaProto = {
    falar: function() {
        console.log(`Ola, meu nome eh ${this.nome}
        e tenho ${this.idade} anos.`);
    }
};

const pessoa = Object.create(pessoaProto);
pessoa.nome = "Maria";
pessoa.idade = 30;
pessoa.falar();
// Saida: Ola, meu nome eh Maria e tenho 30 anos.
```

Listagem 2.59: Utilizando `Object.create` para criar objetos em JS.

Em *JavaScript*, não há um suporte nativo para membros privados em classes. No entanto, é possível simular membros privados usando convenções de nomenclatura ou recursos avançados do *TypeScript*, como modificadores de acesso. Na Listagem 2.60,

os atributos `nome` e `idade` são prefixados com `#`, indicando que eles são membros privados da classe. Essa convenção de nomenclatura é uma prática comum para indicar que esses atributos devem ser tratados como privados e não devem ser acessados diretamente fora da classe. Em *JavaScript* pode ser usado apenas como convenção, e não impede de fato o acesso a esses elementos, diferentemente do *TypeScript*, que emitirá um erro ao tentar acessá-los.

```
class Pessoa {
    #nome: string;
    #idade: number;
    constructor(nome: string, idade: number) {
        this.#nome = nome;
        this.#idade = idade;
    }
    falar(): void {
        console.log(`Ola, meu nome eh ${this.#nome}
        e tenho ${this.#idade} anos.`);
    }
}
let a = new Pessoa("Joana", 22);
console.log(a.#nome);
// Saida em JS: "Joana"
/* Saida em TS: "Error: Property '#nome' is not
accessible outside class 'Pessoa' because it has a
private identifier." */
```

Listagem 2.60: Simulação de membros privados usando convenção de nomenclatura em JS.

O *TypeScript* também oferece modificadores de acesso como o `public`, `private` e `protected` para controlar o acesso aos membros da classe. Esses modificadores fornecem um nível adicional de segurança e encapsulamento em relação ao *JavaScript* puro. A Tabela 2.3 relaciona os três modificadores de acesso em *TypeScript* com suas respectivas usabilidades, dentre eles, o `private`, que deixa o membro acessível somente dentro da própria classe. a Listagem 2.61 mostra o mesmo exemplo da Listagem 2.60, utilizando o comando `private` do *TypeScript*.

Modificador de Acesso	Significado
`private`	O membro (propriedade ou método) é acessível apenas dentro da própria classe em que foi declarado. Não é possível acessá-lo fora da classe ou em classes derivadas.
`public`	O membro é acessível em qualquer lugar, ou seja, em qualquer classe ou parte do código.
`protected`	O membro é acessível dentro da própria classe em que foi declarado e em classes derivadas dessa classe. Não é possível acessá-lo fora desses contextos.

Tabela 2.3: Modificadores de Acesso em *TypeScript*

```
class Pessoa {
    private nome: string;
    private idade: number;
    constructor(nome: string, idade: number) {
        this.nome = nome;
        this.idade = idade;
    }
    falar(): void {
        console.log(`Ola, meu nome eh ${this.nome}
        e tenho ${this.idade} anos.`);
    }
}
let a = new Pessoa("Joana", 22);
console.log(a.nome);
/* Error: Property 'nome' is private and only
accessible within class 'Pessoa'. */
```

Listagem 2.61: Propriedades privadas em TS.

Os membros estáticos em uma classe são aqueles que pertencem à classe em si, em vez de pertencerem a instâncias individuais da classe. Em *JavaScript* e *TypeScript*, os membros estáticos são definidos usando a palavra-chave `static`. Os membros estáticos são úteis quando se deseja compartilhar informações ou comportamentos que se aplicam à classe como um todo, em vez de instâncias individuais. Por exemplo, é possível utilizar um membro estático para armazenar configurações globais ou criar métodos utilitários que não dependem de um objeto específico.

Na Listagem 2.62, é apresentado um exemplo de como adicionar um membro estático em *JavaScript*. O membro estático `pais` é adicionado à classe `Pessoa`, representando o país padrão de uma pessoa. O membro estático é acessado usando

o nome da classe, ou seja, `Pessoa.pais`. Além disso, é definido um método estático `mudarPais()`, que permite alterar o país padrão. A classe `Pessoa` também possui uma propriedade `nome` e uma propriedade `idade`, que são específicas de cada instância de pessoa. Porém, o membro `pais` é compartilhado por todas as instâncias, pois é estático. Isso significa que todas as instâncias de `Pessoa` terão acesso ao mesmo valor de `pais`. O método `falar()` imprime uma mensagem com o nome, idade e país da pessoa, utilizando o membro estático `pais`. No caso do *TypeScript*, a sintaxe correta deve incluir os tipos dos parâmetros e dos membros da classe explicitamente.

```
class Pessoa {
    nome;
    idade;
    static pais = "Brasil";
    constructor(nome, idade) {
        this.nome = nome;
        this.idade = idade;
    }
    falar() {
        console.log(`Ola, meu nome eh ${this.nome}
        e tenho ${this.idade} anos. Sou do(a) ${Pessoa.pais}.`);
    }
    static mudarPais(novoPais){
        Pessoa.pais = novoPais;
    }
}
```

Listagem 2.62: Adicionando membro estático em *JavaScript*.

É importante saber que é possível acessar um membro estático sem a necessidade de criar uma instância da classe. Usando ainda o exemplo da Listagem 2.62, para mudar o valor do país padrão, pode-se chamar o método estático `mudarPais()` diretamente na classe, utilizando `Pessoa.mudarPais("Novo País")`. Isso afetará todas as instâncias de `Pessoa`, as já criadas e as que serão criadas.

2.16.1 Herança de Classes

A herança é um conceito importante na POO que permite criar novas classes com base em classes existentes. A classe que serve como base para a criação de novas classes é chamada de classe pai ou superclasse, e as classes criadas a partir dela são chamadas de classes filhas ou subclasses. Na Listagem 2.63, a classe `Estudante` herda da classe `Pessoa`, definida na Listagem 2.62, usando a palavra-chave `extends`. A classe filha possui um novo atributo matricula e um novo método `estudar()`. No construtor da classe filha, chama-se o construtor da classe pai usando `super(nome, idade)` para garantir que os atributos da classe pai sejam corretamente inicializados.

Para *TypeScript* a lógica é a mesma, contudo, como já enfatizado, deve-se realizar a especificação explícita dos tipos para os métodos e atributos.

```
class Estudante extends Pessoa {
    constructor(nome, idade, matricula) {
        super(nome, idade);
        this.matricula = matricula;
    }

    estudar() {
        console.log(`${this.nome}
        esta estudando. Matricula: ${this.matricula}`);
    }
}
```

Listagem 2.63: Herança de classes em *JavaScript*

A herança de classes permite reutilizar a estrutura e o comportamento definidos na classe pai, evitando a duplicação de código. Além disso, as classes filhas podem adicionar novos atributos e métodos específicos ou substituir os existentes, personalizando o comportamento conforme necessário. Também é importante ressaltar que o *TypeScript* fornece um suporte mais robusto para herança de classes, incluindo a verificação de tipos durante a compilação e a capacidade de definir modificadores de acesso para membros herdados.

2.17 Interfaces

As interfaces permitem definir contratos ou estruturas para objetos e garantem que os objetos tenham certos métodos ou propriedades. Em *JavaScript*, as interfaces não são uma construção nativa da linguagem, mas podem ser simuladas por meio de comentários ou convenções de nomenclatura, como mostra a Listagem 2.64. No exemplo, é criada uma função vazia chamada `PessoaInterface` e são adicionados a ela propriedades e métodos usando o prototype. Essa função pode ser usada como uma interface para objetos que devem ter as propriedades `nome` e `idade`, bem como o método `saudacao()`. É importante observar que essa é apenas uma convenção e não há um mecanismo nativo para verificar a implementação de uma interface em *JavaScript*.

```
// Definindo uma interface para um objeto 'Pessoa'
function PessoaInterface() {}
PessoaInterface.prototype.nome = '';
PessoaInterface.prototype.idade = 0;
PessoaInterface.prototype.saudacao = function() {};
```

Listagem 2.64: Simulação de Interface em JS.

Em contraste, o *TypeScript* tem suporte nativo a interfaces e fornece verificação estática de tipos durante a compilação. As interfaces no *TypeScript* são usadas para definir a forma ou estrutura de um objeto. Como é possível observar na Listagem 2.65, é definida uma interface chamada Pessoa que especifica que um objeto do tipo Pessoa deve ter uma propriedade `nome` do tipo `string`, uma propriedade `idade` do tipo `number` e um método `saudacao()` que não retorna nenhum valor (`void`).

```
// Definindo uma interface para um objeto 'Pessoa'
interface Pessoa {
    nome: string;
    idade: number;
    saudacao(): void;
}
```

Listagem 2.65: Exemplo de Interface em TS.

Pode-se utilizar interfaces para garantir que objetos tenham a estrutura desejada. Na Listagem 2.66 há um exemplo de como usar a interface `Pessoa` em *TypeScript*. No exemplo, a função saudar recebe um parâmetro do tipo `Pessoa` e imprime uma saudação com base nas propriedades desse objeto. Em seguida, foi criada uma variável `pessoa1` que segue a estrutura definida pela interface `Pessoa`, e esse objeto foi passado para a função `saudar`, que chama o método `saudacao()`. Nesse contexto, o *TypeScript* verifica se o objeto `pessoa1` cumpre os requisitos da interface `Pessoa`. Ao adicionar ou remover alguma propriedade ou método necessários pela interface, o *TypeScript* informará sobre o erro durante a compilação.

```
function saudar(pessoa: Pessoa) {
    console.log(`Ola, meu nome eh ${pessoa.nome}
    e tenho ${pessoa.idade} anos.`);
}
let pessoa1: Pessoa = {
    nome: 'Joao',
    idade: 30,
    saudacao: function() {
        console.log('Ola, tudo bem?');
    }
};
saudar(pessoa1);
// Saida: "Ola, meu nome eh Joao e tenho 30 anos."
pessoa1.saudacao();
// Saida: "Ola, tudo bem?"
```

Listagem 2.66: Utilização de Interfaces em TS.

Violar uma interface em *TypeScript* resultará em erros. Na Listagem 2.67, a interface `Animal` é definida com as propriedades `nome` (do tipo `string`) e `idade` (do tipo `number`). No entanto, ao criar o objeto `gato`, o valor atribuído à propriedade `idade` é uma `string` ao invés de um número. Nesse caso, o *TypeScript* detectará o erro durante a compilação e exibirá uma mensagem de erro indicando a incompatibilidade de tipos.

```
interface Animal {
    nome: string;
    idade: number;
}
let gato: Animal = {
    nome: 'Bella',
    idade: '2 anos'
    // Tipo incorreto para a propriedade 'idade'
};
```

Listagem 2.67: Exemplo de erro de tipo ao violar uma interface

Ademais, ao tentar acessar uma propriedade ou chamar um método que não esteja definido na interface, o *TypeScript* também nos alertará sobre o erro. Na Listagem 2.68, a interface `Pessoa` define as propriedades `nome` e `idade`, bem como o método `saudacao()`. No entanto, ao criar o objeto `pessoa`, adiciona-se um método chamado `cumprimento` que não está definido na interface. O *TypeScript* exibirá um erro durante a compilação, informando que a propriedade `cumprimento` não é válida.

```
interface Pessoa {
    nome: string;
    idade: number;
    saudacao(): void;
}
let pessoa: Pessoa = {
    nome: 'Joao',
    idade: 30,
    // Metodo nao definido na interface
    cumprimento: function() {
        console.log('Ola!');
    }
};
pessoa.cumprimento();
// Erro: 'cumprimento' nao eh uma funcao valida
```

Listagem 2.68: Exemplo de erro ao acessar propriedade e método não definidos na interface

Os exemplos mencionados evidenciam a relevância das verificações de tipo fornecidas pelo *TypeScript* para garantir a conformidade das interfaces. Ao realizar verificações de tipo durante a compilação, o *TypeScript* auxilia na detecção antecipada de erros, contribuindo para prevenir problemas e aprimorar a segurança e a robustez do código. Essa abordagem estática de verificação de tipos permite identificar inconsistências e erros de atribuição de valores em tempo de compilação, o que evita falhas e comportamentos inesperados durante a execução do programa.

2.18 Aplicação e Contraste entre *JavaScript* e *TypeScript*

A fim de utilizar as linguagens duas linguagens, *JavaScript* e *TypeScript*, em uma aplicação para destacar os principais contrastes entre ambas, um jogo inspirado no jogo Flappy Bird é usado como exemplo nesta seção.

Na Listagem 2.69 tem-se o código *HTML* básico do jogo Flappy Bird. O jogo é definido dentro das tags `<html>` e `</html>`. No cabeçalho (`<head>`), tem-se a inclusão de um arquivo *JavaScript* chamado "script.js" usando a tag `<script>`. Esse arquivo contém o código *JavaScript* responsável pela lógica do jogo. Também temos a definição do título da página como "Flappy Bird" e a inclusão de um arquivo de estilo chamado "style.css" usando a tag `<link>`. No corpo (`<body>`), há dois elementos `<div>`: um com o id "bird" que representa o pássaro do jogo e outro com o id "score" que exibe a pontuação do jogador.

```
<!DOCTYPE html>
<html>
<head>
    <script src="script.js" defer></script>
    <title>Flappy Bird</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <div id="bird"></div>
    <div id="score">Score: 0</div>
</body>
</html>
```

Listagem 2.69: HTML do jogo

Na Listagem 2.70 está o código CSS responsável pela estilização do jogo. Inicialmente, definem-se algumas propriedades do elemento `body` para remover margens e espaçamentos indesejados. Em seguida, há estilização do elemento `#bird`, que representa o pássaro do jogo. Ele é posicionado de forma absoluta (`position:absolute`) e definido com um tamanho de `50px` por `50px` (`width:50px; height:50px`). A cor de fundo do pássaro é amarela (`background-color:yellow`). Ainda há a estilização da classe `.pipe`, que representa os canos do jogo. Os canos também são posicionados de forma absoluta e têm uma largura de `80px` e altura de `300px`. A cor de fundo

dos canos é verde (`background-color:green`). Por fim, a estilização do elemento `#score`, que exibe a pontuação do jogador. Ele é posicionado de forma absoluta, com uma posição no topo esquerdo da tela (`top:10px; left:10px`). O tamanho da fonte é definido como `24px` (`font-size:24px`).

```
body{
    overflow: hidden;
    padding: 0;
    margin: 0;
}
#bird {
    position: absolute;
    top: 50%;
    left: 50px;
    width: 50px;
    height: 50px;
    background-color: yellow;
}
.pipe {
    position: absolute;
    width: 80px;
    height: 300px;
    background-color: green;
}
#score {
    position: absolute;
    top: 10px;
    left: 10px;
    font-size: 24px;
}
```

Listagem 2.70: CSS do jogo

No trecho de código da Listagem 2.71 observa-se a declaração de várias variáveis globais utilizadas no jogo. Essas variáveis são utilizadas ao longo do código para controlar o funcionamento do jogo. As variáveis globais são:

- `bird`: uma variável que recebe o elemento *HTML* com o id "bird" (o pássaro do jogo).
- `score`: uma variável que recebe o elemento *HTML* com o id "score" (o elemento que exibe a pontuação do jogador).
- `pipes`: uma matriz vazia para armazenar os canos do jogo.
- `gameSpeed`: uma variável que define a velocidade do jogo.
- `maxGravity`: uma constante que define a gravidade máxima.
- `gravity`: uma variável que define o valor atual da gravidade.
- `jumpHeight`: uma constante que define a altura do salto do pássaro.

- `scoreCount`: uma variável que representa a contagem da pontuação do jogador.

```
1  // Selecionando os elementos HTML necessarios
2  // elemento com id 'bird'
3  const bird = document.getElementById('bird');
4  // elemento com id 'score'
5  const score = document.getElementById('score');
6  const pipes = []; // Array para armazenar os canos
7  let gameSpeed = 5; // Velocidade do jogo
8  const maxGravity = 10; // Gravidade maxima
9  let gravity = 10; // Valor da gravidade atual
10 const jumpHeight = -30; // Altura do salto
11 let scoreCount = 0; // Contador de pontuacao
```

Listagem 2.71: Projeto exemplo em JS (Parte 1): Declaração de variáveis globais

A definição da função pipeMovement() é feita na Listagem 2.72, responsável por mover os canos do jogo e verificar as colisões com o pássaro. Dentro da função, há um *loop* que itera sobre todos os canos presentes na matriz pipes. Essa função é chamada repetidamente durante a execução do jogo para movimentar os canos e verificar as colisões. Para cada cano, a função realiza as seguintes ações:

- Linha 5: Move o cano para a esquerda de acordo com a velocidade do jogo.
- Linhas 7 a 10: Verifica se houve colisão entre o pássaro e o cano usando a função `isColliding(bird, pipe)`. Caso haja colisão, a função `gameOver()` é chamada para encerrar o jogo.
- Linhas 12 a 16: Verifica se o cano já passou pelo pássaro. Isso é feito comparando a posição do cano com a posição do pássaro. Se o cano já passou pelo pássaro e a propriedade `passed` do cano é falsa, significa que o jogador ganhou um ponto. Nesse caso, a pontuação é incrementada e exibida no elemento score.

```
function pipeMovement(){
  // Loop para mover e verificar colisoes dos canos
  for (const pipe of pipes) {
    // Movimenta os canos na velocidade do jogo
    pipe.style.left = `${pipe.offsetLeft - gameSpeed}px`;
    // Verifica se o passaro colidiu com o cano
    if (isColliding(bird, pipe)) {
      gameOver();
      callback(()=>{return;});
    }
    // Verifica se o cano ja passou pelo passaro
    if (pipe.offsetLeft + pipe.offsetWidth <
    bird.offsetLeft && !pipe.passed) {
      pipe.passed = true;
      scoreCount++; // aumenta a pontuacao
      score.innerHTML = `Score: ${scoreCount / 2}`;
    }
  }
}
```

Listagem 2.72: Projeto exemplo em JS (Parte 2): Função de movimentação dos canos

Já na Listagem 2.73 a função `pipeSpawn()` é definida, responsável por gerar novos canos e remover os canos que já passaram da tela. Essa função é chamada repetidamente durante a execução do jogo para garantir que sempre haja um número mínimo de canos na tela. Dentro da função, são realizadas as seguintes ações:

- Linhas 3 a 5: Verifica se o número de canos presentes na matriz `pipes` é menor que 4. Se for, chama a função `generatePipes()` para gerar novos canos.
- Linhas 7 a 10: Verifica se o primeiro cano da matriz `pipes` já passou da tela (se a posição do cano mais a largura do cano for menor que zero). Se sim, remove o primeiro cano da matriz usando o método `shift()`. Esse método remove o primeiro elemento do *array*.

```
function pipeSpawn(){
  // Gera 2 novos canos por vez
  if (pipes.length < 2) {
    generatePipes();
  }
  // Remove os canos que ja passaram da tela
  if (pipes.length > 0 && pipes[0].offsetLeft +
  pipes[0].offsetWidth < 0) {
    pipes.shift();
  }
}
```

Listagem 2.73: Projeto exemplo em JS (Parte 3): Função de surgimento de canos

Uma outra função, `applyGravity()`, é declara na Listagem 2.74, responsável por aplicar a gravidade ao pássaro do jogo. Essa função é chamada repetidamente durante a execução do jogo para simular a queda do pássaro. Dentro da função, são realizadas as seguintes ações:

- Linhas 3 a 5: Verifica se a gravidade ainda não atingiu o valor máximo. Se não atingiu, incrementa o valor da gravidade em 2 unidades.
- Linha 7: Move o pássaro para baixo com base na gravidade.

```
function applyGravity(){
  // Aumenta a gravidade ate o maximo
  if (gravity < maxGravity) {
    gravity += 2;
  }
  // Move o passaro para baixo com base na gravidade
  bird.style.top = `${bird.offsetTop + gravity}px`;
}
```

Listagem 2.74: Projeto exemplo em JS (Parte 4): Função de aplicação da gravidade

Na Listagem 2.75 é possível observar a declaração da função `verifyEdges()`, responsável por verificar se o pássaro atingiu os limites da tela, sendo chamada repetidamente durante a execução do jogo. A função verifica se a posição vertical do pássaro, somada à sua altura, é maior que a altura da janela ou se a posição vertical do pássaro é menor que zero. Se alguma dessas condições for verdadeira, significa que o pássaro atingiu os limites da tela. E assim, em caso de colisão com os limites da tela, a função `gameOver()` é chamada para encerrar o jogo.

```
function verifyEdges(){
  if (bird.offsetTop + bird.offsetHeight > window.innerHeight ||
  bird.offsetTop < 0) {
    gameOver();
    callback(()=>{return;});
  }
}
```

Listagem 2.75: Projeto exemplo em JS (Parte 5): Função de verificação de bordas da tela

Na Listagem 2.76 tem-se a definição da função `gameLoop()`, que é a função principal responsável por coordenar o funcionamento do jogo. Dentro da função, são chamadas as seguintes funções em sequência:

- `applyGravity()`: para aplicar a gravidade ao pássaro.
- `pipeMovement()`: para mover os canos.
- `pipeSpawn()`: para gerar novos canos.
- `verifyEdges()`: para verificar se o pássaro atingiu os limites da tela.

Após a execução dessas funções, a função `gameLoop()` é chamada novamente usando `requestAnimationFrame()`. Essa função cria um *loop* de animação que chama a função `gameLoop()` a cada novo *frame*. Essa função é chamada uma vez para iniciar o jogo e, em seguida, é chamada repetidamente para manter o jogo em execução.

```
// Funcao recursiva principal do jogo
function gameLoop() {
  applyGravity();
  pipeMovement();
  pipeSpawn();
  verifyEdges();
  // Chama a funcao gameLoop para o proximo frame
  requestAnimationFrame(gameLoop);
}
```

Listagem 2.76: Projeto exemplo em JS (Parte 6): Função de *loop* principal do jogo

Ainda, na Listagem 2.77, há a definição da função `generatePipes()`, responsável por gerar novos canos para o jogo. Essa função é chamada pela função `pipeSpawn()` para gerar novos canos durante o jogo. Dentro da função, são realizadas as seguintes ações:

- Linhas 4 e 5: Gera duas alturas aleatórias para os canos usando a função `Math.random()`.

- Linhas 8 a 14: Cria quatro elementos de cano usando a função `createPipeElement()`, passando os parâmetros necessários (classe, posição inicial, posição final e altura).
- Linhas 17 a 20: Define a propriedade `passed` como `false` para todos os canos gerados.
- Linhas 23 a 26: Adiciona os elementos de cano ao corpo do documento.
- Linha 27: Adiciona os elementos de cano ao *array* `pipes`.

```
1  // Funcao para gerar novos canos
2  function generatePipes() {
3      // Altura aleatoria dos canos
4      const pipeHeight = Math.random() *
5      (window.innerHeight - 400) + 50;
6
7      // Cria os elementos de cano superior e inferior
8      const pipeTop = createPipeElement('pipe', 0,
9      window.innerWidth, pipeHeight);
10     const pipeBottom = createPipeElement('pipe',
11     pipeHeight + 300, window.innerWidth,
12     window.innerHeight - pipeHeight - 300);
13
14     // Define a propriedade 'passed'
15     pipeTop.passed = false;
16     pipeBottom.passed = false;
17
18     // Adiciona os canos ao HTML e ao array de canos
19     document.body.appendChild(pipeTop);
20     document.body.appendChild(pipeBottom);
21     pipes.push(pipeTop, pipeBottom);
22 }
```

Listagem 2.77: Projeto exemplo em JS (Parte 7): Função de geração de canos

A função `createPipeElement()` é evidenciada na Listagem 2.78, que é uma função responsável por criar um elemento de cano para o jogo. Essa função é chamada pela função `generatePipes()` para criar os elementos de cano do jogo. Dentro da função, são realizadas as seguintes ações:

- Linha 3: Cria um novo elemento de `div`.
- Linha 4: Define a classe do elemento como a classe recebida como parâmetro.
- Linha 5 a 7: Define a posição inicial e a posição final do elemento usando os parâmetros `top`, `left` e `height`.
- Linha 8: Retorna o elemento criado.

```
// Funcao para criar um elemento de cano
function createPipeElement(className, top, left, height) {
  const pipeElement = document.createElement('div');
  pipeElement.className = className;
  pipeElement.style.top = `${top}px`;
  pipeElement.style.left = `${left}px`;
  pipeElement.style.height = `${height}px`;
  return pipeElement;
}
```

Listagem 2.78: Projeto exemplo em JS (Parte 8): Função de criação de canos

Na Listagem 2.79 há a definição da função `isColliding()`, responsável por verificar se houve colisão entre dois elementos do jogo. Essa função é chamada pela função `pipeMovement()` para verificar se houve colisão entre o pássaro e os canos. Dentro da função, são realizadas as seguintes ações:

- Verifica se a posição do elemento `element1` (pássaro) em relação ao lado esquerdo é menor do que a posição do elemento `element1` (cano) em relação ao lado direito.
- Verifica se a posição do elemento `element1` em relação ao lado direito é maior do que a posição do elemento `element2` em relação ao lado esquerdo.
- Verifica se a posição do elemento `element1` em relação ao topo é menor do que a posição do elemento `element1` em relação ao topo.
- Verifica se a posição do elemento `element1` em relação ao topo é maior do que a posição do elemento `element1` em relação ao topo.
- Se todas essas condições forem verdadeiras, significa que houve colisão entre os elementos.

```
// Funcao que verifica se dois elementos colidiram
function isColliding(element1, element2) {
  const rect1 = element1.getBoundingClientRect();
  const rect2 = element2.getBoundingClientRect();
  return (
    rect1.top < rect2.bottom &&
    rect1.bottom > rect2.top &&
    rect1.left < rect2.right &&
    rect1.right > rect2.left
  );
}
```

Listagem 2.79: Projeto exemplo em JS (Parte 9): Função de verificação de colisão

A definição da função `gameOver()` é apresentada na Listagem 2.80, responsável por encerrar o jogo quando ocorre uma colisão entre o pássaro e um cano. Essa função é chamada pela função `pipeMovement()` quando ocorre uma colisão entre o pássaro e um cano, o que encerra o jogo. Dentro da função, são realizadas as seguintes ações:

- Define a velocidade do jogo como zero para parar o movimento dos canos.
- Exibe uma mensagem de "Game Over" na página usando o método `innerHTML` do elemento `score`.
- Remove os eventos de clique (*mousedown*) e de pressionar tecla (*keydown*) do documento para impedir a interação do jogador após o fim do jogo.

```
// Funcao para encerrar o jogo
function gameOver() {
  alert(`Fim do jogo! Seu Score: ${scoreCount / 2}`);
  location.reload();
}
```

Listagem 2.80: Projeto exemplo em JS (Parte 10): Função de fim de jogo

A última função declarada no código é a função `jump()`, que é apresentada na Listagem 2.81. Essa função realiza uma verificação condicional que checa se o código da tecla pressionada é igual a 32 (que representa a barra de espaço) e se a variável `gravity` é maior ou igual a -2. Essa condição garante que o pássaro só pode saltar se a barra de espaço for pressionada e se a gravidade permitir o salto. Se a condição for verdadeira, o valor de `gravity` é incrementado pela variável `jumpHeight`. Isso significa que o pássaro ganhará uma força extra para subir no jogo, pois a gravidade será temporariamente reduzida ou anulada.

```
// Funcao para lidar com o salto do passaro
function jump(event) {
  // Verifica se a barra de espaco esta pressionada
  // e se a gravidade permite o salto
  if (event.keyCode === 32 && gravity >= -2) {
    gravity += jumpHeight;
  }
}
```

Listagem 2.81: Projeto exemplo em JS (Parte 11): Função de pulo

Por fim, como mostra a Listagem 2.82, para configurar o controle do jogo, em escopo global, um gatilho de evento *keydown* (tecla pressionada) é adicionado para chamar a função `jump`, fazendo com que, ao pressionar a tecla espaço (verificação realizada dentro da função `jump`, o pássaro pule. E logo em seguida, o jogo é inicializado pela função recursiva `gameLoop()`.

```
1 // salto ao pressionar enter
2 document.addEventListener('keydown', jump);
3 // Inicia o loop do jogo
4 gameLoop();
```

Listagem 2.82: Projeto exemplo em JS (Parte 12): Inicialização do jogo e evento de pulo ao pressionar enter

No código *TypeScript* (TS), há algumas diferenças em relação ao código *JavaScript* (JS). A Listagem 2.83 apresenta as mudanças em relação a alguns trechos de código já apresentados. Uma das principais diferenças é o uso de tipos estáticos e a declaração de interfaces para elementos do DOM. As principais diferenças entre os códigos *JavaScript* e *TypeScript* são:

- No início do código, a interface `PipeElement` é definida para estender a interface `HTMLDivElement` e adicionar a propriedade booleana `passed`, como está evidenciado na segunda a quarta linha do código.
- Definição do tipo da variável `pipes` como um *array* de elementos que estejam de acordo com a interface `PipeElement` criada, como pode ser observado na quinta linha.
- Dentro da função `generatePipe`, as variáveis dos canos gerados (`pipeTop` e `pipeBottom`) recebem elementos *HTML* convertidos para o molde definido pela interface `PipeElement`, com o atributo `passed` adicionado. Da sexta à décima primeira linha ilustram essa mudança.
- Os parâmetros da função `createPipelineElement` têm tipos definidos, sendo `ClassName` do tipo `string` e `top`, `left` e `height` do `number`, como pode ser visto na décima segunda linha.
- Os parâmetros da função `isColliding` têm tipos definidos como `HTMLElement`, como pode ser visto na décima terceira linha.
- Da mesma forma, a função `jump` tem seu parâmetro com tipo definido como `KeyboardEvent`, como está definido na décima quarta linha.

```
// Interface para o elemento do cano
interface PipeElement extends HTMLDivElement {
    passed: boolean;
}
const pipes: PipeElement[] = [];
function generatePipes() {
    ...
    const pipeTop = document.createElement('div') as
    PipeElement;
    const pipeBottom = document.createElement('div') as
    PipeElement;
    ...
}
function createPipeElement(className:string, top: number,
left: number, height: number) {...}
function isColliding(element1: HTMLElement, element2:
HTMLElement) {...}
function jump(event: KeyboardEvent) {...}
```

Listagem 2.83: Mudanças no código do projeto exemplo ao desenvolver em *TypeScript*

Além disso, uma mudança não apresentada na Listagem 2.83, mas que é importante mencionar é que onde aparecem as variáveis `bird` e `score` deve-se usar um símbolo de exclamação (!) posterior à variável, por exemplo: `bird!.offsetTop`. A exclamação após a variável indica asserção de que a variável não terá valor nulo. Essa mudança é necessária no código, pois o *TypeScript* verifica a possibilidade de a variável ter valor nulo e dá erro.

Ao desenvolver o projeto[9] exemplo proposto nesta seção é possível evidenciar algumas características do *TypeScript* que se sobressaem sobre o *JavaScript* pela perspectiva de desenvolvimento:

- As variáveis são inferidas dinamicamente em tempo de execução no *JavaScript*, permitindo que um valor de qualquer tipo seja atribuído a uma variável. Já no *TypeScript*, as variáveis podem ter seus tipos declarados explicitamente, o que traz mais segurança ao código e facilita a detecção de erros durante a fase de desenvolvimento.
- Embora *JavaScript* seja uma linguagem de POO, o *TypeScript* aprimora esse aspecto com recursos adicionais. *TypeScript* permite a definição de classes, interfaces, herança e polimorfismo de forma mais estruturada e explícita. Esses recursos facilitam a organização do código, tornando-o mais legível e mantível. Além disso, o TypeScript também suporta módulos, permitindo a criação de um código modular e reutilizável.

[9] O código desse projeto está disponível no GitHub em https://github.com/JadsonFaustino/flappy-box-game-js-ts.

- Enquanto o *JavaScript* é interpretado pelo navegador ou pelo ambiente de execução, o *TypeScript* requer um processo de compilação antes da execução. Além disso, o *TypeScript* possui um ecossistema robusto, com ferramentas avançadas de desenvolvimento, como autocompletar, refatoração de código, detecção de erros em tempo real e suporte a IDEs populares. Isso torna o desenvolvimento em *TypeScript* mais eficiente e produtivo.

Capítulo 3

Fundamentos da Internet

Victória Tomé Oliveira (UFC)

Sayonara Santos Araújo (UFC)

Wendley Souza da Silva (UFC)

Fischer Ferreira (Unifei)

3.1 Internet vs Web

É comum aos iniciantes no desenvolvimento web acharem que a Internet e a web são sinônimos. Mas, na verdade, o termo internet está relacionado a um conjunto de redes que conectam vários dispositivos com capacidade de acessar e se comunicar entre si. Porém, a web é um sistema de informação que reúne um conjunto de arquivos interligados. A web também pode ser vista em duas perspectivas: os sites web e as aplicações web. O web site é uma coleção de páginas HTML estáticas, isto é, que não interagem com um banco de dados através de uma linguagem de servidor web. A aplicação web pode conter uma coleção de páginas, porém o conteúdo destas páginas é montado dinamicamente, ou seja, é carregado através de solicitações (requisições) à um banco de dados, que conterá armazenados os textos e a indicação dos caminhos das imagens ou mídias que a página precisa exibir.

A internet é modularizada em conjuntos de camadas. Cada camada possui, de forma bem clara, as definições de cada serviço e os dados de entrada e de saída. Um modelo de camada é um modelo de referência TCP/IP. Este é constituído das camadas de aplicação, transporte, Internet e Host/Rede. A camada mais importante para o desenvolvedor é a camada de aplicação, nela estão as aplicação web, Nessa camada existe um protocolo chamado HTTP (HyperText Transfer Protocol). o HTTP é utilizado para transferência de dados via Internet, utilizando URL (Universal Resource Location) para localização de conteúdo na internet.

3.2 Camada de aplicação da Internet

3.2.1 Hypertext Transfer Protocol (HTTP)

O *Hypertext Transfer Protocol* (HTTP) é um protocolo da camada de aplicação do modelo TCP/IP. O HTTP é o principal protocolo usado pela *World Wide Web* para comunicação, sendo usado desde 1990.

O HTTP funciona como um protocolo de requisição-resposta operando na arquitetura cliente-servidor, a arquitetura cliente e a arquitetura servidor, operam em sistemas finais diferentes, porém, conversam entre si por meio da troca de mensagens HTTP. O cliente envia uma solicitação ao servidor e o servidor responde a essa solicitação. O HTTP define a estrutura dessas mensagens e como o cliente e o servidor trocam as mensagens. A solicitação geralmente inclui um método (ex.: GET, POST, PUT ou DELETE) e um URL que identifica o recurso que o cliente deseja acessar. O servidor, por sua vez, envia uma resposta que pode incluir um código de status, cabeçalhos e o corpo da resposta, que pode ser HTML, texto, imagens, entre outros tipos de dados (KUROSE; ROSS, 2010; KYAW, 2019). A ideia geral da interação entre cliente e servidor é ilustrada na Figura 3.1.

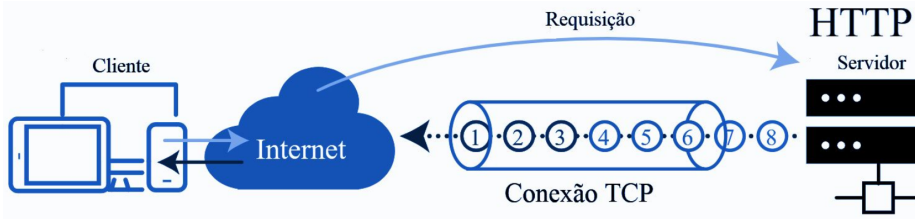


Figura 3.1: Protocolo HTTP.

200 OK
Date:
Server:
Content-Type:

Versões do HTTP

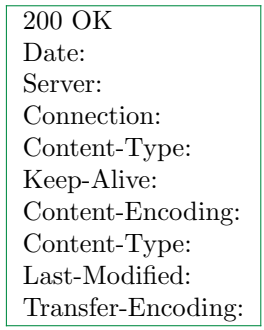
O *HyperText Transfer Protocol* (HTTP), foi lançado em 1991 e originalmente não tinha um número de versão associado a ele. Mais tarde, foi referido como versão 0.9, ou seja, HTTP/0.9 para distingui-lo de revisões futuras.

O HTTP/0.9, versão 0.9, é um protocolo simples para a transferência de dados no formato de texto ASCII pela Internet, o único método de solicitação HTTP disponível é o HTTP GET, que incluí apenas um caminho para o recurso. Ao contrário das versões futuras do protocolo, a URL completa não foi incluída porque na versão do protocolo, o servidor e a porta eram redundantes e, portanto, desnecessários após a conexão. No HTTP/0.9 não há suporte para campos de cabeçalhos nas solicitações. A resposta HTTP incluí apenas o recurso solicitado, que é um arquivo *Hypertext Markup Language* (HTML), não há campos de cabeçalhos na resposta e nem código de status para indicar o sucesso, erro e etc., se um erro ocorrer, o servidor gera um arquivo HTML específico da situação que o cliente apresenta ao usuário para descrever o problema (NEWMARCH; PETTY, 2022).

Depois da versão original HTTP/0.9, a versão HTTP/1.0 foi desenvolvida entre 1992 e 1996. A nova versão do HTTP foi desenvolvida para suprir a necessidade de transferir não apenas texto, o protocolo se tornou mais flexível suportando diferentes tipos de arquivos, o cabeçalho *HTTP Content-Type* foi criado para identificar o conteúdo do corpo da mensagem (NEWMARCH; PETTY, 2022).

O HTTP/1.0 têm melhorias significativas, a primeira melhoria foi adicionar a especificação/versão de protocolo, essa informação foi adicionada na mesma linha do método de solicitação HTTP. Essa melhoria foi necessária para informar ao servidor qual versão do protocolo será usada para a transmissão, a escolha era entre HTTP/0.9 e HTTP/1.0 (BERNERS-LEE et al., 1996).

Os métodos HTTPs disponíveis foram expandidos além do HTTP GET foi incluído o HEAD e POST. O POST foi introduzido para enviar uma mensagem do



servidor para o cliente, enquanto anteriormente a comunicação era essencialmente unidirecional. Outra adição foi uma linha de status para indicar os resultados de cada solicitação HTTP. No HTTP/0.9 o servidor envia um arquivo HTML baseado na situação para o cliente para exibir o resultado de um erro, no HTTP/1.0 os códigos de status HTTP são enviados para indicar sucesso e falha, deixando mais responsabilidade para o navegador. Uma quarta adição foi incluir um cabeçalho HTTP, que permite uma mensagem mais rica e flexível, independentemente de ser uma solicitação ou uma resposta HTTP. Com a adição de cabeçalhos uma nova técnica foi implementada, a cache. A Cache HTTP é uma técnica usada para melhorar o desempenho de um site, podendo ser implementado pelo cliente, servidor, um nó intermediário como um *proxy* ou qualquer combinação destes. Os cabeçalhos HTTP também foram usados para incorporar metainformações, como informações sobre o servidor e o usuário-cliente, bem como a própria mensagem (BERNERS-LEE et al., 1996; FIELDING et al., 2014b). Para reduzir o número de conexões HTTP necessárias, o tipo de conexão HTTP persistente foi introduzido no HTTP/1.0. Reduzindo a sobrecarga, sendo assim, várias solicitações HTTP podem ser feitas em uma única conexão HTTP, sem incorrer na largura de banda adicional necessária para descartar e restabelecer novas.

Depois que a versão HTTP/0.9 foi lançada e a funcionalidade foi posteriormente estendida para HTTP/1.0, sua popularidade e uso cresceram, mas houve diferentes implementações e falta de padronização. Em 1997, quando HTTP/1.1 foi publicado, várias melhorias foram introduzidas. Isso trouxe novos recursos, bem como o esclarecimento de ambiguidades no protocolo, pois muitos aplicativos se autodenominavam HTTP/1.0, mas não implementavam o protocolo totalmente. Apesar do HTTP/1.1 ter sido publicado apenas alguns meses após o HTTP/1.0, muitas funcionalidades foram adicionadas. Foi implementado um controle de cache adicional, negociação de conteúdo que permite que o cliente e o servidor escolhem uma das várias representações do recurso de destino existentes no servidor em diferentes idiomas, codificações de conteúdo e tipos. Também foram implementados novos métodos de solicitações HTTP e suporte para muitos novos cabeçalhos HTTP (FIELDING et al., 1999).

Uma melhoria significativa é a conexão HTTP, No HTTP/1.1, o TCP é usado para fornecer conexões HTTP de curta duração, conexões HTTP persistentes e *pipelining*. A conexão HTTP persistente também é conhecida como conexão *keep-alive* e é o método padrão em HTTP/1.1. Uma vantagem dessa nova versão é o desempenho, pois elimina vários *handshakes*. Especificamente, quanto mais tempo uma conexão HTTP permanecer aberta, mais tempo o algoritmo de controle de congestionamento TCP terá para encontrar a taxa de transferência ideal. Isso ocorre porque cada conexão TCP começa em um modo fortemente limitado de taxa para atenuar o impacto de pacotes de dados perdidos. Se a conexão HTTP suportar mais largura de banda, ela será ajustada gradualmente. As conexões HTTP persistentes consomem recursos mesmo quando estão ociosas e, como resultado, não são mantidas indefinidamente. Uma conexão HTTP inativa será descartada para conservar os recursos do servidor e aumentar o desempenho (FIELDING et al., 1999).

O *pipelining* HTTP é semelhante a uma conexão HTTP persistente em que várias solicitações HTTP podem ser feitas antes de ser fechada. O que distingue esses dois tipos de conexão HTTP é que uma conexão HTTP persistente se destina a ser um conjunto de pares completos de solicitação/resposta HTTP. À medida que cada solicitação HTTP é feita, o servidor gera e envia a resposta HTTP correspondente. Um segue sucessivamente o outro. Com o *pipelining*, várias solicitações HTTP podem ser feitas sem a necessidade de esperar por uma resposta HTTP, o que acaba reduzindo a latência da rede. O *Pipelining* é limitado a métodos HTTP idempotentes, incluindo HTTP GET , HEAD , PUT e DELETE . A razão para isso é que, se a mensagem for perdida, toda a solicitação HTTP poderá ser reenviada sem a preocupação com efeitos colaterais (FIELDING et al., 1999).

O HTTP/1.1 foi difundido e adotado por inúmeros navegadores e usuários finais, devido a isso, surgiu a necessidade de lidar com a flexibilidade de novos serviços, então, foram feitas revisões a versão 1.1 do protocolo, o que levou ao lançamento de um novo padrão em junho de 1999, como RFC2616. Em 2007, o HTTP *Working Group*, lançou uma especificação de seis partes que substituiu a RFC2616 (FIELDING; RESCHKE, 2014c; FIELDING; RESCHKE, 2014d; FIELDING; RESCHKE, 2014b; FIELDING et al., 2014a; FIELDING et al., 2014b; FIELDING; RESCHKE, 2014a). O HTTP/1.1 iniciou uma padronização, novos recursos e aprimorou a eficiência do protocolo por meio da cache, da codificação, da reutilização de conexões HTTP e da *pipeline* de solicitações HTTP.

Em 2015, HTTP/2 foi publicado, representando uma revisão importante do protocolo e foi desenvolvido pelo *HTTP Working Group of the Internet Engineering Task Force* (IETF). A maior motivação por trás da publicação do HTTP/2 foi a necessidade por maior desempenho, isto foi necessário devido ao avanço no desenvolvimento de sites. Os sites estavam sendo desenvolvidos com uma quantidade massiva de mídias e ofereciam mais interação com o cliente. As operações do lado do servidor e os *scripts* do lado do cliente estavam tornando-se maiores e mais complexos, sendo assim, exigiam mais recursos, incluindo largura de banda. A

partir disto, a padronização do protocolo foi apoiada por vários navegadores de internet populares, incluindo Google Chrome, Internet Explorer 11, Mozilla Firefox entre outros. O HTTP/2 foi rapidamente adotado, pois não exigia alterações em aplicativos ou sites do lado do servidor, também não houve perda de informações ou modificação dos cabeçalhos HTTP entre HTTP/1.1 e HTTP/2. A conservação da largura de banda levou a poupanças de custos, em particular para websites com tráfego elevado.

O HTTP/2 é compatível com versões anteriores, as principais mudanças implementadas foram otimizações e correções de *bugs*. O HTTP/2 suporta os recursos do HTTP/1.1, entretanto, é otimizado para transporte HTTP. O HTTP/2, difere-se na forma como os dados representados são formatados e como são transmitidos. Todos os métodos HTTP, códigos de status HTTP e campos de cabeçalho HTTP de versões anteriores são suportados. Um dos objetivos do desenvolvimento do HTTP/2 era ser capaz de traduzir entre HTTP/1.1 e HTTP/2, em qualquer direção, sem perda de informações.

O HTTP/2 distingue-se das suas versões anteriores, mas é um protocolo binário, ou seja, não pode ser escrito ou lido manualmente. A versão 2, introduz compactação de campo de cabeçalho HTTP e suporta múltiplas trocas de dados simultâneas em uma única conexão HTTP, essa atualização redefina uma mensagem, o que permite que o cliente pare no meio da ação e comece novamente, sem precisar descartar e redefinir a conexão HTTP. O controle de fluxo e a priorização concedem fluxos de dados multiplexados e suportam o envio não solicitado de respostas HTTP do servidor para o cliente. A multiplexação permite que várias solicitações HTTP estejam em andamento ao mesmo tempo. A multiplexação veio para resolver o problema de bloqueio da rede, no HTTP/1.1 foi introduzido o conceito de *pipeline* para solucionar esse problema, porém, o HTTP/1.1 não preocupo-se com mensagens maiores e mais lentas, essas mensagens ainda podem bloquear as subsequentes. Além disso, do ponto de vista prático e de implementação, o *pipeline* não era confiável porque intermediários, como servidores proxy, nem sempre lidam com essas solicitações HTTP de maneira adequada. Na multiplexação, o HTTP/2 não faz uso de múltiplas conexões HTTP, em vez disso, todas as solicitações HTTP são feitas simultaneamente utilizando uma única conexão HTTP.

Em Bangkok em 2018, a Internet Engineering Task Force (IETF) reuniu-se e fez uma revisão do HTTP e um novo Internet Draft foi adotado, o protocolo de transporte QUIC, sucessor do HTTP/2, foi renomeado para HTTP/3. Esta grande revisão do padrão altera o protocolo de rede da camada de transporte subjacente do *Transmission Control Protocol* (TCP) para QUIC (*Quick UDP Internet Connections*). A proposta original foi chamada de "Semântica HTTP/2 usando o protocolo de transporte QUIC", depois foi renomeada para "Protocolo de transferência de hipertexto (HTTP) sobre QUIC" e, finalmente, HTTP/3.

Atualmente a especificação para HTTP/3 ainda é um rascunho da IETF. No entanto, ele já é compatível com a maioria dos navegadores da web, sendo utilizado por empresas importantes como o Google e o Facebook e está sendo executado

em milhões de sites. Se você está usando o Chrome e se conectando a um serviço Google, provavelmente já está utilizando o HTTP/3 (LANGLEY et al., 2017).

HTTP/3

O protocolo QUIC foi desenvolvido pelo Google, é um algoritmo de comunicação multiplexado construído sobre o protocolo UDP. Inicialmente, o objetivo do QUIC era oferecer uma alternativa ao HTTP/2 utilizando o protocolo TCP para transmissão de dados e TLS para segurança. O problema é que o protocolo TCP não foi originalmente construído com foco na eficiência. Por exemplo, iniciar uma conexão HTTP requer uma viagem completa de rede para garantir que o cliente e o servidor estejam disponíveis e preparados para trocar dados. Dependendo da distância entre cliente e servidor, bem como do número de intermediários, a latência e os tempos de transmissão resultantes tornam-se mais perceptíveis (IYENGAR; THOMSON, 2018).

Além disso, o bloqueio inicial é outro problema muito comum, que deveria ter sido corrigido no HTTP/2 com a implementação da abordagem pipeline. O pipeline permite múltiplas solicitações HTTP utilizando a mesma conexão TCP, entretanto, essa técnica é inadequada devido à forma como o TCP transmite dados. O protocolo TCP entende e trabalha com dados como um fluxo de byte único, mesmo nos casos em que vários arquivos mutuamente exclusivos são enviados simultaneamente. O problema acontece quando pacotes contendo dados "no início da linha" são perdidos, consequentemente, os pacotes restantes são consequentemente atrasados até que os pacotes perdidos sejam encontrados. Na Figura 3.2, observa-se a comparação entre múltiplas solicitações HTTP/2 e QUIC (LANGLEY et al., 2017).

O protocolo HTTP/3 depende fundamentalmente do QUIC para segurança e integridade de dados, autenticação de pares e entrega de dados confiável e em ordem com desempenho aprimorado. Essas são melhorias no versionamento do HTTP que não são suportadas pelo TCP. A partir disto, QUIC é construído sobre o UDP , ele não é diretamente afetado pelas limitações do TCP (LANGLEY et al., 2017).

Em transmissão de protocolo, o desempenho mede a velocidade de transmissão, que é uma combinação de largura de banda e latência. A largura de banda refere-se ao volume de dados que podem ser enviados de uma vez, enquanto a latência se refere ao tempo gasto na movimentação de dados de um *endpoint* para outro. A latência bidirecional é o tempo necessário para concluir uma viagem completa de ida e volta da rede. O desempenho ideal é alcançado por um conjunto confiável e consistente de conexões de alta largura de banda e baixa latência durante toda a viagem de ida e volta da rede. O desempenho é uma medida crítica porque está diretamente relacionado à perda de pacotes, que por sua vez requer uma abordagem confiável para controle de congestionamento (LANGLEY et al., 2017).

QUIC usa *Headers* compactados , assim como HTTP/2 . No entanto, QUIC utiliza o algoritmo QPACK para compactação de cabeçalho, enquanto HTTP/2 usa HPACK. Comparando o QUIC e o HTTP/2 em relação a carga da CPU, o QUIC leva mais tempo de CPU do que o mesmo volume de dados usando HTTP/2

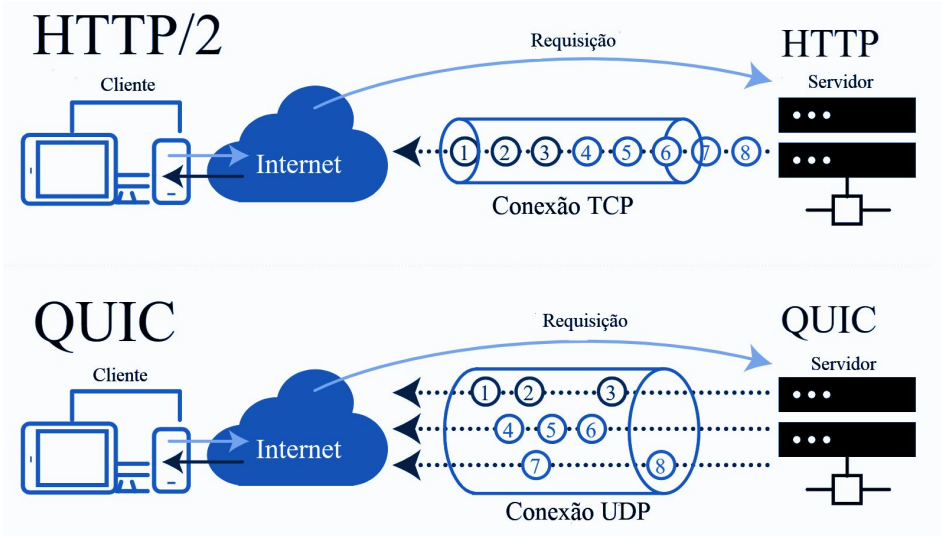


Figura 3.2: Comparação entre múltiplas solicitações HTTP/2 x QUIC

protegido com TLS. Isto acontece devido o UDP não ser tão otimizado quanto o TCP, do ponto de vista de implementação. No entanto, isso é algo que mudará com o tempo, à medida que o QUIC se tornar mais difundido (LANGLEY et al., 2017) .

O TCP e o QUIC utilizam métodos análogos para controlar o congestionamento e limitar a probabilidade e os efeitos da perda de pacotes durante a transmissão. Sendo assim, cada um começa enviando dados a uma taxa relativamente lenta e depois aumenta a velocidade à medida que são recebidas confirmações sobre quais e quantos dados estão sendo recebidos pelo terminal. Esta é uma área de desenvolvimento ativo e onde o QUIC tem uma clara vantagem. Ao reduzir o número de pacotes de confirmação em determinados ambientes, grandes benefícios podem ser obtidos em termos de velocidade. Esta extensão, e outras semelhantes, são relativamente fáceis de implementar em QUIC, mas são consideravelmente mais difíceis de implementar em TCP (LANGLEY et al., 2017).

O QUIC não requer o mesmo número de viagens de ida e volta de rede que o HTTP/2 usando TCP e TLS, reduzindo a sobrecarga para cada conexão HTTP. Isso ocorre porque o TCP não permite o envio de dados não-TCP durante o handshake inicial, portanto a conexão TLS precisa de uma viagem de ida e volta de rede separada para proteger a conexão TCP. Como o QUIC foi projetado tendo em mente essas preocupações de segurança, as duas operações são realizadas em conjunto. Na Figura 3.3, observa-se a comparação entre o númro de viagens de ida e volta do HTTP/2 e do HTTP/3.

Quando uma conexão HTTP é estabelecida e segura, é razoável assumir, sob certas circunstâncias, que o cliente se conectará novamente em pouco tempo. O

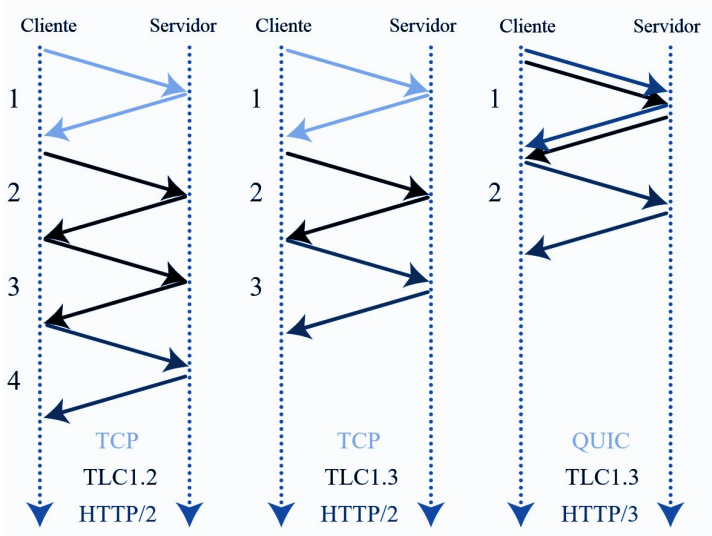


Figura 3.3: Comparação entre números de viagens de ida e volta.

algoritmo QUIC aproveita esse conhecimento preparando-se para uma segunda conexão HTTP antes que a primeira seja encerrada. Ao fazer essa preparação com antecedência, você economiza pelo menos uma viagem completa de ida e volta da rede ao estabelecer a nova conexão HTTP. Isso é conhecido como retomada de tempo de ida e volta zero ou 0-RTT (LANGLEY et al., 2017).

O QUIC possui certos limites ao responder a solicitações 0-RTT para impedir certos tipos de ataques DDoS . Essa limitação de dados é conhecida como limite antiamplificação e refere-se à quantidade de dados que pode enviar em resposta antes que o cliente seja verificado. O limite é definido para três vezes a quantidade de dados recebidos desse endereço (LANGLEY et al., 2017).

Outra melhoria de desempenho suportada pelo protocolo QUIC é a migração de conexão, que pode manter uma conexão HTTP ao transferir de uma rede para outra. Para identificar conexões HTTP, o QUIC introduz um novo parâmetro chamado identificador de conexão (CID) que não muda durante a transferência entre redes. O CID, neste contexto, não é um identificador único. Em vez disso, é uma lista de identificadores gerados aleatoriamente, conhecidos pelo servidor e pelo cliente, que identificam a mesma conexão HTTP . Cada vez que uma nova rede é utilizada, o CID muda para outra da lista. O objetivo disso é proteger ainda mais a privacidade do originador (LANGLEY et al., 2017).

Embora tenha havido muita atenção à melhoria do desempenho com a adoção do QUIC, há também uma vantagem do ponto de vista da segurança e da privacidade dos dados. É útil considerar que uma conexão HTTP segura (ou seja, HTTPS) é feita usando o protocolo *Transport Layer Security* (TLS) sobre a conexão HTTP.

Método	Definição
GET	O método GET solicita dados de um recurso especifico.
HEAD	O método HEAD é quase idêntico a GET, mas sem o corpo da resposta.
OPTIONS	O método OPTIONS descreve as opções de comunicação para o recurso de destino.
TRACE	O método TRACE é usado para executar um teste de loopback de mensagem que testa o caminho para o recurso de destino.
DELETE	O método DELETE exclui o recurso especificado.
PUT	O método PUT é usado para enviar dados a um servidor para criar/atualizar um recurso.
POST	O método POST é utilizado para enviar dados a um servidor para criar/atualizar um recurso.
PATCH	O método PATCH é usado para aplicar modificações parciais a um recurso.
CONNECT	O método CONNECT é usado para iniciar uma comunicação bidirecional (um túnel) com o recurso solicitado.

Tabela 3.1: Métodos e Definição HTTP.

Essa abordagem de duas camadas está em vigor desde os primórdios da Internet, quando havia menos prioridade quando se tratava de criptografar o tráfego. Com o QUIC, a segurança foi mesclada e o TLS 1.3 faz parte do protocolo. Não é mais uma abordagem multicamadas. Uma das únicas partes de uma conexão QUIC que não é criptografada é o CID, que ajuda a direcionar o tráfego. Outra questão relacionada à segurança é que o QUIC só funciona com certificados TLS oficiais (JOARDER; FUNG, 2022).

Métodos HTTP

As especificações HTTP inclui uma coleção de métodos usados para interagir com recursos do lado do servidor. Esses métodos destinam-se a cobrir todos os tipos possíveis de interações com recursos. Os métodos HTTP determinam como o servidor deve se comportar com a URL fornecida no momento da requisição de um determinado recurso. (GOURLEY; TOTTY, 2002)

Códigos de status HTTP

Os códigos de status HTTP são utilizados para indicar uma resposta de sucesso, falha e outras propriedades sobre o resultado de uma solicitação HTTP. O primeiro dígito de um código de status indica a categoria, que geralmente indica se a solicitação foi bem-sucedida ou falhou. Um cliente agirá de acordo com o código de status da resposta, independentemente do conteúdo do corpo de uma mensagem

de resposta HTTP. Cada solicitação de resposta HTTP é seguida por um código de status. Cada um dos códigos de status de resposta é usado para transmitir informações gerais sobre o resultado da solicitação. Atualmente, existem cinco categorias principais de códigos de resposta HTTP, cada uma identificável pelo primeiro dígito, sendo elas,

1. 1xx: Informativo - Informa ao cliente que a solicitação continua;
2. 2xx: Sucesso - Informa que uma resposta bem-sucedida foi recebida, interpretada, corrigida e aceita;
3. 3xx: Redirecionamento - Indica que ações adicionais precisam ser tomadas antes que a solicitação seja concluída;
4. 4xx: Erro do cliente - Indica que a solicitação não pode ser concluída devido a um problema com o cliente ou à sintaxe da solicitação;
5. 5xx: Erro no servidor - Indica que a solicitação é válida, mas não pode ser concluída devido a um problema do lado do servidor, como falta de recursos disponíveis.

Além das cinco categorias principais de códigos de status HTTP mencionadas acima, códigos de status adicionais também podem ser encontrados na *World Wide Web*. A seguir, segue a Tabela contendo todos os códigos de status HTTP.

3.3 Computação em Nuvem

A Computação em Nuvem permite o acesso, sob demanda, a recursos computacionais configuráveis como máquinas virtuais, serviços de armazenamento e aplicativos (MELL et al., 2011). Com esse modelo, o usuário tem a disposição serviços de Nuvem independente da sua localização e de forma facilitada, bastando ter acesso a uma rede e uma ferramenta que permita a comunicação com a Nuvem, como um navegador *web* (SOUSA et al., 2009).

Atualmente os provedores de Computação em Nuvem possuem uma quantidade vasta de servidores e uma diversidade de serviços, o que proporciona ao usuário recursos virtualmente ilimitados e aplicáveis a inúmeros cenários. Além desses fatores, as principais características do modelo são (MELL et al., 2011):

- **Autoatendimento sob demanda:** o usuário demanda recurso de acordo com sua necessidade e pode configurá-lo de forma independente, sem interação humana com o provedor. Em pouco tempo, por exemplo, é possível que o usuário provisione uma aplicação por um período e depois a remova.
- **Amplo acesso à rede:** os serviços de Nuvem estão disponíveis na rede e são acessados através de ferramentas heterogêneas dos clientes, como celulares, *tablets* e *laptops*. Ademais os serviços estão disponíveis independente do horário ou da localização do cliente.

1xx: Informational	415 Unsupported Media Type	520 Web Server Is Returning an Unknown Error
100 Continue	416 Range Not Satisfiable	521 Web Server Is Down
101 Switching Protocols	417 Expectation Failed	522 Connection Timed Out
102 Processing	418 I'm a Teapot	523 Origin Is Unreachable
103 Early Hints	419 Page Expired	524 A Timeout Occurred
2xx: Success	420 Method Failure or Enhance Your Calm	525 SSL Handshake Failed
200 OK	421 Misdirected Request	526 Invalid SSL Certificate
201 Created	422 Unprocessable Entity	527 Railgun Listener to Origin
202 Accepted	423 Locked	529 The Service Is Overloaded
203 Non-Authoritative Information	424 Failed Dependency	530 Site Frozen
204 No Content	425 Too Early	561 Unauthorized
205 Reset Content	426 Upgrade Required	598 Network Read Timeout Error
206 Partial Content	428 Precondition Required	599 Network Connect Timeout Error
207 Multi-Status	429 Too Many Requests	**Additional status codes**
208 Already Reported	430 HTTP Status Code	110 Response Is Stale
218 This Is Fine	431 Request Header Fields Too Large	111 Revalidation Failed
226 IM Used	440 Login Time-Out	112 Disconnected Operation
3xx: Redirection	444 No Response	113 Heuristic Expiration
300 Multiple Choices	449 Retry With	199 Miscellaneous Warning
301 Moved Permanently	450 Blocked by Windows Parental Controls	214 Transformation Applied
302 Found	451 Unavailable For Legal Reasons	299 Miscellaneous Persistent Warning
303 See Other	460 Client Closed Connection Prematurely	999 Unauthorized
304 Not Modified	463 Too Many Forwarded IP Addresses	
305 Use Proxy	464 Incompatible Protocol	
306 Switch Proxy	494 Request Header Too Large	
307 Temporary Redirect	495 SSL Certificate Error	
308 Permanent Redirect	496 SSL Certificate Required	
4xx: Client error	497 HTTP Request Sent to HTTPS Port	
400 Bad Request	498 Invalid Token	
401 Unauthorized	499 Token Required or Client Closed Request	
402 Payment Required	**5xx: Server error**	
403 Forbidden	500 Internal Server Error	
404 Not Found	501 Not Implemented	
405 Method Not Allowed	502 Bad Gateway	
406 Not Acceptable	503 Service Unavailable	
407 Proxy Authentication Required	504 Gateway Timeout	
408 Request Timeout	505 HTTP Version Not Supported	
409 Conflict	506 Variant Also Negotiates	
410 Gone	507 Insufficient Storage	
411 Length Required	508 Loop Detected	
412 Precondition Failed	509 Bandwidth Limit Exceeded	
413 Payload Too Large	510 Not Extended	
414 URI Too Long	511 Network Authentication Required	

Tabela 3.2: Códigos de Status HTTP.

- **Agrupamento de recursos:** os recursos do provedor são agrupados para atender vários consumidores com uma infraestrutura de multilocação, como diferentes recursos físicos e virtuais atribuídos dinamicamente aos consumidores. Dessa maneira, uma mesma infraestrutura é compartilhada com um certo nível de isolamento que permite a segurança e privacidades dos consumidores, além do uso econômico dessa infraestrutura.

- **Escalabilidade e elasticidade:** os recursos podem ser dimensionados facilmente, inclusive de forma automatizada. Assim é possível, por exemplo, aumentar ou diminuir o número de instâncias de uma aplicação de forma proporcional à quantidade de requisições recebidas por ela.

- **Serviço medido:** os provedores possuem sistemas de medição que permitem o controle e a otimização do uso de recurso. Com essas funcionalidades, o consumo de serviço pode ser monitorado e controlado para cada usuário, proporcionando uma visão mais transparente para o consumidor.

3.3.1 Modelos de implantação

Para implementar Computação em Nuvem em uma organização, além entender as necessidades desta e a viabilidade financeira, é preciso conhecer os modelos de implantação. Eles diferem entre si principalmente pela restrição de compartilhamento da infraestrutura e a responsabilidade pelo seu gerenciamento. Os modelos são (BELLO et al., 2021):

- **Nuvem privada:** os recursos são de uso exclusivo de uma única organização. A rede desse modelo é privada, e os *softwares* e *hardwares* são dedicados à organização. A Nuvem privada é interessante para organizações que presam sobretudo pela confidencialidade das informações. Geralmente ela é usada por órgãos governamentais, instituições financeiras e empresas com operações críticas (MICROSOFT, 2023a).

- **Nuvem pública:** os serviços são fornecidos por uma empresa provedora, responsável por manter e gerenciar a pilha de instalações base da infraestrutura. Os consumidores, pessoas físicas ou empresas, compartilham essa mesma infraestrutura e acessam os serviços via Internet. Geralmente esse é o modelo ideal para pequenas empresa. Exemplos de provedores de Nuvem pública são Amazon Web Services, Microsoft Azure e Gloogle Cloud.

- **Nuvem comunitária:** a infraestrutura é compartilhada por várias organizações com objetivos em comuns. Os serviços desse modelo são acessíveis apenas por essas organizações participantes. A Nuvem comunitária se assemelha a uma Nuvem privada utilizada por mais de uma empresa.

- **Nuvem híbrida:** consiste na combinação dos modelos de implantação citados anteriormente. Como exemplo dessa combinação, pode-se ter os sistemas

Modelo	Consumidores	Fornecedores
Privado	Apenas uma organização	A organização consumidora ou empresa terceira
Público	Público em geral (qualquer pessoa ou empresa)	Provedores de Nuvem pública
Comunitário	Um conjunto específico de organizações	O conjunto das organizações consumidoras ou empresa terceira
Híbrido	Irá depender da combinação dos modelos	Irá depender da combinação dos modelos

Tabela 3.3: Comparação entre os modelos de implantação.

financeiros de uma empresa na Nuvem privada, enquanto os serviços com dados não confidenciais ficam hospedados na pública.

A Tabela 3.3 apresenta um resumo do tipo de modelo de implantação, com os consumidores dos recursos e os responsáveis por gerenciar e fornecer os serviços de Nuvem. A escolha da implantação irá depender do negócio da organização, das restrições de segurança, do planejamento financeiro, entre outros fatores.

3.3.2 Modelos de serviço

Os serviços de Nuvem são disponibilizados aos usuários com certos níveis de abstração, e essa disponibilização se divide em modelos de serviços. Existem três principais modelos de serviços de Computação em Nuvem (RASHID; CHATURVEDI, 2019):

- **Infraestrutura como Serviço:** seu termo em inglês é *Infrastructure as a Service* (IaaS). Ele consiste em fornecer ao usuário um conjunto de serviços de infraestrutura virtualizados como disco, máquina, rede. Nesse modelo, o usuário não tem o trabalho de manter a parte física da infraestrutura ou o sistema de virtualização dos recursos. Exemplos desse tipos de serviços são o Amazon Elastic Cloud Computing (EC2) e o Google Compute Engine.

- **Plataforma como Serviço:** é também conhecido pela sigla PaaS – *Platform as a Service*. Nesse caso, o fornecedor disponibiliza ao usuário recursos para o *design*, desenvolvimento e hospedagem de aplicações. Então ficam sob responsabilidade do fornecedor o sistema operacional e a pilha de ferramentas que auxiliam no desenvolvimento e hospedagem. AWS Elastic Beanstalk, Heroku e Google App Engine são exemplos de PaaS.

- **Software como Serviço:** sua sigla é SaaS – *Software as a Service*. O modelo consistem na disponibilização de aplicações que são acessíveis através

de uma interface *web*. O fornecedor é responsável por manter a aplicações e todas as camadas abaixo dela. Normalmente ferramentas do tipo SaaS são destinada a um usuário final e podem ser acessadas por diferentes dispositivos, como *smartphones* e *laptops*. Google Docs, Dropbox e Red Hat Insights são exemplos de SaaS.

Ao longo dos anos, foram atribuídos, aos modelos de serviço, novas tecnologias e maior granularidade dos produtos oferecidos pelos provedores. Com isso, surgiram modelos como:

- **Contêiner como Serviço:** o serviço do tipo CaaS – *Computing as a Service* – permite ao usuário desenvolver, implantar e gerenciar aplicações com virtualização em contêiner. Esse modelo pode ser visto como uma extensão do IaaS com o provisionamento em contêiner (CLOUD, 2023b). Entretanto, diferente do IaaS, a manutenção do sistema sistema operacional e das camadas abaixo é feia pelo provedor de Nuvem. Como exemplo de CaaS, têm-se o Google Kubernetes Engine (GKE) e Azure Kubernetes Service (AKS).
- **Função como Serviço:** também conhecido como FaaS – *Function as a Service* –, é um tipo de computação *serveless* orientada a evento, na qual o usuário não precisa se preocupar em configurar e gerenciar o servidor que executa as aplicações. Nesse modelo, o desenvolvedor irá tratar a lógica de negócio como uma função que será implantada na Nuvem, preocupando-se apenas em escreve o código e gerenciar os pacotes da função. IBM Cloud Functions, AWS Lambda e OpenFaaS são exemplos de FaaS (HAT, 2020).

Todos esses modelos permitem definir o quê será cobrado no pagamento do serviço e o nível de gerenciamento feito pelo usuário e pelo provedor. A Figura 3.4 mostra uma comparação entre o sistema *on-premise* e os modelos de serviço de Computação em Nuvem, na qual as camadas em azul são gerenciadas pelo provedor e as em laranja, pelo usuário. O sistema *on-premise* representa o uso da infraestrutura tradicional, sem a contribuição da Nuvem, com todos os componentes de computação instalados localmente, sendo de total responsabilidade do usuário. Em contrapartida, IaaS, CaaS, PaaS, FaaS e SaaS transferem a responsabilidade de certos componentes ao provedor de Nuvem.

3.3.3 Provedores de serviço de Nuvem

Provedores de Nuvem são empresas terceiras que podem gerenciar a infraestrutura de uma Nuvem privada ou oferecer recursos de Nuvem pública sob demanda. Atualmente existem diversas empresas nesse ramo com um catálogo amplo de serviços. Segundo o relatório do Gartner de 2022 (GARTNER, 2022), os provedores de Nuvem líder no mercado são Amazon Web Services (AWS), Google e Microsoft, empresas que possuem um grande volume de clientes e podem atender uma quantidade extensa de casos de uso.

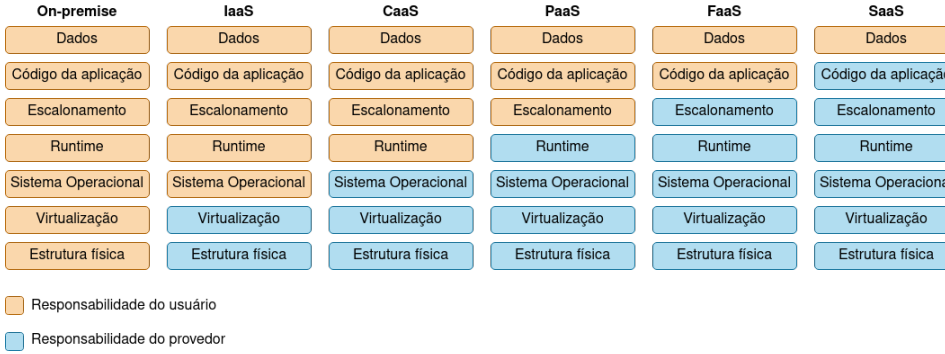


Figura 3.4: Comparação entre os serviços de Nuvem IaaS, CaaS, PaaS, FaaS e SaaS.

3.4 Virtualização de servidores

Duas tecnologias populares de virtualização servidores são as máquinas virtuais e os contêineres. Essas abordagens permitem a implantação de *softwares* remotos. A implantação é um processo que disponibiliza o *software* para uso e mantêm-o operacional (ARCANGELI et al., 2015). A implantação de uma aplicação ou servidor requisita *softwares* adicionais – como bibliotecas e pacotes – chamados de dependências, que devem ser instalados no sistema operacional (SERVICES, 2023). Contêineres e máquinas virtuais permitem a instalação de aplicações e suas dependência de forma isolada e escalável. Além disso, essas virtualizações fornecem mecanismos para executar vários e diferentes sistemas operacionais em um mesmo dispositivo físico (POTDAR et al., 2020). Com isso, é possível compartilhar recursos físicos para executar mais de uma aplicação de forma isolada. Ambas tecnologias são bastante presentes no mercado, principalmente em ambiente de Computação em Nuvem.

3.4.1 Máquina Virtual

Máquina virtual – *Virtual Machine* (VM) – é uma instância que pode realizar praticamente as mesmas funções de um computador real, como executar sistemas operacionais e aplicações. Ela também possui também características semelhantes, como memória, CPU, disco, interface de rede. Porém, diferentes dos computadores reais que possuem componentes físicos, tudo na VM é lógico, existe apenas como um código (MICROSOFT, 2023b).

As máquinas virtuais executam em dispositivos físicas através do *hipervisor*, um mecanismo que cria e mantém essas virtualizações. O *hipervisor* permite provisionar várias instâncias, de forma isolada, em um mesmo dispositivo, abstraindo os recursos físicos que um computador real possui (CLOUD, 2023a). Dessa maneira, é possível otimizar o uso dos recursos físicos computacionais.

Devido ao isolamento e a otimização do uso dos recursos, VM é aplicada em diversos casos de usos. Alguns exemplos do uso dessa virtualização são: implantação servidores, criação de ambientes de desenvolvimento e de teste, composição de mecanismo de recuperação de desastre e realizar tarefas com certo risco, como acesso a arquivos infectados.

3.4.2 Virtualização em contêiner

Contêiner é uma instância de virtualização a nível de sistema operacional (SO) que empacota uma aplicação e suas dependências. Nesse método de virtualização, o *kernel* – núcleo do SO – permite a execução e o isolamento de múltiplos contêineres em uma mesma máquina, chamada de hospedeira (BACHIEGA et al., 2018). Uma característica muito importante desse método é a imagem de contêiner, que é um artefato leve que possui tudo o que é necessário para executar uma aplicação, como código, bibliotecas e configurações do sistema (INC., 2023). Com esse artefato, é possível portar e executar, de forma ágil, uma aplicação em vários dispositivos diferentes.

A Figura 3.5 ilustra a diferença entre a virtualização em contêiner e a máquina virtual. Ambas abordagens possuem os *softwares* que realiza a virtualização, de um lado o *hypervisor* e do outro o mecanismo de contêiner (*container engine*). Outro diferença evidente é o fato do contêiner não precisar de um sistema operacional para cada instância, pois eles compartilham os recursos do SO hospedeiro, o que torna essa virtualização mais leve. Apesar dessa vantagem, o contêiner possui um gerenciamento limitado de recursos de *hardware* se comparado a VM.

Devido a essas diferenças, cada método pode ser interessante para cenários diferentes. Como a VM permite emular todo um computador, é possível simular um *smartphone*, sendo útil para o desenvolvimento de aplicativos móveis. Por outro lado, com sua leveza e portabilidade, o contêiner é relevante em arquitetura de microsserviços, na qual são executados diversos serviços de forma rápida e distribuída. No contexto de Nuvem, as máquinas virtuais são utilizadas em Infraestrutura com Serviço – *Infrastructure as a Service (IaaS)*, e os contêineres em Plataformas como Serviços – *Platform as a Service (PaaS)* (LI et al., 2017).

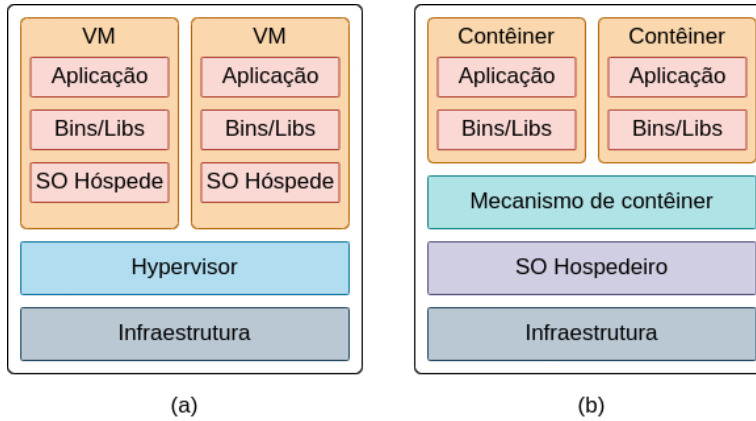


Figura 3.5: Comparação entre (a) máquina virtual e (b) contêiner.

Capítulo 4

Ruby on Rails, Laravel e Django

Vitoria Nobre Lima (UFC)

Carlos Elmano de Alencar e Silva (UFC)

Evilasio Costa Junior (UFC)

Fischer Ferreira (Unifei)

Uma etapa crucial durante o desenvolvimento de um projeto para web é a escolha de tecnologias a serem utilizadas que sejam adequadas às especificidades do domínio da aplicação, pois uma escolha errada pode acarretar na falha do projeto, assim devem ser considerados vários aspectos da tecnologia para que a aplicação cumpra os requisitos do projeto. Nessa perspectiva, o desenvolvedor web dispõe de diversas tecnologias e suas respectivas ferramentas, bibliotecas e *frameworks*, entre outros recursos possíveis, que se unem a uma grande variedade de tipos de domínios de sistemas e restrições de projeto, tornando a escolha não trivial.

Este capítulo apresenta os *frameworks* `Django`, `Laravel` e `Ruby on Rails` na seguinte ordem: a Seção 4.1 apresenta uma visão geral do `Django`, a Seção 4.2 apresenta a arquitetura do `Django`. As Seções 4.3 e 4.4 apresentam, respectivamente, algumas vantagens do uso do Django e um exemplo prático de uso. A Seção 4.5 apresenta uma visão geral do `Laravel`, a Seção 4.6 apresenta a arquitetura do `Laravel`, a Seção 4.7 apresenta um exemplo prático de uso do `Laravel`, a Seção 4.8 apresenta uma visão geral do `Ruby on Rails`, a Seção 4.9 apresenta a arquitetura do `Ruby on Rails`, a Seção 4.10 apresenta um exemplo prático de uso do `Ruby on Rails`.

4.1 Visão geral do Framework Django

O `Django`[1] é um *framework* web gratuito e de código aberto, desenvolvido em `Python`[2], que visa facilitar a criação de aplicações web, permitindo concluir projetos com maior rapidez e menos código. Criado em 2003 por uma equipe de desenvolvedores web, inicialmente como um sistema para gerenciar *sites* jornalísticos, o `Django` foi expandido após a equipe perceber a necessidade de reutilizar códigos e padrões de *design* comuns. Em 2005, esse processo culminou na criação de um *framework* de desenvolvimento web de código aberto (BENNETT, 2009), que rapidamente ganhou popularidade nos anos seguintes. Desde então, o `Django` tem sido constantemente aprimorado, incorporando novas funcionalidades, como o suporte a diversos tipos de bancos de dados e a introdução de novas classes, como as *Class-Based Views* (CBVs), que proporcionam uma maneira mais flexível e reutilizável de lidar com as *views*, e novas classes de formulários para facilitar a validação e o processamento de dados. Além disso, a forte atuação de uma comunidade de voluntários tem impulsionado o crescimento e a expansão do ecossistema de softwares relacionados ao `Django` (DJANGO, 2023).

Além disso, o `Django` se destaca por facilitar o desenvolvimento rápido de aplicações com um design limpo e pragmático, favorecendo uma abordagem ágil e eficiente. Ele adota o princípio DRY (*Don't Repeat Yourself*), esse princípio busca evitar a repetição de código, promovendo a reutilização e reduzindo redundâncias (MACIEL, 2020). Além disso, o `Django` é altamente escalável, pois combina velocidade e flexibilidade para lidar com grandes volumes de tráfego. Sua

[1] https://www.djangoproject.com/
[2] https://www.python.org/

versatilidade permite a criação de diversos tipos de aplicações, desde sistemas de gerenciamento de conteúdo até redes sociais e plataformas de computação científica. Outro diferencial é a segurança, com recursos integrados para autenticação de usuários, gerenciamento de contas e senhas, administração de conteúdo e mapeamento do *site* (DJANGO, 2023).

Assim, para melhor compreender o `Django`, é importante diferenciá-lo do `Django REST Framework`[3] porque, embora ambos sejam *frameworks* escritos em `Python`, eles possuem diferentes finalidades. Enquanto o `Django` é um *framework* completo para o desenvolvimento de aplicações web, o `Django REST Framework` é uma biblioteca baseada no `Django`, usada especificamente para criar *APIs REST*. Ainda que o `Django` tenha recursos para criar APIs, o `Django REST Framework` fornece funcionalidades específicas para facilitar a criação de *APIs REST* de forma eficiente e consistente. Sendo assim, o `Django REST Framework` funciona como um complemento do `Django`, ou seja, a instalação deste é requisito para se utilizar aquele (REST, 2024).

O ambiente de desenvolvimento `Django` é um requisito fundamental para iniciar o desenvolvimento e realizar testes em aplicações baseadas nesse *framework*. Para isso, o `Django` fornece um conjunto de *scripts* em `Python` para a criação de projetos, além de um servidor web local, executado no terminal, que permite testar a aplicação durante o desenvolvimento. No entanto, para que o `Django` seja executado corretamente, é necessário que o computador possua um ambiente de desenvolvimento `Python`, contendo, no mínimo, um interpretador e um gerenciador de pacotes. Esses componentes podem ser obtidos por meio de qualquer distribuição `Python` compatível com o sistema operacional do desenvolvedor. Após a instalação desses recursos, o próximo passo é criar um ambiente virtual com o módulo *virtualenv*, que permite isolar as dependências do projeto e garantir um conjunto independente de pacotes. Esse processo é realizado por meio da execução do comando descrito na Listagem 4.1 no *prompt* de comando do sistema operacional em uso.

```
1 pip install virtualenv
```

Listagem 4.1: Comando de instalação do módulo *virtualenv*

A seguir, ainda no *prompt* de comando e dentro do diretório do projeto, deve-se executar o comando apresentado na Listagem 4.2 para criar um ambiente virtual, denominado *venv*. Após sua criação, é necessário ativá-lo com o comando apresentado na Listagem 4.3, o que permite a instalação dos pacotes necessários ao projeto. Para desativar o ambiente virtual, basta executar o comando *deactivate*.

```
1 virtualenv venv
```

Listagem 4.2: Criando novo ambiente virtual

[3]https://www.django-rest-framework.org/

```
nome_da_virtualenv\Scripts\Activate
```

Listagem 4.3: Ativando ambiente virtual

O próximo passo é a instalação do `Django`, que pode ser realizada de três formas: a primeira, utilizando o comando *pip*; a segunda, por meio do gerenciador de pacotes do Sistema Operacional (SO); e a terceira, a partir do código-fonte. A própria documentação oficial do `Django` recomenda a instalação via *pip* (DJANGO, 2023) conforme a Listagem 4.4 que realiza a instalação do *framework*, o qual deve ser executado dentro do ambiente virtual criado.

```
pip install django
```

Listagem 4.4: Comando de instalação do `Django`

4.2 Arquitetura do Django

De acordo com sua documentação, o `Django` adota a arquitetura MTV (*Model, Template, View*), um padrão de projeto que estrutura a aplicação em camadas (DJANGO, 2023). Esse modelo separa a interação com o usuário das regras de negócio, facilitando o mapeamento do domínio por meio de URLs e tornando o código mais organizado e manutenível. A Figura 4.1 ilustra a arquitetura do `Django`, destacando a interação entre as camadas *Model*, *Template* e *View*.

Conforme ilustrado na Figura 4.1, todas as camadas da arquitetura MTV estão interligadas para prestar um serviço específico e atender às solicitações dos usuários. O fluxo inicia-se quando o usuário faz uma requisição HTTP pelo *browser*, que é processada pelas rotas definidas no arquivo *urls.py*. Em seguida, o método da camada *View* é executado, acionando o *Model* para acessar o banco de dados e obter as informações necessárias. Os dados recuperados são então renderizados pela camada *Template* e apresentados ao usuário no *browser*. Em resumo, a camada *Model* gerencia as regras de negócio, a camada *View* processa a lógica da aplicação e a camada *Template* cuida da exibição dos dados na interface do usuário. A seguir, cada uma dessas camadas será abordada em detalhes.

4.2.1 Model

A camada *Model* é responsável por definir a estrutura e o comportamento dos dados armazenados no banco de dados. No `Django`, os modelos são descritos no arquivo *models.py*, em que cada tabela do banco de dados é representada por uma classe em `Python`. Seus registros correspondem a instâncias dessa classe, e cada atributo do modelo corresponde a um campo da tabela. Uma característica essencial dos modelos no `Django` é que todas as classes herdam de *django.db.models.Model*, o que permite definir e manipular a estrutura dos dados diretamente em código

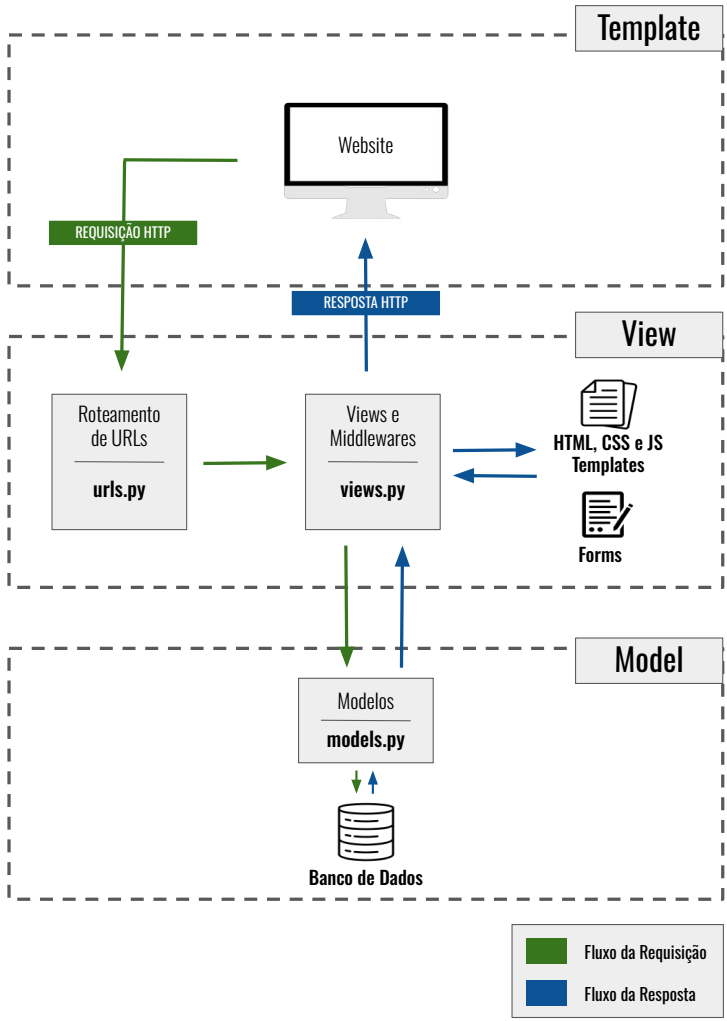


Figura 4.1: Arquitetura `Django`. Adaptado de (RAMOS, 2023)

`Python`. Essa abordagem simplifica a interação com o banco de dados e desempenha papel fundamental na API de dados do `Django`.

A API de acesso ao banco de dados do `Django` é gerada automaticamente e se baseia no padrão *Object-Relational Mapper* (ORM) (DJANGO, 2023), que é um mapeador que estabelece relações entre as tabelas, facilitando o gerenciamento dos dados. A Figura 4.2 ilustra o suporte nativo do `Django` a esse recurso. Pode-se observar que o `Django` fornece uma abstração de alto nível ao tratar os dados como

objetos, cabendo à camada ORM fazer a tradução das operações para *SQL* que, por sua vez, manipula tabelas relacionais. Com isso, o desenvolvedor trabalha com o banco de dados de forma mais eficiente e produtiva.

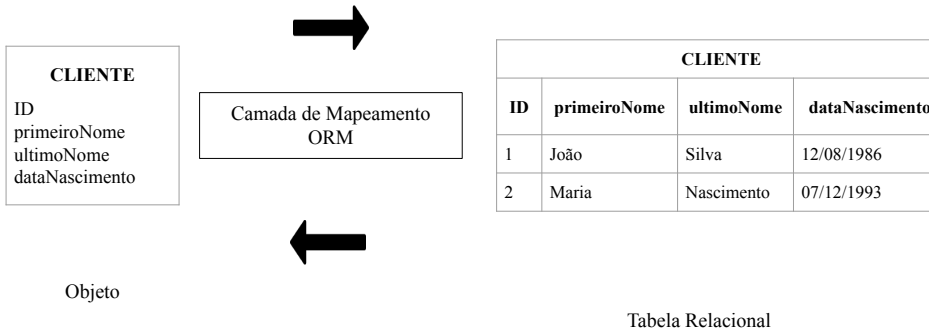


Figura 4.2: *Object-Relational Mapper* (ORM)

Uma das principais características do ORM do `Django` é o suporte a diferentes tipos de cardinalidade entre tabelas, como um para um, um para muitos e muitos para muitos. Além disso, ele oferece uma API de consulta eficiente, permitindo recuperar, filtrar, ordenar e manipular dados no banco de dados de forma intuitiva. As consultas podem ser construídas por meio de métodos encadeados, o que permite a criação de consultas complexas de forma simplificada. O ORM também permite a agregação de múltiplas operações de banco de dados em uma única transação, mantendo a atomicidade das operações. Isso garante que todas as ações sejam concluídas com sucesso ou, em caso de falha, revertidas integralmente, assegurando a consistência dos dados.

Para garantir a consistência entre a definição do modelo e o esquema do banco de dados, o `Django` dispõe de um sistema de migração integrado, conhecido como *migrations*. Essas migrações são arquivos `Python` responsáveis por registrar e gerenciar as alterações no banco de dados ao longo do desenvolvimento do projeto, assegurando que sua estrutura permaneça alinhada com as modificações feitas nos modelos. Projetadas para facilitar a adaptação do modelo a diferentes bancos de dados suportados pelo `Django`, as *migrations* simplificam a manutenção da base de dados e permitem evoluir seu esquema de forma controlada e consistente, acompanhando o crescimento da aplicação.

Além disso, o `Django` mantém um controle interno de todas as *migrations* criadas. Assim, quando uma alteração nos modelos é feita, usa-se o comando *makemigrations* para criar uma nova migração, que identifica as alterações e gera um arquivo de migração correspondente que descreve como adequar o banco de dados atual à nova estrutura definida. Para aplicar as mudanças ao banco de dados, usa-se o comando *migrate* que verifica o histórico de *migrations* e aplica apenas

as alterações necessárias. Há ainda um sistema de versionamento de *migrations* no `Django`, o qual torna possível reverter uma ou mais *migrations* resultando no retorno a uma versão anterior do banco de dados.

A Figura 4.3 ilustra o fluxo de trabalho das *migrations*. O processo se inicia com a definição dos modelos no arquivo *models.py*. Sempre que uma alteração no modelo é detectada, o comando *makemigrations* é utilizado para versionar essas alterações, gerando arquivos de migração no diretório *migrations*. Em seguida, as migrações são aplicadas ao banco de dados pelo comando *migrate*, garantindo que as mudanças sejam refletidas na estrutura do banco de dados de forma controlada e segura.

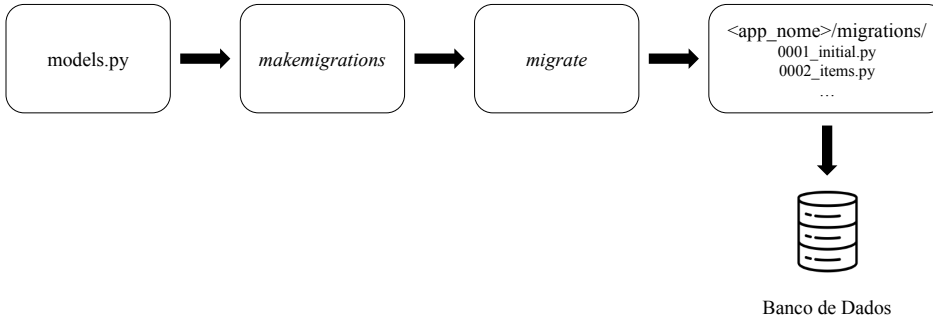


Figura 4.3: Fluxo de trabalho das migrações

Embora na prática o `Django` suporte vários Sistemas de Gerenciamento de Banco de Dados (SGBDs), desde que realizados os devidos ajustes recomendados pela respectiva documentação do SGBD, na documentação oficial do `Django` o suporte é atestado apenas para `PostgreSQL`, `MariaDB`, `MySQL`, `Oracle` e `SQLite`, sendo, ainda, recomendado utilizar `SQLite` para projetos pequenos ou para aqueles que não operem em um ambiente de produção (DJANGO, 2023). No entanto, como o `SQLite` apresenta muitas peculiaridades em comparação com outros bancos de dados, é recomendável desenvolver a aplicação com o banco de dados que se planeja usar em produção.

Por padrão, a configuração do `Django` usa o banco de dados `SQLite`, contudo, para configurar projetos que utilizem outro banco de dados, basta instalar as configurações de conexão de banco de dados apropriadas e alterar os parâmetros no item *DATABASES "default"* no arquivo *listadecompromissos/settings.py*. Em alguns casos, podem ser necessárias configurações adicionais, como *USER*, *PASSWORD* e *HOST*. A Listagem 4.5 apresenta um exemplo de uso do banco de dados `PostgreSQL` e as respectivas configurações necessárias.

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": "meudatabase",
        "USER": "meudatabaseuser",
        "PASSWORD": "minhasenha",
        "HOST": "127.0.0.1",
        "PORT": "5432",
    }
}
```

Listagem 4.5: Exemplo de uso do banco de dados `PostgreSQL`

4.2.2 View

A camada *View* é responsável por processar as requisições dos usuários, interagir com o modelo de dados e retornar uma resposta adequada. Em termos de relacionamento, ela depende da camada *Model* para acessar e manipular os dados, permitindo que os modelos forneçam as informações necessárias para serem renderizadas. Cada *view* retorna um objeto HTTP com o conteúdo da página requisitada e é responsável por renderizar os *templates*. Além disso, pode lidar com formulários por meio de uma API nativa, realizar autenticação de usuários, efetuar redirecionamentos de URLs e executar outras tarefas relacionadas à interface e à lógica de interação com o usuário.

No `Django` é possível definir uma *view* como uma função ou como uma classe. Ao definir uma *view* como função, usa-se a sintaxe de funções do `Python`, que recebe como argumento um objeto de requisição e retorna uma resposta HTTP. Já na definição como classe, a *view* herda uma das classes de visualização fornecidas pelo `Django`, como, por exemplo, a "*View*" ou "*TemplateView*", sendo essa uma abordagem útil quando há necessidade de utilizar recursos adicionais oferecidos por essas classes. Dessa forma, a escolha da abordagem para definição de uma *view* depende das necessidades do projeto e dos tipos de recursos a serem implementados. Ao definir uma *view*, esta deve ser mapeada a uma URL específica, permitindo que o `Django` encaminhe as requisições do usuário para a *view* correta.

A Listagem 4.6 apresenta um exemplo de definição de *view* baseada em função, cuja estrutura se organiza da seguinte forma: na Linha 1, realiza-se a importação do módulo `django.http`, utilizado para o envio da resposta HTTP; na Linha 3, define-se a função propriamente dita; as Linhas 4 a 6 compõem o corpo da função; e, na Linha 7, é especificado o retorno da resposta. Por sua vez, a Listagem 4.7 exibe um exemplo de definição de *view* baseada em classe, estruturado da seguinte maneira: na Linha 2, importa-se o módulo `django.views`, responsável por suportar *views* baseadas em classes; na Linha 4, declara-se a classe, que herda da classe `View`; e, a partir da Linha 5, define-se um método que, na Linha 9, retorna uma instância de resposta HTTP.

```python
from django.http import HttpResponse

def new_view(request):
    # Lógica da view

    # Resposta da view
    return HttpResponse("Hello, world!")
```

Listagem 4.6: Exemplo de definição de *view* como função

```python
from django.http import HttpResponse
from django.views import View

class NewView(View):
    def get(self, request):
    # Lógica da view para a requisição GET

    # Resposta da view
    return HttpResponse("Hello, world!")
```

Listagem 4.7: Exemplo de definição de *view* como classe

As requisições do usuário são processadas a partir da URL que este desejar acessar, cabendo ao próprio `Django` fazer o roteamento das requisições por meio do arquivo *urls.py*, que é um módulo `Python` que mapeia padrões URL e funções `Python`, relacionando-as. Todo esse processo é representado esquematicamente na Figura 4.4, seguindo o seguinte fluxo: quando o usuário requisita uma página, o `Django` percorre cada um dos padrões de URL na ordem do código até encontrar a URL que corresponda à requisição; em seguida, a *view* associada é processada e o *template* é renderizado, para, por fim, a página solicitada ser enviada como resposta ao usuário. Embora o teste seja feito com cada um dos módulos, essa operação é rápida, pois os caminhos são compilados em expressões regulares. Se nenhuma das URLs corresponder à requisição, o `Django` executa uma *view* especial para o código de exceção 404.

Durante o fluxo de processamento no `Django`, as requisições passam, antes de alcançar a *view* correspondente, por componentes leves e de baixo nível chamados *middlewares*. Esses elementos são responsáveis por processar dados de entrada e de saída, podendo atuar em diversas frentes, como o gerenciamento de requisições, o tratamento de erros e até mesmo a modificação global das próprias requisições. Além disso, podem executar ações adicionais ou interromper o processamento quando necessário. A Figura 4.5 ilustra a interação entre a requisição e a resposta com os *middlewares*: a requisição percorre os *middlewares* definidos no arquivo *settings.py*, respeitando a ordem estabelecida; em seguida, a *view* correspondente é executada. Após o processamento, a resposta percorre novamente os *middlewares*, agora em sentido inverso, até ser finalmente enviada ao usuário (HOLOVATY; KAPLAN-MOSS, 2009).

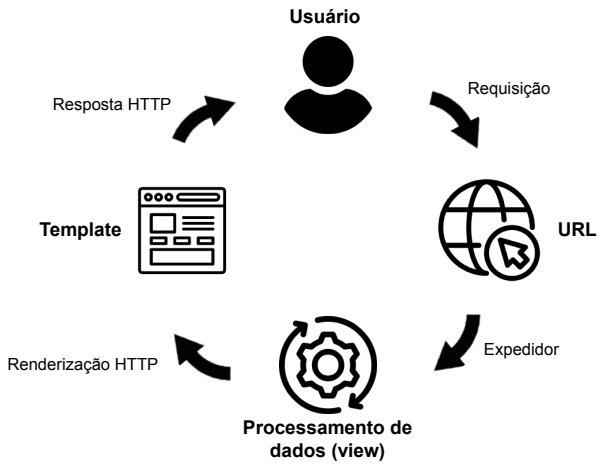


Figura 4.4: Fluxo de processamento de requisições de URLs. Adaptado de (ANKUSH_953, 2016)

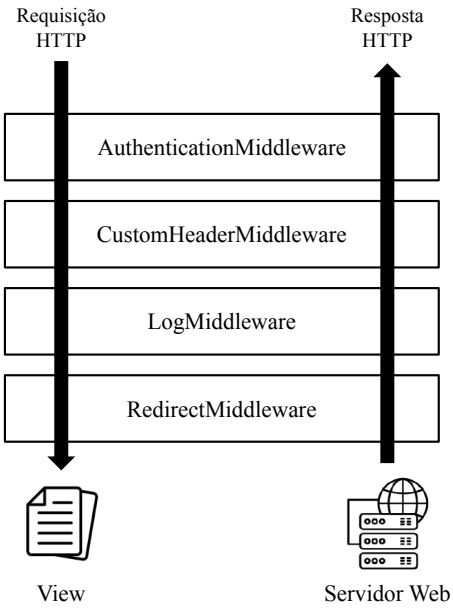


Figura 4.5: Diagrama de *Middleware*. Adaptado de (BOUCHER, 2016)

Os *middlewares* são implementados como classes que definem métodos específicos, chamados durante o processamento das requisições, os quais podem realizar ações específicas antes, durante ou após o processamento das *views* (HOLOVATY; KAPLAN-MOSS, 2009). As configurações de *middlewares* são especificadas no arquivo *settings.py* na forma de lista e a ordem em que estão listados é a ordem de execução durante o processamento das requisições. A ordem de execução é importante, pois cada *middleware* pode modificar a requisição e a resposta antes de passá-las ao próximo *middleware*. Embora ao criar um projeto, certos *middlewares* sejam adicionados automaticamente, é possível adicionar novos, remover ou reorganizar os *middlewares* de acordo com as necessidades do projeto.

4.2.3 Template

A camada *Template* é responsável pela renderização das páginas da web a partir dos dados recebidos da camada *View*, obtidos por intermédio do mecanismo de renderização do `Django`, retornando para o cliente a página com as informações requisitadas, garantindo a separação da lógica de apresentação da lógica de negócios, promovendo uma melhor organização do código. Estruturalmente, um *template* é um arquivo de texto que define a estrutura da página exibindo informações obtidas na *view*, por meio da junção entre HTML e código `Python`, permitindo inserir dinamicamente informações na página por meio da substituição de marcações.

Dessa forma, as marcações no *template* são elementos especiais que permitem a inserção dinâmica de dados e a criação de estruturas condicionais, podendo ser de dois tipos: *tags* ou filtros. Enquanto as *tags* permitem controlar o fluxo de execução e a estrutura do conteúdo exibido, os filtros permitem manipular valores de variáveis diretamente nos *templates*. O `Django` oferece uma ampla variedade de *tags* e filtros embutidos, e também permite escrever *tags* e filtros personalizados para atender às necessidades específicas do projeto. A documentação oficial dispõe de uma lista completa e detalhada de todas as *tags* e filtros disponíveis (DJANGO, 2023).

Há ainda três maneiras de utilizar as marcações apresentadas na Listagem 4.8. A primeira forma é a variável de *template*, delimitada por chaves duplas "{{ }}", que permite exibir dados de uma variável obtidos pela *view*, como exemplificado na Linha 1. A segunda forma é o *filter* que também é representado por chaves duplas, porém contém dentro das chaves o caractere *pipe* "{{|}}" conforme a Linha 3, adicionando uma formatação à informação que será renderizada e permitindo o encadeamento de filtros. A terceira forma é a *tag expression* delimitada por "{% %}", exemplificada nas Linhas 5 a 8, que permite executar comandos `Python` dentro dos *templates*.

```
<p>{{ aluno.nome }}</p>

<p>{{ aluno.data_de_nascimento | date:'d/m/Y'}}</p>

{% for aluno in alunos %}
    <p>{{ aluno.nome }}</p>
    <p>{{ aluno.matricula }}</p>
{% endfor %}
```

Listagem 4.8: Exemplos de marcações de *template*

Adicionalmente, o `Django` utiliza o conceito de "herança de *templates*" que permite reduzir a redundância nos *templates*, definindo apenas o que é único. Assim, é possível ter uma página *base.html* como um "esqueleto", semelhante a que é apresentada na Listagem 4.9, que contenha os elementos comuns às páginas do *site*. Ainda neste *template*, é definida uma série de blocos para que *templates* filhos possam preenchê-los. Na Listagem 4.10 é apresentado um exemplo de *template* filho, em que a *tag* {% extends %} indica a herança do *template* pai. O uso da "herança de *templates*" permite criar múltiplas versões de um *site* com diferentes *templates* base, como, por exemplo, a criação de versões mobile de uma aplicação web originalmente desenvolvida para computador.

```
<html>
<head>
   <title>{% block title %}{% endblock %}</title>
</head>
<body>
    {% block content %}{% endblock %}
</body>
</html>
```

Listagem 4.9: Exemplo de *template base.html*

```
{% extends "base.html" %}

{% block title %}Lista de alunos de {{ semestreIngresso }}{% endblock
    %}

{% block content %}
<h1>Lista de alunos de {{ semestreIngresso }}</h1>

{% for aluno in alunos %}
    <p>{{ aluno.nome }}</p>
    <p>Matrícula: {{ aluno.matricula }}</p>
{% endfor %}
{% endblock %}
```

Listagem 4.10: Exemplo de *template* filho

Por fim, o `Django` interpreta arquivos adicionais, como imagens, arquivos `JavaScript` e `CSS`, como arquivos estáticos que, por padrão, são armazenados em uma pasta chamada *static*, localizada dentro do diretório *app*. Na Listagem 4.11 é apresentada, na Linha 1, a *tag* {% *load static* %}, utilizada para referenciar arquivos estáticos; já nas Linhas 7 e 9, exemplos de arquivos são incluídos na página, sendo eles, respectivamente, uma imagem e um arquivo `JavaScript`.

```
1  {% load static %}
2  <html>
3  <head>
4      <title>Título da página</title>
5  </head>
6  <body>
7      <img src="{% static 'images/sitelogo.png' %}" alt="Logo">
8
9      <script src="{% static 'script.js' %}"></script>
10 </body>
11 </html>
```

Listagem 4.11: Exemplo de uso de arquivos estáticos

Dessa forma, as páginas de exibição de erro podem ser facilmente substituídas quando for necessário um comportamento personalizado. Inclusive, a documentação do `Django` (DJANGO, 2023) sugere a criação de uma página de erro consistente para todo o *site*, de modo que o `Django` capture a página e a retorne como a página de erro padrão da aplicação, juntamente com um código de erro HTTP. Além disso, dispõe de *middlewares* exclusivos para o tratamento de erros, ideais para detectar URLs inválidas.

4.3 Vantagens do Django

Interface Admin: o `Django` oferece uma interface de administração nativa, semelhante a um site, conforme ilustrado na Figura 4.6. Essa interface é personalizável por meio da classe *ModelAdmin* e permite que usuários com perfil administrativo modifiquem ou removam objetos no banco de dados de forma intuitiva, eliminando a necessidade de desenvolver uma interface própria para essas operações. O usuário administrador é ativado por padrão no modelo de projeto criado pelo comando `startproject`. Para adicionar um novo usuário administrador, utiliza-se o comando `createsuperuser`. O site de administração pode ser acessado na URL padrão `/admin/`.

Sistema de autenticação de usuário: o `Django` possui um sistema de autenticação e autorização de usuário que lida com contas de usuário, grupos, permissões e sessões baseadas em *cookies* (DJANGO, 2023), permitindo verificar as credenciais do usuário e definir as permissões de acesso de cada um (DOCS, 2024). A estrutura de autenticação visa ser genérica e flexível, permitindo criar, por exemplo, URLs e formulários, a partir do zero ou chamando a API fornecida pelo *framework*

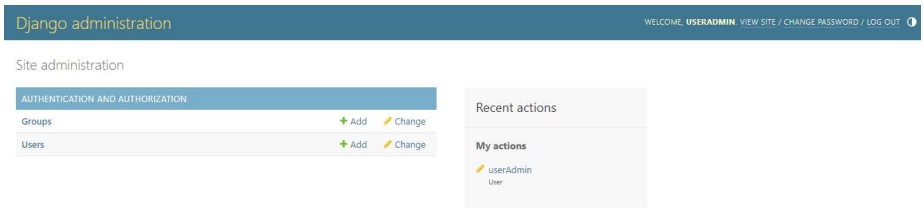


Figura 4.6: Fragmento de tela inicial do site de administração

para efetuar o *login* do usuário (DOCS, 2024). Além disso, permite personalizar a autenticação de acordo com as necessidades do projeto a ser desenvolvido (DJANGO, 2023).

4.4 Exemplo prático com Django

O exemplo apresentado nesta seção consiste em uma aplicação de Lista de Compromissos, desenvolvida com o *framework* `Django`, que permite ao usuário cadastrar, editar, visualizar e excluir atividades por meio de uma interface web. Utilizando um banco de dados, a aplicação armazena de forma persistente a descrição da atividade fornecida pelo usuário, bem como a data e o horário do cadastro, garantindo que essas informações não sejam perdidas mesmo após o recarregamento da página. As Seções 4.4.1 e 4.4.2 detalham o processo completo de desenvolvimento dessa aplicação.

4.4.1 Preparação do ambiente

Após a instalação do *framework*, para inicializar um novo projeto, executa-se o comando da Listagem 4.12 que, neste caso, inicializa um projeto chamado *listadecompromissos*. Nesse comando, o termo *startproject* cria automaticamente um diretório interno chamado, também, de *listadecompromissos*, além de arquivos de inicialização, cada qual com um propósito específico. Tais arquivos de inicialização são ilustrados na Figura 4.7, que contém os nomes dos arquivos e a organização dos diretórios.

```
1 django-admin startproject listadecompromissos
```

Listagem 4.12: Inicializando projeto

O diretório *listadecompromissos/* é um contêiner para o projeto e só contém o *script* `Python` *manage.py*, utilitário de linha de comando para tarefas administrativas,

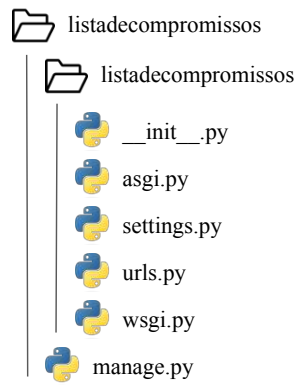


Figura 4.7: Arquivos de Inicialização

e um diretório homônimo que constitui o pacote para o projeto, composto pelos seguintes arquivos:

- *__init__.py*: arquivo vazio que serve para indicar ao `Python` que naquele diretório contém código executável;
- *settings.py*: módulo de configuração do projeto;
- *urls.py*: contém o mapeamento das URLs internas do projeto como um índice do *site*;
- *asgi.py* e *wsgi.py*: pontos de integração para servidores web compatíveis com ASGI e WSGI, respectivamente.

Para iniciar o servidor web local de hospedagem do projeto para testes, é preciso acessar o diretório *listadecompromissos/* no terminal e executar o arquivo *manage.py* com a tarefa *runserver*, conforme a Listagem 4.13. Em seguida, deve-se acessar, por meio do *browser*, o endereço http://127.0.0.1:8000, que acarreta a exibição da tela inicial padrão de uma aplicação `Django`, conforme a Figura 4.8, a qual contém instruções básicas para personalizar projetos e *links* úteis com informações extras.

```
1 python manage.py runserver
```

Listagem 4.13: Inicializando servidor web local

A partir de agora, o projeto está configurado para trabalhar com as *apps*, que são aplicações web que desempenham uma função específica, como, por exemplo, o gerenciamento das tarefas da Lista de Compromissos. O projeto é, então, uma coleção de configurações e *apps* para um determinado *site*, podendo o próprio projeto conter várias *apps*.

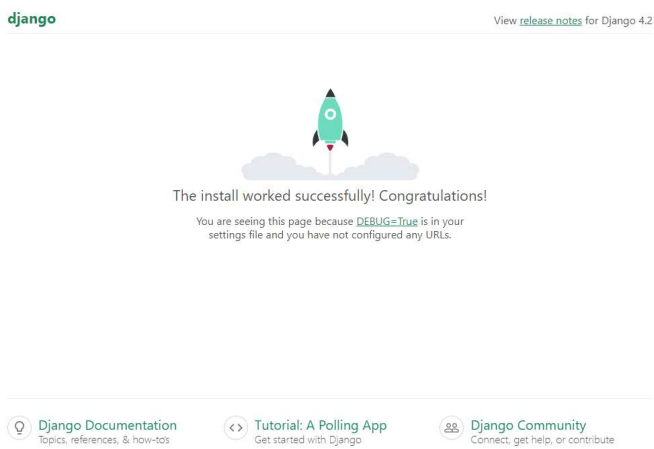


Figura 4.8: Tela inicial de uma aplicação `Django`

4.4.2 Implementação do exemplo

As *apps* podem ficar em qualquer diretório no caminho de busca dos módulos `Python`. Neste projeto, a *app* será criada no mesmo diretório do arquivo *manage.py*, para ser importada como módulo de nível superior; para isso, navega-se, no terminal, para o mesmo diretório em que o *manage.py* se encontra e executa-se o comando da Listagem 4.14 para criar uma *app* referente às tarefas da lista de compromissos, isto é, um novo diretório chamado "tarefas". Em seguida, cria-se uma nova *view* com o código `Python` da Listagem 4.15 no arquivo *tarefas/views.py*, que adiciona uma função *index* com uma resposta HTTP como retorno. Por enquanto, o retorno HTTP da *view* criada será usado apenas para verificar se o acesso está correto.

```
python manage.py startapp tarefas
```

Listagem 4.14: Criando *app* "tarefas"

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world!")
```

Listagem 4.15: Criando *view* de teste

Para que uma *view* seja acessível, é necessário mapear sua URL criando um arquivo `urls.py` no diretório da aplicação de tarefas. Nesse arquivo, utiliza-se a função `path`, importada do módulo `django.urls`, para definir a rota `index`,

conforme ilustrado na Listagem 4.16. A função `path` recebe dois argumentos obrigatórios, `route` e `view`, e um argumento opcional, `name`. O argumento `route` é uma *string* que define o padrão da URL, enquanto `view` corresponde à função que será executada quando a URL requisitada atender ao padrão definido. Já o argumento `name` permite referenciar essa *view* em outras partes do projeto, como em arquivos de *template* ou em redirecionamentos.

```
from django.urls import path
from . import views

    urlpatterns = [
        path("", views.index, name="index"),
    ]
```

Listagem 4.16: Mapeando *view index*

Para que a nova *app* seja reconhecida pelo projeto, é necessário importar a função `include` do módulo `django.urls` no arquivo `listadecompromissos/urls.py` e adicioná-la à lista `urlpatterns`, conforme ilustrado na Listagem 4.17. Essa configuração permite que outras *URLconfs* sejam referenciadas no projeto. Em seguida, o servidor deve ser reiniciado com o comando apresentado na Listagem 4.13. Após a inicialização, acessa-se a URL `http://127.0.0.1:8000/tarefas/` no navegador para verificar se a *view* está funcionando corretamente. A página deverá exibir a mensagem "*Hello, world!*", enviada como resposta pela *view*.

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path("tarefas/", include("tarefas.urls")),
    path("admin/", admin.site.urls),
]
```

Listagem 4.17: Mapeando a *app* "tarefas"

No arquivo *listadecompromissos/settings.py* estão definidas as configurações do banco de dados, além da lista `INSTALLED_APPS`, que reúne todas as aplicações ativas no projeto. A documentação oficial do `Django` (DJANGO, 2023) descreve a configuração padrão definida na inicialização do projeto. Para utilizar essas aplicações, é necessário criar as tabelas no banco de dados, já que algumas dependem de pelo menos uma estrutura de armazenamento. Para isso, executa-se o comando `migrate`, conforme apresentado na Listagem 4.18, que analisa as aplicações registradas em `INSTALLED_APPS` e gera as tabelas conforme as configurações do banco de dados. Neste projeto, foi utilizado o `SQLite`, escolhido por sua simplicidade de uso.

```
python manage.py migrate
```

Listagem 4.18: Criando tabelas no banco de dados

O projeto "Lista de Compromissos" é composto por apenas um modelo chamado "Tarefa", que possui dois campos: o conteúdo da tarefa e a data de criação. Para isso, no arquivo *tarefas/models.py*, adiciona-se o conteúdo da Listagem 4.19. Pode-se observar que "Tarefa" é uma classe simples do `Python`, na qual cada campo é representado por uma instância da classe *Field*. O *CharField* é utilizado para campos de texto e exige, obrigatoriamente, o argumento *max_length*. Já o *DateTimeField* é utilizado para campos do tipo data/hora, indicando ao `Django` o tipo de dado a ser armazenado em cada campo. Um *Field* pode conter diversos argumentos opcionais, definidos conforme a necessidade da aplicação. As instâncias "conteudo" e "criado" correspondem aos nomes dos campos que o banco de dados utilizará para nomear as colunas da tabela.

```
from django.db import models
class Tarefa (models.Model):
    conteudo=models.CharField(max_length=300)
    criado= models.DateTimeField(auto_now_add=True)
    def __str__ (self):
       return self.coteudo
```

Listagem 4.19: Criando modelo "Tarefas"

Ademais, para incluir a *app* "Tarefas" no projeto, é necessário referenciar a classe de configuração da *app*, *TarefasConfig*, na configuração de *INSTALLED_APPS*. Para tanto, como a classe *TarefasConfig* está no arquivo *tarefas/apps.py*, basta adicionar o caminho *tarefas.apps.TarefasConfig* na lista de *INSTALLED_APPS*, como mostrado na Listagem 4.20. Deve-se, ainda, incluir a *app* "Tarefas" por meio do comando *makemigrations* seguido do nome do diretório, como mostrado na Listagem 4.21 para gerar uma *migration* da alterações feitas no modelo. Para aplicar as alterações ao banco de dados, o comando da Listagem 4.18 deve ser executado novamente.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'tarefas',
]
```

Listagem 4.20: Adicionando caminho à lista "INSTALLED_APP"

```
python manage.py makemigrations tarefas
```

Listagem 4.21: Incluindo *app* "Tarefas" no banco de dados

Para criar um usuário administrador que possa acessar o *site* de administração, o comando *createsuperuser* deve ser executado conforme demonstrado na Listagem 4.22. Em seguida, no terminal, digita-se o nome de usuário desejado, um endereço de *e-mail* e uma senha. Para acessar o *site* de administração por meio do *browser*, basta navegar até o endereço http://127.0.0.1:8000/admin/ e, então, é exibida uma tela de *login* contendo um formulário de *login* que dá acesso ao sistema, como exemplifica a Figura 4.9. Ao acessar o sistema com a conta de superusuário criada, será exibida a página inicial do *site* de administração, na qual há alguns tipos de conteúdo editável, incluindo grupos e usuários, cujas funcionalidades são fornecidas pelo *framework* de autenticação do `Django`.

```
1 python manage.py createsuperuser
```

Listagem 4.22: Criando superusuário

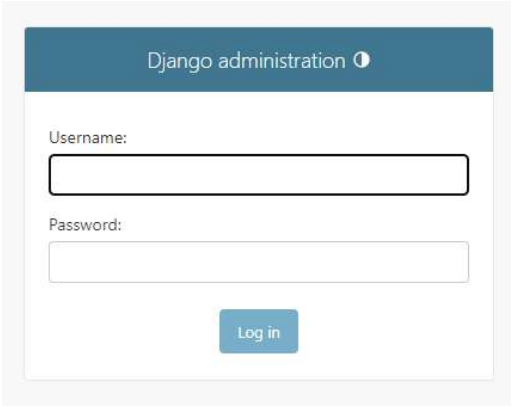


Figura 4.9: Fragmento de tela de *login* do site de administração

Para tornar a *app* "Tarefas" visível na página principal do *site* de administração, é necessário editar o arquivo *tarefas/admin.py* e registrar a *app*, conforme o código apresentado na Listagem 4.23. Na Linha 4, a função *register()* registra a *app* para que seja gerenciada pelo *site* de administração. A *app* deve então ser exibida no *site* de administração ao recarregar a página no *browser*.

```
from django.contrib import admin
from .models import Tarefa

admin.site.register(Tarefa)
```

Listagem 4.23: Ativando *app* "Tarefas" no site de administração

Já para criar os *templates*, é necessário criar um diretório chamado *templates* e, dentro dele, outro diretório chamado *tarefas*, onde será criado o arquivo *index.html* com o conteúdo apresentado na Listagem 4.24. O conteúdo dessa página, delimitado pela tag `<body>`, é composto por um formulário (Linhas 10 a 14) e por uma estrutura de repetição (Linhas 17 a 22). O formulário, que utiliza o método `POST`, é formado por um *token* (Linha 11) de proteção contra ataques de falsificação de requisições entre sites (CSRF), por um campo `input` para inserir o conteúdo da tarefa e por um botão de envio. A estrutura de repetição percorre a variável `tarefa`, exibindo seu conteúdo e data de criação, além de fornecer *links* para edição e exclusão da tarefa, cujas URLs são identificadas pelo valor `tarefa.id` passado como argumento.

```
<html lang="pt-br">
<head>
     <meta charset="UTF-8">
     <meta http-equiv="X-UA-Compatible" content="IE=edge">
     <meta name="viewport" content="width=device-width, initial-scale
         =1.0">
     <title>Lista de Compromissos</title>
</head>
<body>
    <form method="POST">
        {% csrf_token %}
        <input type="text" name='conteudo'>
        <button type="submit">Enviar</button>
    </form>
    {% for tarefa in tarefa %}
        <p>{{tarefa.conteudo}}</p>
        <p><a href="{% url 'editar_tarefa' tarefa.id %}">Editar</a></p
            >
        <p><a href="{% url 'delete' tarefa.id %}">Excluir</a></p>
        <p>{{tarefa.criado}}</p>
    {% endfor %}
</body>
</html>
```

Listagem 4.24: Conteúdo da página *index.html*

O próximo passo é atualizar o conteúdo da *view index* no arquivo *tarefas/views.py*, conforme mostra a Listagem 4.25, que exibe as tarefas cadastradas. Assim, quando uma nova tarefa é adicionada, ela é salva no banco de dados e o usuário é redirecionado para a mesma página.

```
from django.shortcuts import render, redirect, get_object_or_404
```

```
2 from .models import Tarefa
3 def index (request):
4     tarefa=Tarefa.objects.all()
5     if request.POST:
6         tarefa=request.POST.get('conteudo')
7         tarefa=Tarefa(conteudo=tarefa)
8         tarefa.save()
9         return redirect('index')
10    context={
11            'tarefa':tarefa,
12    }
13    return render(request, 'tarefas\index.html',context)
```

Listagem 4.25: Atualizando *view index*

Além disso, outra *view* deve ser adicionada, conforme o código da Listagem 4.26 no arquivo *tarefas/views.py*, abaixo da *view index*, recebendo um identificador da tarefa como argumento, indicando a tarefa a ser excluída. Na Linha 2, a variável "tarefa" recebe do banco de dados a tarefa com o *id* correspondente; na Linha 3, a tarefa é excluída do banco de dados; e, por fim, na Linha 4, o usuário é redirecionado para a *view index*, onde a página será renderizada com as tarefas restantes no banco. A nova *view excluir_tarefa* deve então ser mapeada no arquivo *tarefas/urls.py* incluindo na lista *urlpatterns* outra rota, como na Listagem 4.27, por meio da função *path()* que recebe como argumento o *id* da tarefa a ser excluída.

```
1 def excluir_tarefa(request,id):
2     tarefa=Tarefa.objects.get(id=id)
3     tarefa.delete()
4     return redirect('index')
```

Listagem 4.26: Criando nova *view excluir_tarefa*

```
1 path('delete/<int:id>/', views.excluir_tarefa, name='delete')
```

Listagem 4.27: Mapeando rota para excluir tarefa

Ademais, para adicionar uma *view* de edição de tarefa, deve-se definir a função *editar_tarefa*, conforme apresentado na Listagem 4.28, que recebe como argumentos uma requisição e o *id* da tarefa a ser editada. No corpo da função, a tarefa é buscada no banco de dados com base no *id* informado e, caso a requisição seja do tipo `POST`, o conteúdo da tarefa é atualizado com os dados enviados pelo formulário. Em seguida, as alterações são salvas no banco de dados e o usuário é redirecionado para a página inicial da aplicação.

```
def editar_tarefa(request, id):
    tarefa = get_object_or_404(Tarefa, id=id)
    if request.method == 'POST':
        conteudo = request.POST.get('conteudo')
        tarefa.conteudo = conteudo
        tarefa.save()
        return redirect('index')
    context = {
            'tarefa': tarefa,
        }
    return render(request, 'tarefas/editar.html', context)
```

Listagem 4.28: Criando nova *view editar_tarefa*

Em seguida, no diretório *templates/tarefas* deve ser adicionado um novo *template editar.html* com o conteúdo apresentado na Listagem 4.29, que cria o formulário de edição. Além disso, a nova rota de edição deve ser mapeada no arquivo *urls.py* conforme a Linha 7 da Listagem 4.30, que recebe a url e a *view* a ser exibida ao acessar a rota.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale
        =1.0">
        <title>Editar tarefa</title>
</head>
<body>
    <form method="POST">
        {% csrf_token %}
        <label for="conteudo">Editar Tarefa:</label>
        <input type="text" name="conteudo" id="conteudo" value="{{
            tarefa.conteudo }}">
        <button type="submit">Salvar</button>
    </form>
</body>
</html>
```

Listagem 4.29: Conteúdo da página *editar.html*

```
from django.urls import path
from . import views

urlpatterns = [
    path("", views.index, name="index"),
    path('delete/<int:id>/',views.excluir_tarefa,name='delete'),
    path('editar/<int:id>/', views.editar_tarefa, name='editar_tarefa'
        ),
]
```

Listagem 4.30: Mapeando rota para editar tarefa

Por fim, para estilizar a página *index.html*, utiliza-se um arquivo estático. Para isso, criam-se os diretórios *tarefas/static/tarefas* e, dentro deles, adiciona-se o arquivo *style.css* com o conteúdo apresentado na Listagem 4.31, que altera a cor de fundo da página.

```
 body {
   background-color: #E7F5CF;
}
```

Listagem 4.31: Exemplo de conteúdo da página *style.css*

Em seguida, deve-se adicionar o conteúdo da Listagem 4.32 na tag `<head>` do arquivo *index.html*, para referenciar o arquivo de estilização. Após isso, é necessário executar novamente o comando de inicialização do servidor, apresentado anteriormente na Listagem 4.13. Dessa forma, o projeto "Lista de Compromissos" finalizado resultará na interface exibida na Figura 4.10.

```
{% load static %}

<link rel="stylesheet" href="{% static 'tarefas/style.css' %}">
```

Listagem 4.32: Referenciando estilização no HTML



Figura 4.10: Tela do projeto finalizado em `Django`

4.5 Visão geral do Framework Laravel

O `Laravel`[4] é um *framework web* gratuito e de código aberto, desenvolvido em `PHP`[5]. Com uma sintaxe expressiva e elegante (LARAVEL, 2023), foi projetado para oferecer aos desenvolvedores uma experiência intuitiva e eficiente, combinando recursos avançados com uma abordagem simplificada. Considerado uma ferramenta completa para codificação (CARDOSO, 2021), o Laravel permite a criação de projetos de alta qualidade, reduzindo a necessidade de escrita manual de código. Isso é possível por meio do uso de métodos e classes, minimizando o uso direto de código nativo em `PHP` e facilitando a geração de arquivos por meio de linhas de comando (GABARDO, 2017).

Desenvolvido por Taylor Otwell, o `Laravel` teve sua primeira versão lançada em junho de 2011, trazendo recursos como um ORM próprio, denominado *Eloquent*, roteamento baseado em *closures*, funções que permitem passar objetos e parâmetros para outras funções e métodos, atribuir variáveis e até mesmo serializar. Dessa forma, Otwell decidiu criar o `Laravel` para superar as limitações que encontrou no `CodeIgniter`[6], um dos primeiros *frameworks* em `PHP`, inspirado no `Ruby on Rails`. O `CodeIgniter` era conhecido por sua simplicidade, facilidade de uso, boa documentação e forte comunidade. No entanto, apresentou lentidão na incorporação de recursos das versões mais recentes do `PHP`, o que motivou a criação do `Laravel` (STAUFFER, 2019). A partir da versão `Laravel 4`, sua estrutura passou a incorporar componentes do `Symfony`[7], outro *framework* em `PHP`, voltado a padrões de design corporativo e aplicações de comércio eletrônico. Embora tenha se inspirado no `Symfony`, com sua constante evolução, o `Laravel` tornou-se um *framework* independente e cresceu significativamente em recursos (STAUFFER, 2019).

O `Laravel` se caracteriza por possuir uma estrutura progressiva, capaz de acompanhar o crescimento do projeto, atendendo desde aplicações simples até sistemas web com cargas de trabalho corporativas. Trata-se de um *framework* escalável, apto a lidar com centenas de milhões de requisições por mês, especialmente quando combinado com serviços e ferramentas como filas, cache e balanceamento de carga. Além disso, integra-se aos principais pacotes do ecossistema `PHP`, buscando oferecer uma solução robusta e amigável ao desenvolvedor. Essa proposta resultou em uma comunidade ativa, composta por milhares de desenvolvedores que colaboram continuamente na evolução do `framework` (LARAVEL, 2023).

Assim, os principais valores do `Laravel` são promover a agilidade no desenvolvimento e aumentar a satisfação do desenvolvedor, oferecendo um código claro, simples e elegante, que favorece a escrita de soluções limpas e sustentáveis. Ter a felicidade do desenvolvedor como prioridade influenciou diretamente o estilo do *framework* e as decisões que orientaram sua evolução. O `Laravel` se destaca por

[4] https://laravel.com/
[5] https://www.php.net/
[6] ⟨https://codeigniter.com/⟩
[7] ⟨https://symfony.com/⟩

apresentar uma curva de aprendizado suave, permitindo que mesmo quem não possui um conhecimento aprofundado da linguagem `PHP` consiga iniciar e evoluir em seus projetos. Seu foco é reduzir ao máximo as etapas entre o início de uma aplicação e sua publicação. Tarefas recorrentes no desenvolvimento web, como autenticação e integração com bancos de dados, entre outras, são facilitadas pelos componentes e ferramentas que o `Laravel` oferece (STAUFFER, 2019).

O `Laravel` tem como requisito de instalação a linguagem `PHP`, podendo ser instalada por meio do pacote gratuito `XAMPP`[8], e seu gerenciador de dependências, `Composer`[9]. Após a instalação das dependências, executa-se o comando da Listagem 4.33 que instala o `Laravel` na máquina e cria um novo projeto, neste caso, sendo chamado de "novo-projeto".

```
composer create-project --prefer-dist laravel/laravel novo-projeto
```

Listagem 4.33: Comando de instalação do `Laravel` e criação de um novo projeto

4.6 Arquitetura do Laravel

A arquitetura do `Laravel` segue o padrão de projeto MVC (*Model*, *View*, *Controller*), no qual as camadas de lógica e de apresentação são separadas para melhor organização do código, além de proporcionar vantagens ao longo do ciclo de vida do projeto (GABARDO, 2017). Esse padrão arquitetural se popularizou por meio do `Ruby on Rails`, além de outras ferramentas e convenções que influenciaram o desenvolvimento de aplicações web, principalmente quanto à rapidez (STAUFFER, 2019).

Conforme ilustrado na Figura 4.11, o fluxo de dados da aplicação inicia-se com uma requisição HTTP do usuário. Essa requisição é mapeada pelas rotas e encaminhada a um *Controller*, responsável por processá-la e repassar os dados ao *Model*, que interage com o banco de dados. Após essa etapa, o *Controller* envia os dados obtidos à camada *View*, encarregada de exibi-los ao usuário na interface da aplicação.

4.6.1 Rotas

As rotas mapeiam as requisições HTTP do usuário e as redirecionam para um controlador apropriado por meio de uma lista de roteamento. Embora esta não seja, de fato, uma camada do modelo MVC, é a abordagem mais comum para tratar de requisições HTTP (GABARDO, 2017). No `Laravel`, a classe responsável por definir as rotas da aplicação e por carregar automaticamente os arquivos de rotas do diretório */routes* é a *Route Service Provider*.

[8] https://www.apachefriends.org/pt_br/index.html
[9] https://getcomposer.org/

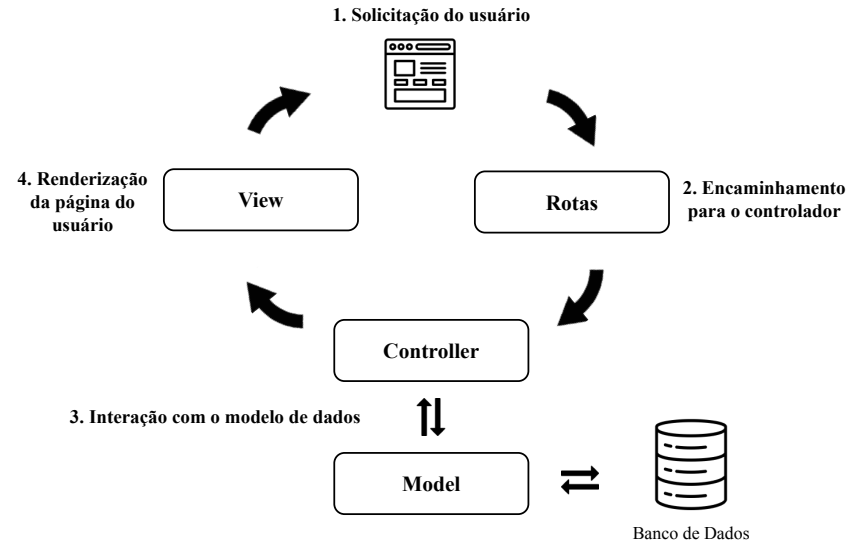


Figura 4.11: Arquitetura `Laravel`. Adaptado de (PATEL, 2023a).

Por padrão do `Laravel`, os arquivos de rotas são divididos em *web.php* e *api.php*. No arquivo *web.php* estão as rotas acessadas pelo usuário final; ou seja, são requisições filtradas por *middlewares* que fornecem recursos, como controle de sessão e encriptação de *cookies*. O termo *middleware* caracteriza uma camada intermediária que possibilita a comunicação entre aplicações distribuídas (MACIEL; ASSIS, 2004). No arquivo *api.php* estão as rotas da API (Application Programming Interface), se houver.

Há duas maneiras de definir rotas no `Laravel`, conforme apresentado na Listagem 4.34: *Route Verbs* e *Route Handling*. Nas Linhas 1 a 3, é mostrado um exemplo de rota definida por *Route Verbs*, que utiliza uma requisição do tipo `GET` acessada pela URL raiz "/" e executa uma função anônima que retorna uma *view* chamada "welcome". Também é possível passar argumentos por meio da rota. Na Linha 5, o parâmetro "nome" é passado à função anônima por meio da URL, e a instrução `echo`, na Linha 6, é utilizada para exibir conteúdo HTML na página de retorno.

Já ao utilizar o *Route Handling*, é possível otimizar o desempenho da aplicação, reduzindo até centenas de milissegundos por solicitação em arquivos com lógica de roteamento mais complexa (STAUFFER, 2019). Nesse caso, informa-se o nome do *controller* e do método como *string*, substituindo a função anônima, como mostrado na Linha 9. Isso indica ao `Laravel` que a requisição deve ser direcionada ao método `index` do *controller* localizado em `app/Http/Controllers/WelcomeController`. Assim como no *Route Verbs*, o *Route Handling* aceita os mesmos parâmetros e é

processado da mesma forma, configurando-se como alternativa para a definição de rotas.

```
Route::get('/', function () {
    return view('welcome');
});

Route::get('/aluno/{nome}', function ($nome) {
    echo "<h1>Aluno: $nome </h1>";
});

Route::get('/', 'WelcomeController@index');
```

Listagem 4.34: Definindo rotas

Em ambas as formas de definição de rotas, é possível tornar os parâmetros opcionais ao incluir um ponto de interrogação (`?`) após o nome do parâmetro, conforme ilustrado na Listagem 4.35. Nesse caso, é necessário fornecer um valor *default* para a variável correspondente, garantindo que a aplicação funcione corretamente mesmo quando o parâmetro não for informado na URL.

```
Route::get('users/{id?}', function ($id = 'fallbackId') {
     // Escopo da função
});
```

Listagem 4.35: Definindo parâmetros de rota opcionais

4.6.2 Controller

A camada *Controller* é responsável por agrupar a lógica de negócios, recebendo requisições por meio das rotas. Além disso, interage com os modelos para consumir e enviar registros, e repassa os dados necessários às *views* (GABARDO, 2017). Os *controllers* são armazenados no diretório *app/Http/Controllers* e podem centralizar o tratamento das requisições em uma única classe. Por exemplo, uma classe *UserController* pode ser responsável por manipular requisições relacionadas a usuários, como exibir, criar, atualizar ou excluir registros no banco de dados (LARAVEL, 2023).

O *Artisan* é um conjunto de comandos de linha de comando integrado ao `Laravel`, capaz de executar *migrations*, criar usuários e outros registros no banco de dados, além de realizar diversas outras tarefas (STAUFFER, 2019). Para gerar um novo *controller* via terminal, utiliza-se o comando *Artisan* `make:controller` seguido do nome desejado, conforme ilustrado na Listagem 4.36. No exemplo apresentado, o comando cria um novo *controller* chamado `UserController`.

```
php artisan make:controller UserController
```

Listagem 4.36: Criando novo *controller*

Tabela 4.1: Métodos *Resource controller*. Adaptado de (STAUFFER, 2019)

HTTP Verb	URL	Método do Controller	Descrição
GET	users	index( )	Exibe todos os usuários
GET	users/create	create( )	Exibe um formulário de criação de um novo usuário
POST	users	store( )	Aceita a submissão do formulário de criação de um novo usuário
GET	users/{user}	show( )	Exibe um usuário
GET	users/{user}/edit	edit( )	Edita um usuário
PUT/PATCH	users/{user}	update( )	Aceita o formulário de edição de um usuário
DELETE	users/{user}	destroy( )	Exclui um usuário

O `Laravel` oferece uma convenção para rotas CRUD por meio do *resource controller*. Utilizando um comando na linha de comando, como mostrado na Listagem 4.37, é possível gerar um *controller* com métodos para cada operação de recurso disponível, além de rotas prontas para uso. A Tabela 4.1 apresenta todas as ações disponíveis em um *resource controller*, cada uma associada a uma chamada padrão de URL e a um HTTP *Verb* específico. A primeira coluna da tabela apresenta o HTTP *Verb*, a segunda exibe a URL correspondente, a terceira indica o método do *controller*, e a quarta fornece a descrição de cada ação. Para vincular automaticamente as rotas listadas na Tabela 4.1 aos métodos do *controller*, utiliza-se o recurso de roteamento no arquivo *web.php*, conforme apresentado na Listagem 4.38.

```
1 php artisan make:controller UserController --resource
```

Listagem 4.37: Utilizando *resource controller*

```
1 Route::resource('tasks', 'TasksController');
```

Listagem 4.38: Referenciando rota com o *resource controller*

A ideia dos *middlewares* é que funcionem como uma camada que envolve a aplicação, inspecionando as solicitações e decidindo se devem ser aceitas ou rejeitadas com base em regras previamente definidas (STAUFFER, 2019), conforme exemplificado na Figura 4.12. Os *middlewares* estão localizados no diretório *app/Http/Middleware* e, por meio deles, é possível, por exemplo, verificar se o usuário está autenticado. Caso a condição seja atendida, a requisição é autorizada a prosseguir para a aplicação; caso contrário, o usuário é redirecionado para a tela de *login*.

Além disso, os *middlewares* podem ser aplicados à lógica da aplicação tanto por meio das rotas quanto diretamente nos *controllers*. A Listagem 4.39 apresenta um exemplo de uso de *middleware* em uma rota, na qual o *middleware* `auth` verifica a autenticação do usuário antes da execução do método `show`. De forma semelhante,

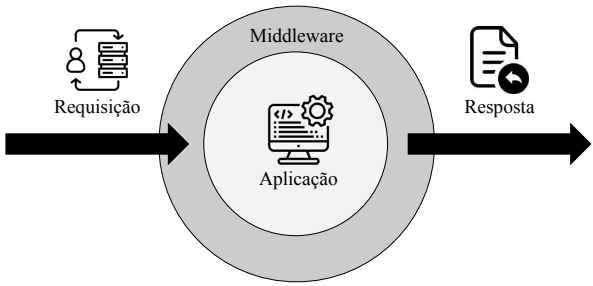


Figura 4.12: *Middleware* no `Laravel`. Adaptado de (STAUFFER, 2019).

a Listagem 4.40 apresenta um exemplo de aplicação de *middleware* diretamente em um *controller*, em que o *middleware* é atribuído a todos os métodos da classe.

```
Route::get('profile', [UserController::class, 'show'])->middleware('
    auth');
```

Listagem 4.39: Exemplo de *middleware* atribuído à rota do controlador

```
use Illuminate\Http\Request;

class UserController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth');
    }

    public function show($id)
    {
        // Lógica para mostrar um usuário específico
    }
}
```

Listagem 4.40: Exemplo de *middleware* atribuído ao controlador

Assim, todas as rotas e *controllers* devem retornar ao navegador uma resposta à solicitação do usuário. Para isso, o `Laravel` oferece diversas formas de retorno, como *strings*, *arrays* e objetos de resposta. No entanto, de forma geral, o retorno mais comum das ações de rota é a *view*, responsável por renderizar o conteúdo exibido ao usuário (STAUFFER, 2019).

4.6.3 View

A *View* é a camada de exibição da aplicação, responsável por separar a lógica do *controller* de sua apresentação (LARAVEL, 2023). As *views* são arquivos de saída armazenados no diretório *resources/views* e podem conter, conforme a necessidade do projeto, conteúdos em JSON, XML ou, mais comumente, em HTML (STAUFFER, 2019). No `Laravel`, há dois formatos principais de *views*: PHP simples ou *Blade templates*. Arquivos com a extensão `.php` são processados diretamente pelo mecanismo do `PHP`, enquanto arquivos com a extensão `.blade.php` é renderizado utilizando o mecanismo de template *Blade*.

Os arquivos que utilizam `PHP` simples seguem a sintaxe padrão da linguagem e inserem código `PHP` diretamente no arquivo para gerar o conteúdo HTML, conforme ilustrado na Listagem 4.41, utilizando a tag de abertura `<?php ?>`. No exemplo apresentado, uma mensagem contendo o nome do usuário será renderizada sempre que a condição `if($user)` for avaliada como verdadeira.

```
1 <h1>Bem-vindo</h1>
2 <p>Esse é um exemplo de view utilizando PHP simples</p>
3 <?php if($user): ?>
4     <p>Olá, <?php echo $user->name; ?>!</p>
5 <?php endif; ?>
```

Listagem 4.41: Exemplo de uso da sintaxe `PHP` simples

Os arquivos *Blade* oferecem mais funcionalidades do que o uso simples de `PHP`, fornecendo uma maneira limpa e concisa de trabalhar com estruturas de controle da linguagem (LARAVEL, 2023). Entre os principais recursos estão o uso da herança de *template*, que permite que *views* herdem seções de outras *views*, facilitando a reutilização de código, além da utilização de componentes e de *slots*. Os componentes são estruturas reutilizáveis, úteis para a componentização de partes comuns da aplicação, como cabeçalhos e rodapés, enquanto os *slots* representam áreas variáveis que podem ser preenchidas dinamicamente, proporcionando maior flexibilidade no conteúdo exibido. Também é possível utilizar diretivas específicas para estruturas de repetição e condicionais, o que torna a incorporação de lógica em arquivos de *template* mais legível e organizada (STAUFFER, 2019). A Listagem 4.42 apresenta um exemplo de arquivo *Blade* que exibe, na tela, cada item de uma lista, organizado com a tag `<li>`. Além disso, o uso das chaves duplas ({{ }}), na Linha 4, é empregado para encapsular e imprimir o valor de uma variável em tempo de execução.

```
1 <h1>Bem-vindo</h1>
2 <p>Esse é um exemplo de view utilizando Blade</p>
3 @foreach($items as $item)
4     <li>{{ $item }}</li>
5 @endforeach
```

Listagem 4.42: Exemplo de uso da sintaxe do *Blade*

O mecanismo de *template Blade*, ao contrário de outros mecanismos de *template* em `PHP`, não impõe restrições ao uso de código `PHP` simples nas *views*. Isso porque, em vez de serem executadas diretamente pelo servidor web, as *views* Blade são compiladas em código `PHP` puro e armazenadas em *cache* até que sejam modificadas. Dessa forma, não se adiciona nenhuma sobrecarga (*overhead*) ao desempenho da aplicação (LARAVEL, 2023).

Para criar uma nova *view* no terminal, utiliza-se o comando `Artisan make:view`, como mostrado na Listagem 4.43, que gera uma *view* chamada `users` com a extensão `.blade.php`. Em seguida, é possível exibir essa *view* na aplicação por meio de uma rota, utilizando o auxiliar global `view`, conforme ilustrado na Listagem 4.44. Nesse exemplo, o primeiro argumento corresponde ao nome da *view*, e o segundo é um *array* de dados que estarão disponíveis dentro dela, como o parâmetro `name`. Como alternativa ao uso de um *array* completo como segundo argumento da função `view`, pode-se utilizar o método `with`, como demonstrado na Listagem 4.45. Esse método retorna uma instância do objeto `view`, permitindo o encadeamento de métodos antes da renderização.

```
1 php artisan make:view users
```

Listagem 4.43: Comando de exemplo para criar uma *view*

```
1 Route::get('/', function () {
2     return view('user', ['name' => 'Maria']);
3 });
```

Listagem 4.44: Exibindo *view* de exemplo por meio da rota

```
1 Route::get('/', function () {
2     return view('user')
3         ->with('name', 'João')
4         ->with('occupation', 'Professor');
5 });
```

Listagem 4.45: Utilizando método *with* na exibição da *view* de exemplo

4.6.4 Model

A camada *Model* do padrão MVC é responsável por objetos que manipulam os dados da aplicação por meio de um ORM (Mapeamento Objeto-Relacional). No `Laravel`, o pacote *Eloquent* é o ORM utilizado para implementar essa camada, destacando-se por sua simplicidade e expressividade (STAUFFER, 2019). Com o *Eloquent*, cada tabela do banco de dados possui uma classe correspondente que permite interagir diretamente com ela, possibilitando operações como recuperação, inserção, atualização e exclusão de registros (LARAVEL, 2023). Os arquivos

do *Model* estão localizados no diretório *app/Models* e estendem a classe base `Illuminate\Database\Eloquent\Model`.

Além disso, o `Laravel` permite a definição de múltiplas conexões de banco de dados, possibilitando selecionar qual delas será utilizada por padrão por meio da interface fornecida pelo *Eloquent*, o que garante maior flexibilidade no gerenciamento de dados (STAUFFER, 2019). Esse suporte a múltiplas conexões permite, por exemplo, utilizar diferentes bancos de dados para distintos propósitos na mesma aplicação, como ler dados de uma fonte e gravar em outra.

Dessa forma, o `Laravel` simplifica a interação com o banco de dados e oferece suporte a uma variedade de sistemas, permitindo o uso de `SQL` bruto, de um construtor de consultas fluente e do ORM *Eloquent*. O *framework* fornece suporte nativo para os seguintes bancos de dados: `MariaDB`[10], `MySQL`[11], `PostgreSQL`[12], `SQLite`[13] e `SQL Server`[14] (LARAVEL, 2023). As configurações dos serviços de banco de dados estão localizadas no arquivo *config/database.php*, onde é possível definir múltiplas conexões e especificar qual delas será utilizada como conexão padrão pela aplicação.

O *Eloquent* fornece um gerenciamento eficiente para múltiplos relacionamentos entre tabelas, facilitando o trabalho com estruturas de dados mais complexas. Os relacionamentos são definidos como métodos nas classes de modelo, e essas definições funcionam como construtores de consultas, permitindo o encadeamento e a execução de métodos de forma poderosa e expressiva. Além disso, o construtor de consultas do `Laravel` oferece uma interface fluente e intuitiva para criar e executar comandos no banco de dados, sendo compatível com todos os sistemas de banco suportados pelo `Laravel` (LARAVEL, 2023).

No `Laravel`, utilizam-se as *migrations* como forma de controle de versão do banco de dados (LARAVEL, 2023). Uma *migration* é um arquivo que contém a lógica necessária para criar, modificar ou excluir estruturas no banco de dados, como tabelas e colunas, e é executada em ordem cronológica, com base na data de criação. Essa organização torna possível, inclusive, reverter uma *migration* já aplicada. Cada classe de *migration* contém dois métodos principais: `up` e `down`. O método `up` é responsável pela criação de novas tabelas, colunas ou índices, enquanto o método `down` desfaz as operações executadas por `up`, permitindo a reversão dessas alterações.

4.6.5 Vantagens do Laravel

Starter Kits: o `Laravel` fornece *kits* de autenticação e inicialização de aplicativos que oferecem uma vantagem inicial no processo de desenvolvimento. Esses *kits* estruturam o aplicativo com as rotas, *controllers* e *views* necessárias para

[10] ⟨https://mariadb.org/⟩
[11] ⟨https://www.mysql.com/⟩
[12] ⟨https://www.postgresql.org/⟩
[13] ⟨https://www.sqlite.org/⟩
[14] ⟨https://www.microsoft.com/pt-br/sql-server⟩

funcionalidades como registro e autenticação de usuários (LARAVEL, 2023). Para uma configuração rápida do sistema de autenticação, o `Laravel` disponibiliza o *Starter Kit* chamado *Laravel Breeze*, que apresenta uma implementação mínima e simples (LARAVEL, 2023), oferecendo recursos como *login*, registro, redefinição de senha e verificação de e-mail. A camada *view* do *Laravel Breeze* é composta por *templates Blade* estilizados com `Tailwind CSS`[15]. Outro *Starter Kit* disponível é o *Laravel Jetstream*, que expande as funcionalidades do *Laravel Breeze* ao incluir recursos como autenticação em dois fatores, gerenciamento de sessões, entre outros.

Manipulador de erros: o `Laravel` já vem configurado, ao iniciar um novo projeto, com um sistema de tratamento de erros e exceções. Entre os recursos oferecidos estão as páginas de erro padrão para códigos de *status* HTTP, como 404 e 500, localizadas no diretório *resources/views/errors*. Os erros são registrados pela classe `App\Exceptions\Handler` e, em seguida, exibidos em tela. No arquivo *config/app.php*, a opção `debug` determina o nível de detalhamento das informações exibidas ao usuário em caso de erro. Por padrão, essa opção é controlada por uma variável de ambiente definida no arquivo *.env*. Durante o desenvolvimento, é comum que o valor dessa variável seja definido como `true`, permitindo a visualização completa dos erros. No entanto, em ambiente de produção, o valor deve sempre ser `false`, a fim de evitar a exposição de informações sensíveis da aplicação ao usuário final (LARAVEL, 2023).

4.7 Exemplo prático com Laravel

O exemplo apresentado nesta seção consiste em uma aplicação de Sistema de Gerenciamento de Contatos, desenvolvida com o *framework* `Laravel`. A aplicação permite adicionar contatos, visualizá-los em uma página, além de oferecer funcionalidades para editar e excluir registros da lista. Com o uso de um banco de dados, as informações são armazenadas de forma persistente, não sendo perdidas mesmo após o recarregamento da página. As Seções 4.7.1 e 4.7.2 descrevem

4.7.1 Preparação do ambiente

Para a implementação deste projeto, ao instalar o `Laravel`, conforme apresentado na Listagem 4.33, o nome do projeto deve ser definido como "gerenciamento-de-contatos", substituindo o termo "novo-projeto" utilizado no comando de exemplo. Após a criação do projeto, é necessário navegar até o diretório correspondente e executar o comando apresentado na Listagem 4.46 para iniciar o servidor local com o *Laravel Artisan*. Em seguida, acessa-se o endereço *http://localhost:8000* no navegador, onde será exibida a página inicial padrão de uma aplicação `Laravel`, com links úteis e informações, como as versões do `Laravel` e do `PHP` utilizadas no projeto, conforme ilustrado na Figura 4.13.

[15] ⟨https://tailwindcss.com/⟩

```
1 php artisan serve
```

Listagem 4.46: Inicializando servidor local do `Laravel`

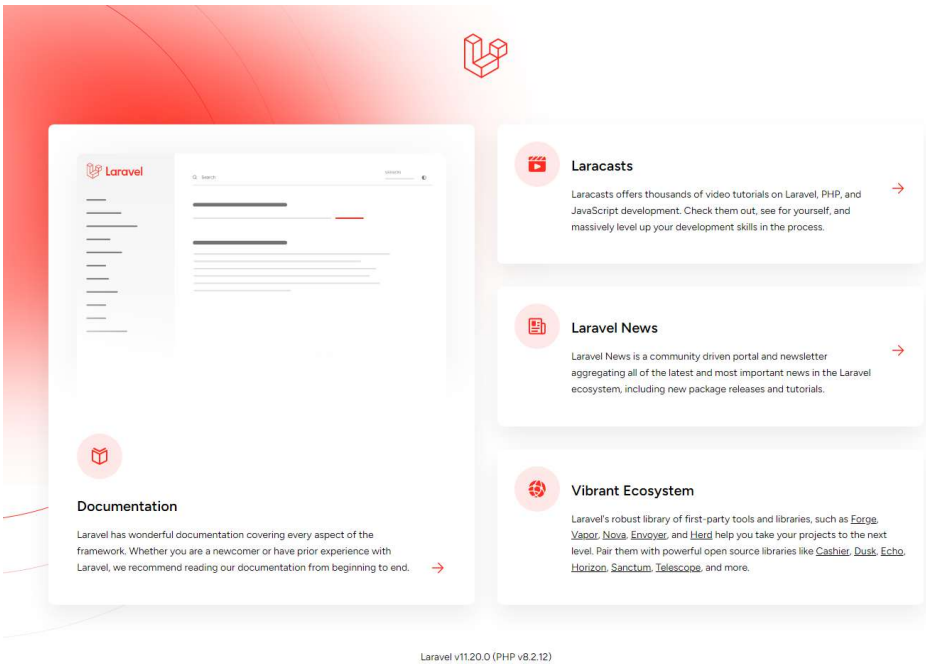


Figura 4.13: Tela inicial de uma aplicação `Laravel`

4.7.2 Implementação do exemplo

No arquivo *routes/web.php*, onde são definidas as rotas da aplicação, há uma rota responsável por exibir a página inicial padrão do projeto. Essa rota deve ser substituída por uma nova, chamada `/contatos`, conforme ilustrado na Listagem 4.47, que utiliza um *resource controller*. Essa definição recebe, como argumentos, a URL da rota e o nome do *controller*, `ContatosController`. Como demonstrado na Linha 4, é necessário que o *controller* seja corretamente referenciado no arquivo de rotas para que a aplicação funcione conforme o esperado.

```php
<?php

use Illuminate\Support\Facades\Route;
use App\Http\Controllers\ContatosController;

Route::resource('contatos', ContatosController::class);
```

Listagem 4.47: Configuração da rota contatos

Para criar o *controller* `ContatosController`, referenciado na rota definida anteriormente, deve-se executar, no terminal, o comando apresentado na Listagem 4.48. Esse comando cria um novo *controller* e adiciona, por padrão, os métodos correspondentes às operações CRUD. O arquivo gerado, localizado em *app/Http/Controllers/ContatosController.php*, contém uma classe que estende a classe `Controller` nativa do `Laravel` e é responsável por agrupar a lógica associada ao *controller*.

```
php artisan make:controller ContatosController --resource
```

Listagem 4.48: Criando um novo *controller*

Ao escopo do método `index`, adiciona-se o conteúdo da Listagem 4.49, que define, por enquanto, de forma estática, um *array* de contatos enviados para a *view* retornada pela função. Para criar essa *view*, deve-se adicionar um novo arquivo ao diretório *resources/views*, com o nome `listar-contatos.blade.php`. O código referente à estrutura da página, apresentado na Listagem 4.50, deve ser inserido nesse arquivo. A página contém um formulário para criar novos contatos e uma lista de contatos existentes. Na Linha 12, é inserido o token `@csrf`, necessário para que o *middleware* de proteção contra requisições forjadas possa validar a solicitação do tipo `POST` enviada pelo formulário.

```
public function index() {
    $contatos = [
        'João',
        'Maria'
    ];
    return view('listar-contatos', ['contatos' => $contatos]);
    };
```

Listagem 4.49: Adicionando escopo do método *index*

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale
        =1.0">
        <title>Lista de Contatos</title>
</head>
<body>
    <form action="/contatos" method="POST">
        @csrf
    <input type="text" name='nome' placeholder="Nome">
    <input type="text" name="telefone" id="telefone" placeholder="
        Telefone">
    <button type="submit">Enviar</button>
    </form>
        <ul>
            @foreach ($contatos as $contato)
                <li>
                    {{ $contato->nome }} - {{ $contato->telefone }}
                </li>
            @endforeach
        </ul>
</body>
</html>
```

Listagem 4.50: Conteúdo da página *listar-contatos.blade.php*

Dada a simplicidade do projeto, o banco de dados utilizado neste exemplo será o `SQLite`, pois esse sistema exige poucas configurações para uso. Para configurá-lo, deve-se criar um arquivo chamado *database.sqlite* no diretório *database*. Em seguida, no arquivo de configurações de variáveis de ambiente, localizado em *vendor/.env*, apagam-se as configurações de banco de dados já existentes, mantendo-se apenas a variável `DB_CONNECTION` com o valor `sqlite`. Para criar uma nova tabela no banco de dados, utiliza-se o comando `make:migration` seguido do nome da *migration*, neste exemplo: `create_contatos_table`, conforme apresentado na Listagem 4.51. A execução desse comando gera um arquivo no diretório *database/migrations*, posicionado abaixo dos arquivos gerados automaticamente na inicialização do projeto.

```
php artisan make:migration create_contatos_table
```

Listagem 4.51: Criando nova *migration*

No arquivo gerado pela *migration*, já estão presentes uma coluna `id`, com auto incremento, e uma função `timestamp`, que cria automaticamente as colunas `created_at` e `updated_at`. Utilizando o código apresentado na Listagem 4.52, são adicionadas novas colunas à tabela, correspondentes ao nome e ao telefone do contato. Na Linha 2, define-se um campo do tipo `string` para armazenar o nome,

com limite de 150 caracteres. Na Linha 3, cria-se uma coluna para o número de telefone, também do tipo `string`, com até 20 caracteres e permitindo valores nulos. Para aplicar essas definições ao banco de dados, executa-se o comando `migrate`, conforme mostrado na Listagem 4.53.

```
1 $table->id();
2 $table->string('nome', 150);
3 $table->string('telefone', 20)->nullable();
4 $table->timestamps();
```

Listagem 4.52: Adicionando novas colunas à tabela

```
1 php artisan migrate
```

Listagem 4.53: Executando *migrate*

Para criar um novo modelo no terminal, executa-se o comando apresentado na Listagem 4.54, que gera um arquivo no diretório *app/Models* chamado `Contato.php`. O modelo criado estende a classe base `Model` e permite utilizar os métodos fornecidos pelo *Eloquent* para manipulação do banco de dados. Por convenção, o *Eloquent* associa automaticamente o modelo à tabela do banco de dados com o nome em plural, no caso, `contatos`. Além disso, é necessário importar o modelo no arquivo do *controller*, conforme ilustrado na Listagem 4.55, para que o `Laravel` o reconheça corretamente e o permita utilizar nas operações com o banco de dados.

```
1 php artisan make:model Contato
```

Listagem 4.54: Criando novo modelo

```
1 use App\Models\Contato;
```

Listagem 4.55: Importando modelo

No arquivo *Http/Controllers/ContatosController.php*, após a importação do modelo `Contato`, adiciona-se o conteúdo da Listagem 4.56 ao escopo do método `store`, responsável por salvar novos contatos no banco de dados. Esse método recebe como argumento um objeto de requisição, e nas Linhas 2 e 3 os valores dos campos do formulário são atribuídos, respectivamente, às variáveis `nomeContato` e `telefoneContato`, por meio do método `input()`. Nas Linhas 5 e 6, esses valores são associados às propriedades do modelo `Contato`, e o registro é salvo no banco de dados pelo método `save()`. Por fim, na Linha 9, o código redireciona o usuário de volta à página inicial de exibição dos contatos.

```
1  public function store(Request $request) {
2      $nomeContato = $request->input(key:'nome');
3      $telefoneContato = $request->input(key: 'telefone');
4      $contato = new Contato();
5      $contato->nome = $nomeContato;
6      $contato->telefone = $telefoneContato;
7      $contato->save();
8
9      return redirect(to:'/contatos');
10 }
```

Listagem 4.56: Salvando contatos no banco de dados

A exibição dos contatos, que inicialmente foi implementada de forma estática, agora precisa ser modificada para que os dados sejam obtidos diretamente do banco de dados. Para isso, o código da Listagem 4.57 é inserido no escopo do método `index` do arquivo *Http/Controllers/ContatosController.php*. Na Linha 2, é realizada uma consulta do tipo `SELECT` pelo ORM *Eloquent*, recuperando todos os contatos salvos no banco. Esses dados são, então, repassados à *view* `listar-contatos`. Com isso, o método `index` passa a exibir dinamicamente os contatos armazenados no banco de dados, utilizando a sintaxe fluente e expressiva do *Eloquent*.

```
1  public function index() {
2      $contatos = Contato::all();
3      return view('listar-contatos', ['contatos'=>$contatos]);
4      }
```

Listagem 4.57: Obtendo e exibindo contatos do banco de dados

Para implementar a funcionalidade de edição de contato, algumas alterações devem ser feitas no arquivo *resources/views/listar-contatos.blade.php*, conforme a Listagem 4.58. Na Linha 22 o elemento *li* da lista recebe, além do nome e o telefone do contato, um *link* de redirecionamento para uma nova *view* a ser criada posteriormente.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Lista de Contatos</title>
</head>
<body>

<form action="/contatos" method="POST">
    @csrf
    <input type="text" name='nome' placeholder="Nome">
    <input type="text" name="telefone" id="telefone" placeholder="
        Telefone">
    <button type="submit">Enviar</button>
</form>

<ul>
    @foreach ($contatos as $contato)
    <li>
        {{ $contato->nome }} - {{ $contato->telefone }}
        <a href="/contatos/{{ $contato->id }}/edit">Editar</a>
    </li>
    @endforeach
</ul>
</body>
</html>
```

Listagem 4.58: Adicionando *link* de edição na página inicial

No arquivo do *controller Http/Controllers/ContatosController.php*, no método `edit`, adiciona-se o conteúdo da Listagem 4.59, que recebe como argumento o `id` do contato a ser editado. Em seu escopo, realiza-se a busca do contato correspondente no banco de dados, cujo resultado é armazenado na variável `contato`. O método, então, retorna uma nova *view* para edição. Para isso, deve-se criar, no diretório *resources/views*, um novo arquivo chamado `editar-contato.blade.php`, contendo o código da Listagem 4.60. Esse arquivo apresenta um formulário para edição do contato, utilizando o método HTTP `PUT`.

```
public function edit($id) {
    $contato = Contato::findOrFail($id);
    return view('editar-contato', compact('contato'));
}
```

Listagem 4.59: Buscando contato no banco de dados

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale
        =1.0">
    <title>Editar Contato</title>
</head>
<body>
    <h1>Editar Contato</h1>

    <form action="/contatos/{{ $contato->id }}" method="POST">
        @csrf
        @method('PUT')

        <input type="text" name="nome" value="{{ $contato->nome }}"
            placeholder="Nome">
        <input type="text" name="telefone" value="{{ $contato->
            telefone }}" placeholder="Telefone">
        <button type="submit">Salvar Alterações</button>
    </form>
</body>
</html>
```

Listagem 4.60: Conteúdo da página *editar-contato.blade.php*

No arquivo *Http/Controllers/ContatosController.php*, adiciona-se ao escopo do método `update` o conteúdo da Listagem 4.61, que recebe como argumentos uma requisição e o `id` do contato a ser editado. Em seu escopo, na Linha 2, o contato é localizado no banco de dados com base no `id` fornecido. Em seguida, os dados do contato são atualizados com as informações enviadas pelo formulário de edição. Ao final, as alterações são salvas no banco de dados e o usuário é redirecionado para a página inicial da aplicação.

```
public function update() {
    $contato = Contato::findOrFail($id);

    $contato->nome = $request->input('nome');
    $contato->telefone = $request->input('telefone');
    $contato->save();

    return redirect('/contatos');
}
```

Listagem 4.61: Editando os dados e salvando no banco

Por fim, para implementar a funcionalidade de exclusão de um contato da lista, deve-se adicionar, no arquivo *resources/views/listar-contatos.blade.php*, um formulário que utilize o método HTTP `DELETE`, conforme exemplificado na Linha 24 da Listagem 4.62, que contém um botão para a exclusão do contato. No *controller Http/Controllers/ContatosController.php*, deve-se incluir, no escopo do método `destroy`, o conteúdo da Listagem 4.63. Esse método recebe como argumento o `id` do contato a ser excluído, remove o registro correspondente do banco de dados e, ao final da operação, redireciona o usuário para a página inicial da aplicação.

```
1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale
           =1.0">
7      <title>Lista de Contatos</title>
8  </head>
9  <body>
10     <form action="/contatos" method="POST">
11         @csrf
12         <input type="text" name='nome' placeholder="Nome">
13         <input type="text" name="telefone" id="telefone" placeholder="
               Telefone">
14         <button type="submit">Enviar</button>
15     </form>
16     <ul>
17         @foreach ($contatos as $contato)
18         <li>
19             {{ $contato->nome }} - {{ $contato->telefone }}
20             <a href="/contatos/{{ $contato->id }}/edit">Editar</a>
21         </li>
22         <form action="/contatos/{{ $contato->id }}" method="POST"
               style="display:inline;">
23             @csrf
24             @method('DELETE')
25             <button type="submit" onclick="return confirm('Tem certeza
                   que deseja excluir este contato?')">Excluir</button>
26         </form>
27         @endforeach
28     </ul>
29 </body>
30 </html>
```

Listagem 4.62: Adicionando formulário de exclusão na página inicial

```php
public function destroy($id) {
    $contato = Contato::findOrFail($id);
    $contato->delete();

    return redirect('/contatos');
}
```

Listagem 4.63: Método de exclusão de contato

Ao final do processo de desenvolvimento da aplicação, a página inicial resultante é apresentada na Figura 4.14. Para adicionar estilização à interface, cria-se um novo arquivo em *public/css/style.css* com o conteúdo da Listagem 4.64. Em seguida, é necessário adicionar, no escopo da tag `<head>` da página inicial, o código da Listagem 4.65 para referenciar o arquivo de estilos. A estilização de outras páginas da aplicação pode ser feita de forma análoga. Com isso, o Sistema de Gerenciamento de Contatos adquire a aparência exibida na Figura 4.15.

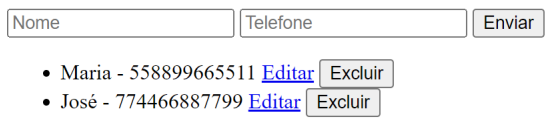


Figura 4.14: Fragmento da Página inicial da aplicação

```css
body {
    background-color: #c4e6db;
}
```

Listagem 4.64: Estilização da página inicial

```php
<link rel="stylesheet" href="{{ asset('css/style.css') }}">
```

Listagem 4.65: Referenciando arquivo CSS na página inicial



Figura 4.15: Fragmento da Página inicial da aplicação estilizada

4.8 Visão geral do Framework Ruby on Rails

O `Ruby on Rails`[16] (doravante, `Rails`) é um *framework* web gratuito e de código aberto, escrito na linguagem dinâmica e orientada a objetos `Ruby`[17]. Com mais de seis mil colaboradores contribuindo com seu código, o `Rails` foi projetado para reduzir a complexidade do desenvolvimento de aplicações web modernas (RAILS, 2024), tornando-as mais fáceis de criar, implantar e manter (RUBY; THOMAS, 2023), além de sustentar sites de alto volume de tráfego (VISWANATHAN, 2008). Lançado em sua primeira versão em 2004, o `Rails` evoluiu e passou a contar com um vasto conjunto de bibliotecas associadas, formando o que se conhece como Ecossistema `Rails` (RUBY; THOMAS, 2023). Trata-se de um *framework full-stack*, oferecendo ferramentas para o desenvolvimento completo de aplicações web, tanto no *front-end* quanto no *back-end*, incluindo renderização de *templates* HTML, gerenciamento de banco de dados, enfileiramento de *jobs* assíncronos e mecanismos robustos de proteção contra ataques comuns (RAILS, 2024).

Além disso, o `Rails` é reconhecido como uma estrutura promissora para o desenvolvimento ágil de aplicações web, fundamentada em seis princípios: simplicidade, capacidade de expansão, testabilidade, produtividade, reutilização e manutenção. A simplicidade está relacionada à funcionalidade de *Scaffolding*, que gera automaticamente a estrutura básica da aplicação com padrões de configuração predefinidos, reduzindo a complexidade do desenvolvimento e economizando tempo (MICHAEL et al., 2007). A capacidade de expansão permite que o projeto seja escalado, possibilitando ao desenvolvedor aprofundar-se em aspectos específicos para tornar a aplicação mais robusta. A testabilidade refere-se ao suporte nativo à criação de testes automatizados, que facilitam a identificação e correção de erros no código. A produtividade é impulsionada pela facilidade de iniciar tarefas rapidamente, algo que o criador do *framework*, David Heinemeier Hansson, descreve como "compressão conceitual" (RUBY; THOMAS, 2023). Já a reutilização e a capacidade de manutenção do código baseiam-se no princípio DRY (*Don't Repeat Yourself*) (MICHAEL et al., 2007), promovendo um *software* mais confiável, compreensível e fácil de manter (THOMAS; HUNT, 2019).

Uma vez que o `Ruby` esteja instalado, para instalar o `Rails`, deve-se executar no terminal o comando apresentado na Listagem 4.66, que instala o *framework* por meio do gerenciador de pacotes *RubyGems*. Esse gerenciador facilita a criação, o compartilhamento e a instalação de bibliotecas, sendo instalado por padrão junto com o `Ruby` (RUBY, 2024). Para verificar se a instalação foi concluída com sucesso, pode-se executar o comando da Listagem 4.67, cujo retorno esperado é o nome do *framework* seguido da versão instalada, conforme exemplificado na Linha 2.

[16] ⟨https://rubyonrails.org/⟩
[17] ⟨https://www.ruby-lang.org/pt/⟩

```
gem install rails
```

Listagem 4.66: Comando de instalação

```
rails --version
% Rails 7.1.0
```

Listagem 4.67: Comando de verificação da instalação

4.9 Arquitetura do Ruby on Rails

O *framework* `Rails` adota o padrão arquitetural MVC (*Model*, *View*, *Controller*), que organiza a aplicação em três camadas interconectadas (VERMA, 2014). Essas camadas abrangem a interface de usuário, as funcionalidades e o conteúdo informativo (PRESSMAN; MAXIM, 2021). A camada *Model* é responsável por lidar com os dados da aplicação, bem como pelas regras de negócio e pela lógica de processamento. A camada *View* fornece a interface de usuário, permitindo a apresentação do conteúdo e da lógica da aplicação. Já a camada *Controller* atua como intermediária entre as camadas *Model* e *View*, coordenando o fluxo de dados entre elas e gerenciando as requisições do usuário (PRESSMAN; MAXIM, 2021).

No `Rails`, as duas principais subestruturas que compõem sua arquitetura são o *Action Pack* e o *Active Record*, os quais estabelecem a base do padrão MVC utilizado pelo *framework*. O *Action Pack* é responsável pelas funcionalidades relacionadas ao *Controller* e à *View*, sendo subdividido em três módulos: *Action Dispatch*, *Action Controller* e *Action View* (RUBY; THOMAS, 2023).

A Figura 4.16 ilustra a arquitetura do *framework* `Rails`, destacando, em azul, os submódulos que compõem o *Action Pack*. Nessa arquitetura, a requisição é iniciada pelo usuário e enviada ao servidor web, que a encaminha ao *Action Dispatcher*, responsável por rotear a requisição para o *controller* apropriado. No *Action Controller*, a requisição pode envolver operações com o *Active Model* e, posteriormente, com o *Active Record*. A consulta é então realizada ao banco de dados, que retorna os dados solicitados ou mensagens de erro, caso não sejam encontrados. A resposta é enviada de volta ao *Action Controller*, que passa a ser responsável por preparar os dados para renderização, encaminhando-os ao *Action View*. Dentro do *Action Controller*, também é possível redirecionar a requisição para outra URL, como ocorre, por exemplo, quando um usuário é direcionado a uma nova página após o *login*. Por fim, no *Action View*, a página é exibida ao usuário, com os dados processados da requisição.

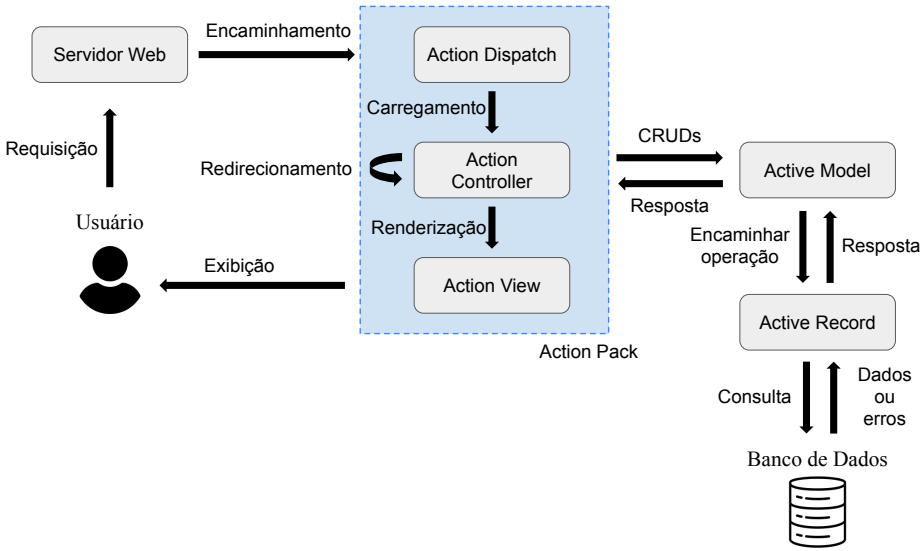


Figura 4.16: Arquitetura `Ruby on Rails`. Adaptado de (PATEL, 2023b)

4.9.1 Action Pack

O módulo *Action Pack* abrange as camadas *Controller* e *View* da arquitetura MVC, sendo subdividido nos módulos *Action Dispatch*, *Action Controller* e *Action View*, que fornecem suporte ao processamento de solicitações e à geração de respostas. Embora esses módulos estejam agrupados em um único componente, o `Rails` promove uma separação clara entre o controle da aplicação e a lógica de apresentação, facilitando a organização e a manutenção do código (RUBY; THOMAS, 2023).

O servidor web padrão do `Rails`, denominado *Puma*, permite abrir uma janela do navegador e acessar o endereço `http://localhost:3000` para visualizar e navegar pelas páginas da aplicação. Um dos benefícios desse servidor é que ele não precisa ser reiniciado após alterações nos arquivos do projeto, pois as mudanças são detectadas automaticamente e refletidas em tempo real na interface da aplicação.

O *Action Dispatch* é responsável pelo roteamento das requisições do usuário ao servidor web. Ele realiza o processamento relacionado ao protocolo HTTP, incluindo a decodificação de parâmetros, a manipulação de lógica de *cache*, o gerenciamento de *cookies* e de sessões. O roteamento diz respeito às rotas da aplicação, que definem o caminho percorrido pelo usuário, determinando quais páginas serão exibidas e quais ações serão executadas com base nas requisições recebidas.

O `Rails` permite duas abordagens para a definição de rotas: uma abordagem abrangente e outra conveniente, sendo possível ainda combiná-las conforme a necessidade da aplicação. A forma abrangente possibilita o mapeamento direto de URLs, com base em correspondências explícitas, permitindo definir rotas específicas para

diferentes caminhos utilizando expressões regulares (RAILS, 2024). A Listagem 4.68 apresenta um exemplo dessa abordagem, com a criação de rotas para manipulação de *posts* de um *blog*, em que o *id* é passado como parâmetro na requisição. Na Linha 2, a rota com o método `get` exibe um *post* na tela; na Linha 3, a rota permite a edição do conteúdo; e, na Linha 4, a rota com o método `delete` exclui o *post*.

Por outro lado, a forma conveniente permite definir rotas com base em modelos, gerando automaticamente as rotas necessárias para as *actions* padrão: `index`, `show`, `new`, `edit`, `create`, `update` e `destroy` (RAILS, 2024). Utilizando o mesmo exemplo de *posts* de um *blog*, a Listagem 4.69 demonstra como o próprio `Rails` é capaz de gerar todas essas rotas de forma automática. Após o roteamento ser processado, o *Action Dispatch* encaminha a requisição ao controlador correspondente, que dará continuidade ao fluxo da aplicação.

```
1 Rails.application.routes.draw do
2   get '/posts/:id', to: 'posts#show', as: 'post'
3   get '/posts/:id/edit', to: 'posts#edit', as: 'edit_post'
4   delete '/posts/:id', to: 'posts#destroy'
5 end
```

Listagem 4.68: Exemplo de definição de rotas da maneira abrangente

```
1 Rails.application.routes.draw do
2   resources :posts
3 end
```

Listagem 4.69: Exemplo de definição de rotas da maneira conveniente

O *Action Controller* é responsável por receber as solicitações encaminhadas pelo *Action Dispatch* e retornar uma visualização ao *Action View*. Cada controlador trata uma ação ou objeto específico e envia a resposta correspondente, que será exibida no navegador. Para cada controlador, existe um diretório associado ao diretório *app/views*, contendo os arquivos de *template* que serão renderizados na tela conforme as ações definidas no respectivo controlador (RAILS, 2024).

Além disso, o *Action Controller* permite a execução de métodos adicionais por meio de filtros, sendo os principais o `before_action` e o `after_action`. O `before_action` (pré-filtro) executa um método antes da execução da ação propriamente dita, enquanto o `after_action` (pós-filtro) executa um método após a conclusão da ação (MICHAEL et al., 2007). A Listagem 4.70 apresenta um exemplo de uso do `before_action` para verificar a autenticação do usuário, utilizando o método `authenticate_user!`. Caso o usuário esteja autenticado, a ação de exibição do perfil, demonstrada na Linha 4, será executada. Já a Listagem 4.71 apresenta um exemplo de aplicação do `after_action`, que registra a hora em que determinada ação foi executada. Na Linha 12, a hora atual é registrada após a execução da ação `create`.

```ruby
class ProfilesController < ApplicationController
  before_action :authenticate_user!

  def show
    @user = current_user
  end
end
```

Listagem 4.70: Exemplo de uso de *before_action*

```ruby
class PostsController < ApplicationController
  after_action :log_action_time, only: [:create]

  def create
    # Lógica da action
  end

  private

  def log_action_time
    Rails.logger.info "Ação 'create' foi executada em #{Time.now}"
  end
end
```

Listagem 4.71: Exemplo de uso de *after_action*

Após o processamento das ações, o *Action Controller* estabelece a conexão com o *Active Record* e obtém os dados necessários do banco de dados para a renderização da página solicitada pelo usuário. A exibição da resposta é realizada pelo submódulo *Action View*, responsável por exibir no navegador o conteúdo resultante da requisição. A saída final exibida ao usuário é composta por três elementos principais: *templates*, *partials* e *layouts*.

Os *templates* são arquivos responsáveis por estruturar a exibição da interface da aplicação, permitindo a escrita de código `Ruby` no HTML. Além disso, oferecem suporte à linguagem de marcação XML (RAILS, 2024). A Listagem 4.72 apresenta um exemplo de *template* que utiliza um *loop* para exibir uma lista de nomes, utilizando as *tags* de incorporação padrão (`<% %>`), que são empregadas para executar código `Ruby` sem gerar saída direta na tela.

```
<h1>Nomes</h1>
<% @pessoas.each do |pessoa| %>
  Nome: <%= pessoa.nome %><br>
<% end %>
```

Listagem 4.72: Exemplo de *template*

Já os *partials* são trechos de código reutilizáveis que podem ser renderizados em uma *view*, com o objetivo de evitar duplicação e promover a organização do código. Esses arquivos podem ser reutilizados múltiplas vezes na aplicação e são nomeados com um prefixo sublinhado (_), conforme especificado pela convenção do `Rails` (RAILS, 2024). A Listagem 4.73 apresenta um exemplo de *partial* que define uma estrutura para a exibição de *posts*, incluindo o título e o corpo da publicação. Já a Listagem 4.74 ilustra como renderizar um *partial* em uma *view*.

```
    <div class="post">
      <h2><%= post.title %></h2>
      <p><%= post.body %></p>
    </div>
```

Listagem 4.73: Exemplo de um arquivo *partial*

```
<h1>Posts</h1>
<%= render @post %>
```

Listagem 4.74: Exemplo de renderização de um *partial*

Por sua vez, os *layouts* são utilizados para renderizar *templates* comuns a várias *views* (RAILS, 2024), como, por exemplo, a estrutura padrão do HTML que envolve elementos como cabeçalhos, rodapés e barras de navegação. A Listagem 4.75 apresenta um exemplo de *layout*, em que, na Linha 11, o método `yield` indica o ponto onde o conteúdo da *view* específica será inserido. Dessa forma, ao combinar os três elementos: *template*, *partial* e *layout* é possível compor uma *view* completa, cujo conteúdo é preenchido dinamicamente em tempo de execução.

```
<!DOCTYPE html>
<html>
<head>
  <title>Minha página</title>
  <%= csrf_meta_tags %>
  <%= csp_meta_tag %>
  <%= stylesheet_link_tag 'application', media: 'all', 'data-
      turbolinks-track': 'reload' %>
  <%= javascript_include_tag 'application', 'data-turbolinks-track': '
      reload' %>
</head>
<body>
  <%= yield %>
</body>
</html>
```

Listagem 4.75: Exemplo de renderização de um *layout*

Dessa forma, para evitar sobrecarregar os *templates* com código `Ruby`, o `Rails` disponibiliza classes auxiliares chamadas de *helpers*, localizadas no diretório *app/helpers*. Esses *helpers* são responsáveis por realizar tarefas comuns, como a criação de formulários, a formatação de datas e a manipulação de *strings* (RAILS, 2024). A Listagem 4.76 apresenta um exemplo de arquivo *helper* chamado `uppercase`, que converte *strings* em letras maiúsculas. Já a Listagem 4.77 mostra como renderizar esse *helper* em uma *view*, na qual, na Linha 2, o resultado esperado é exibido na tela.

```
module StringHelpers
  def uppercase(string)
    string.upcase
  end
end
```

Listagem 4.76: Exemplo de um arquivo *helper*

```
<%= uppercase("hello world") %>
# HELLO WORLD
```

Listagem 4.77: Exemplo de renderização de um *helper*

4.9.2 Active Record

O *Active Record* representa a camada *Model* da arquitetura MVC e é responsável por conter a lógica de negócios da aplicação, conectando as tabelas do banco de dados às suas representações em classes `Ruby` (RAILS, 2024). Esse módulo transforma as requisições provenientes do *Action Controller* em comandos SQL (MICHAEL et al., 2007). O *Active Record* segue o padrão ORM (Object-Relational Mapping),

no qual cada classe está diretamente associada a uma tabela do banco de dados; cada registro corresponde a um objeto dessa classe, e cada coluna é representada por um atributo (MICHAEL et al., 2007). Essa estrutura permite representar modelos e seus dados de forma orientada a objetos, além de possibilitar a execução de operações no banco de dados de maneira simples e intuitiva (RAILS, 2024).

Como parte de sua filosofia, o `Rails` adota o conceito de convenções em vez de configurações, conforme apresentado em sua documentação oficial. Ao seguir essas convenções, espera-se que o desenvolvedor precise escrever pouca ou nenhuma configuração adicional. Um exemplo clássico dessa abordagem é a pluralização automática dos nomes das classes para associá-las às respectivas tabelas do banco de dados. Por exemplo, a classe `User` será automaticamente associada à tabela `users`. Para nomes compostos, utiliza-se a convenção *CamelCase*, de modo que a classe `BookClub` corresponderá à tabela `book_clubs`. No entanto, também é possível realizar associações manuais. Por exemplo, se houver uma classe chamada `Usuario` e a tabela correspondente for `usuarios`, o `Rails` não fará essa associação automaticamente, sendo necessário especificá-la manualmente por meio de configuração (RAILS, 2024).

O acrônimo CRUD refere-se às quatro operações básicas realizadas sobre dados: *Create*, *Read*, *Update* e *Delete*. Nativamente, o `Rails` oferece suporte à implementação dessas funcionalidades por meio do módulo *Active Record*, sem a necessidade de configurações adicionais. Esse módulo cria automaticamente os métodos necessários para a leitura e a manipulação de registros no banco de dados. Além disso, o *Active Record* permite validar o estado de um modelo antes de persistir os dados, por meio de métodos como `create` e `update`, que retornam `false` caso os dados sejam inválidos, impedindo, assim, que operações incorretas sejam executadas no banco (RAILS, 2024).

4.9.3 Vantagens do Ruby on Rails

Banco de dados: o `Rails` permite a conexão com diferentes sistemas de gerenciamento de banco de dados, entre os quais estão `SQLite3`, `DB2`, `MySQL`, `PostgreSQL` e `SQL Server`, sendo necessária, com exceção do `SQLite3`, a instalação de um *driver* específico para cada banco (RUBY; THOMAS, 2023). Para cada projeto, é possível definir diferentes bancos de dados conforme o ambiente de execução, como, por exemplo, um banco de dados para desenvolvimento, outro para testes e um terceiro para produção (MICHAEL et al., 2007).

Convenção sobre configuração: o `Rails` adota um conjunto de convenções nativas que visam reduzir a necessidade de configurações manuais, aumentando a produtividade do desenvolvedor. Essas convenções não apenas simplificam o desenvolvimento, mas também facilitam o processo de aprendizagem, permitindo que iniciantes criem aplicações simples mesmo com pouca experiência prévia (RAILS, 2024).

4.10 Exemplo prático com Ruby on Rails

O exemplo apresentado nesta seção consiste em uma aplicação denominada "*Blog Simples*", desenvolvida com o *framework* `Rails`. Nessa aplicação, é possível adicionar *posts* com título e conteúdo, listá-los na página principal, bem como editar e excluir individualmente cada *post*. Com o uso de um banco de dados, as informações permanecem armazenadas de forma persistente, mesmo após o recarregamento da página. As Seções 4.10.1 e 4.10.2 descrevem detalhadamente o processo de desenvolvimento da aplicação.

4.10.1 Preparação do ambiente

Após a instalação do `Rails`, deve-se instalar o banco de dados `SQLite3`, que será utilizado no desenvolvimento do projeto. Por padrão, o `Rails` disponibiliza *scripts* projetados para facilitar a criação de aplicações, denominados geradores (RAILS, 2024). Assim, ao iniciar um novo projeto, o uso de um gerador de aplicações fornece uma base estruturada, reduzindo a necessidade de configurações manuais. Para isso, no terminal, deve-se navegar até o diretório desejado e executar o comando apresentado na Listagem 4.78, que cria um novo projeto. No exemplo apresentado, o projeto é denominado *Blog*.

```
1 rails new blog
```

Listagem 4.78: Inicializando novo projeto

O novo diretório criado para o projeto contém diversas pastas e arquivos gerados automaticamente, que compõem a estrutura da aplicação. Cada elemento desempenha uma função específica no funcionamento do projeto. As principais pastas e arquivos são:

- *app*: contém as camadas da arquitetura MVC;
- *bin*: reúne os *scripts* de inicialização da aplicação, bem como outros *scripts* utilizados para configurar e atualizar o projeto;
- *config*: armazena as configurações da aplicação, incluindo as rotas;
- *db*: contém o esquema do banco de dados e os arquivos de *migrations*;
- *public*: responsável por armazenar os arquivos estáticos;
- *README.md*: arquivo de instruções e informações gerais sobre o uso da aplicação.

Após a criação do novo projeto, deve-se navegar até o diretório principal no terminal e executar o comando apresentado na Listagem 4.79 para inicializar o servidor web. Em seguida, ao acessar o endereço `http://localhost:3000` no

navegador, é possível visualizar a página inicial do projeto, conforme ilustrado na Figura 4.17. Essa página inicial, em uma nova aplicação `Rails`, é conhecida como *smoke test* (teste de fumaça) e tem a finalidade de garantir que o *software* foi configurado corretamente e está apto a exibir uma página básica (RAILS, 2024).

```
1 ruby bin\rails server
```

Listagem 4.79: Inicializando o servidor



Figura 4.17: Página Inicial do `Ruby on Rails`

4.10.2 Implementação do exemplo

Inicialmente, é criada uma rota no arquivo *config/routes.rb*, que mapeia as requisições do tipo `GET /posts` para a ação `index` de um controlador, conforme apresentado na Listagem 4.80. Em seguida, para criar o *controller*, utiliza-se, no terminal, o comando apresentado na Listagem 4.81, que gera um controlador chamado `PostsController` juntamente com sua respectiva ação `index`.

```
Rails.application.routes.draw do
  get "/posts", to: "posts#index"
end
```

Listagem 4.80: Criando uma nova rota

```
rails generate controller Posts index
```

Listagem 4.81: Criando um novo *controller*

Após a execução do comando, o `Rails` gera vários arquivos, e o primeiro a ser editado é o *app/controllers/posts_controller.rb*, que possui, em seu código, uma ação *index* de escopo, conforme a Listagem 4.82. Quando uma ação não renderiza explicitamente uma *view*, o `Rails`, por convenção de configuração, renderizará uma *view* correspondente ao nome do *controller* e da ação (RAILS, 2024).

```
class PostsController < ApplicationController
  def index
  end
end
```

Listagem 4.82: Exemplo de arquivo *PostsController*

As *views* estão localizadas no diretório *app/views*, e a ação *index* é renderizada no arquivo *app/views/posts/index.html.erb* que contém o conteúdo apresentado na Listagem 4.83. Posteriormente, o conteúdo padrão exibido na página será substituído pelo conteúdo principal da aplicação.

```
   <h1>Posts#index</h1>
    <p>Find me in app/views/posts/index.html.erb</p>
```

Listagem 4.83: Exemplo de conteúdo do arquivo *index*

Para que a página *index* criada a partir da rota *posts* seja exibida como página inicial, é necessário adicionar essa rota como raiz do projeto, incluindo, no arquivo *config/routes.rb*, a rota *root* no topo do bloco *Rails.application.routes.draw*, conforme apresentado na Linha 2 da Listagem 4.84. A partir disso, a nova página inicial da aplicação deverá conter o fragmento apresentado na Figura 4.18 com um texto padrão escrito pelo próprio `Rails`.

```
Rails.application.routes.draw do
  root "posts#index"

  get "/posts", to: "posts#index"
end
```

Listagem 4.84: Adicionando a rota *root*



Figura 4.18: Fragmento da página inicial do projeto após a adição da rota *root*

Após a definição do *controller* e da *view*, cria-se o *model* da aplicação, seguindo o padrão arquitetural MVC. Em `Rails` os modelos são representados como uma classe e, por convenção, a nomenclatura dos modelos é no singular, pois o modelo instanciado representa um único registro de dados (RAILS, 2024). Um novo modelo pode ser definido executando, no terminal, a linha de comando da Listagem 4.85, que cria dois campos, *string* e *text*, respectivamente, para o título e o corpo do *post*.

```
1 rails generate model Post title:string body:text
```

Listagem 4.85: Criando modelo *Post*

As *migrations* são um recurso que permite registrar modificações no esquema do banco de dados ao longo do tempo, de forma consistente, utilizando uma DSL em `Ruby`, em vez de comandos SQL puros. A DSL (*Domain-Specific Language*) é uma linguagem desenvolvida dentro do próprio `Ruby` para descrever as alterações realizadas no banco de dados, sendo que cada *migration* pode ser interpretada como uma nova "versão" do banco, possibilitando a reversão para qualquer ponto do histórico de alterações. O arquivo de migração *db/migrate/¡timestamp¿_create_posts.rb* contém o código responsável por criar, no banco de dados, a tabela `posts` com os campos desejados. Para que a migração seja aplicada, é necessário executar o comando apresentado na Listagem 4.86, que exibe no terminal uma mensagem de confirmação da criação da tabela a partir da *migration*.

```
1 rails db:migrate
```

Listagem 4.86: Executando migração

Para o presente projeto, são implementadas as operações CRUD, sendo que a primeira operação a ser desenvolvida é a *Read*, correspondente à leitura dos dados. Essa funcionalidade é incorporada por meio de uma modificação na ação `index`, localizada no arquivo *app/controllers/posts_controller.rb*, conforme mostrado na Listagem 4.87. Na Linha 3, a variável de instância `@posts` é declarada — o símbolo @ indica que a variável estará disponível na *view* — e recebe o resultado da chamada ao modelo `Post` por meio do método `all`, que retorna todos os registros da tabela associada.

```
class PostsController < ApplicationController
  def index
    @posts = Post.all
  end
end
```

Listagem 4.87: Adicionando método *index*

As variáveis de instância definidas no *controller* podem ser acessadas pela *view*, localizada no arquivo *app/views/posts/index.html.erb*, substituindo-se todo o conteúdo desse arquivo pelo código HTML apresentado na Listagem 4.88. Na Linha 4, a variável de instância `@posts` é utilizada para exibir, em tela, todos os *posts* armazenados no banco de dados. O código HTML apresenta ainda, na Linha 1, o título da página; na Linha 3, inicia-se uma lista não ordenada que percorre e exibe o título de cada *post*. As marcações `<% %>` e `<%= %>` são *tags* regulares utilizadas para incorporar código `Ruby` ao HTML.

```
<h1>Blog</h1>

<ul>
  |<% @posts.each do |post| %>|
    <li>
      |<%= post.title %>|
    </li>
  |<% end %>|
</ul>
```

Listagem 4.88: Conteúdo da página de exibição dos *posts*

Como boa prática e com base no princípio DRY (*Don't Repeat Yourself*), recomenda-se a criação de uma *view* compartilhada, denominada *partial*, que contém a estrutura comum de uma página HTML. Para isso, cria-se o arquivo *app/views/posts/_page_structure.html.erb* com o conteúdo apresentado na Listagem 4.89, em que, na Linha 11, é inserido dinamicamente o conteúdo da página. Dessa forma, todas as páginas da aplicação poderão reutilizar esse *partial*, que é inicialmente aplicado na página *app/views/posts/index.html.erb*, conforme mostrado na Listagem 4.90.

```
<html>
<head>
  <title>"Blog"</title>
  |<%=| stylesheet_link_tag 'application', media: 'all' |%>|
  |<%=| javascript_pack_tag 'application' |%>|
  |<%=| csrf_meta_tags |%>|
</head>
<body>
  <div class="container">
    |<%=| yield |%>|
  </div>
</body>
</html>
```

Listagem 4.89: *Partial* da estrutura comum das páginas HTML

```
|<%=| render "page_structure" |%>|
<h1>Blog</h1>

<ul>
  |<% @posts.each do |post| %>|
    <li>
      |<%= post.title %>|
    </li>
  |<% end %>|
</ul>
```

Listagem 4.90: Conteúdo da página de exibição dos *posts* utilizando *partial*

Antes de implementar as demais operações CRUD, é recomendável criar uma *view* para a visualização individual de cada *post*. Esse processo se inicia com a adição de uma nova rota no arquivo *config/routes.rb*, que mapeia a ação correspondente no *controller*, conforme exemplificado na Linha 5 da Listagem 4.91. A nova rota inclui um parâmetro `:id`, que permite identificar e buscar no banco de dados o *post* correspondente.

```
Rails.application.routes.draw do
  root "posts#index"

  get "/posts", to: "posts#index"
  get "/posts/:id", to: "posts#show"
end
```

Listagem 4.91: Criando nova rota *show*

Essa busca pode ser implementada por meio da inclusão de uma nova ação no arquivo *app/controllers/posts_controller.rb*, como exemplificado pela ação `show` na Listagem 4.92. Nessa ação, utiliza-se o método `find`, que recebe o parâmetro `:id` e retorna o *post* correspondente, atribuindo-o à variável de instância `@post`. Uma vez

localizadas as informações do *post*, é necessário estruturá-las para exibição em tela. Isso pode ser feito por meio da criação do arquivo *app/views/posts/show.html.erb*, cujo conteúdo é apresentado na Listagem 4.93, que exibe o título e o conteúdo do *post*, além de um *link* para retornar à página inicial. Por fim, para permitir o redirecionamento da página inicial para a página de exibição individual do *post*, é necessário modificar o arquivo *app/views/posts/index.html.erb*, adicionando a tag `<a>`, conforme mostrado na Listagem 4.94.

```
    class PostsController < ApplicationController
      def index
        @posts = Post.all
      end
      def show
        @post = Post.find(params[:id])
      end
    end
```

Listagem 4.92: Criando nova ação *show* no *controller*

```
|<%=| render "page_structure" |%>|
<h1>|<%=| @post.title |%>|</h1>

<p>|<%=| @post.body |%>|</p>

<ul>
  <li>
    <a href="/posts">
        Voltar para página inicial
    </a>
  </li>
</ul>
```

Listagem 4.93: Criando página de exibição de *post*

```
|<%| content_for :title, "Blog" |%>|

|<%=| render "page_structure" |%>|
<h1>Blog</h1>

<ul>
  |<% @posts.each do |post| %>|
    <li>
        <a href="/posts/<%= post.id %>">
            |<%= post.title %>|
        </a>
    </li>
  |<% end %>|
</ul>
```

Listagem 4.94: Adicionando *link* aos títulos dos *posts*

Por padrão, em aplicações `Rails`, as entidades que integram rotas, ações de *controller* e *views* para a realização de operações CRUD são denominadas recursos. Nesta aplicação, os *posts* são considerados um recurso, e para lidar com eles, o `Rails` fornece o método `resources`, que mapeia automaticamente todas as rotas convencionais associadas a um recurso. Isso permite substituir declarações manuais no arquivo *config/routes.rb*, conforme ilustrado na Listagem 4.95. A inspeção das rotas geradas pode ser realizada pelo comando `rails routes`, cujo retorno é exemplificado na Listagem 4.96.

```
1 Rails.application.routes.draw do
2     root "posts#index"
3
4     resources :posts
5 end
```

Listagem 4.95: Criando o método *resources*

```
1  rails routes
2  #     Prefix Verb   URI Pattern                  Controller#Action
3  #       root GET    /                            posts#index
4  #      posts GET    /posts(.:format)             posts#index
5  #   new_post GET    /posts/new(.:format)         posts#new
6  #       post GET    /posts/:id(.:format)         posts#show
7  #            POST   /posts(.:format)             posts#create
8  #  edit_post GET    /posts/:id/edit(.:format)    posts#edit
9  #            PATCH  /posts/:id(.:format)         posts#update
10 #            DELETE /posts/:id(.:format)         posts#destroy
```

Listagem 4.96: Mapeando rotas criadas pelo método *resources*

Após a substituição no arquivo *config/routes.rb*, é necessário também alterar o arquivo *app/views/posts/index.html.erb*, conforme mostrado na Listagem 4.97. Essa alteração inclui a utilização do *helper* `link_to` (Linha 7), responsável por renderizar um *link* na página. Esse *helper* recebe dois argumentos principais: o primeiro define o texto a ser exibido no *link*, e o segundo indica o destino para o qual o *link* redirecionará.

```
1  |<%=| render "page_structure" |%>|
2 <h1>Blog</h1>
3
4 <ul>
5   |<% @posts.each do |post| %>|
6     <li>
7         |<%=| link_to post.title, post |%>|
8     </li>
9   |<% end %>|
10 </ul>
```

Listagem 4.97: Adicionando o *helper link_to*

A seguir, a implementação da operação *Create*, do conjunto CRUD, é realizada em etapas. Inicialmente, conforme apresentado na Listagem 4.98, no arquivo *app/controllers/posts_controller.rb*, abaixo da ação `show`, implementa-se a ação `new`, que instancia um novo *post*, porém sem salvá-lo no banco de dados. O salvamento é responsabilidade da ação `create`, definida a seguir. Na Linha 13, é feita a tentativa de salvar o *post*. Caso a operação seja bem-sucedida, a aplicação redireciona o navegador para a página inicial, exibindo o novo *post*, conforme a Linha 14. Se o salvamento falhar, a ação renderiza novamente o formulário, retornando um código de *status* 422, como demonstrado na Linha 16.

```
1  class PostsController < ApplicationController
2    def index
3      @post = Post.all
4    end
5    def show
6      @post = Post.find(params[:id])
7    end
8    def new
9      @post = Post.new
10   end
11   def create
12     @post = Post.new(title: "...", body: "...")
13     if @post.save
14       redirect_to root_path
15     else
16       render :new, status: :unprocessable_entity
17     end
18   end
19 end
```

Listagem 4.98: Criando ações *new* e *create* no *controller*

A etapa de criação de um novo *post* envolve, ainda, um formulário para obtenção dos dados, que pode ser criado utilizando um recurso chamado construtor de formulário (*form builder*), que permite construir um formulário totalmente configurado com uma quantidade mínima de código, conforme exemplifica a Listagem 4.99, que constitui o conteúdo do novo arquivo *app/views/posts/new.html.erb*, no qual o método auxiliar *form_with*, da Linha 4, instancia um construtor de formulário e os métodos *label*, *text_field* e *text_area* são usados para gerar elementos apropriados para o formulário.

```
1  |<%=| render "page_structure" |%>|
2  <h1>Novo Post</h1>
3
4  |<%=| form_with model: @post do |form| |%>|
5    <div>
6      |<%=| form.label "Título" |%>|<br>
7      |<%=| form.text_field :title |%>|
8    </div>
9
10   <div>
11     |<%=| form.label "Conteúdo" |%>|<br>
12     |<%=| form.text_area :body |%>|
13   </div>
14
15   <div>
16     |<%=| form.submit "Salvar" |%>|
17   </div>
18 |<% end %>|
```

Listagem 4.99: Formulário de criação de novo *post*

Os dados obtidos pelo formulário são enviados para a ação `create` como parâmetros de *hash*, juntamente com os parâmetros de rota. Esses dados podem ser acessados individualmente, como no caso do título, por `params[:post][:title]`, e do corpo, por `params[:post][:body]`. No entanto, esse tipo de acesso individual aos valores pode ser suscetível a erros, tornando-o vulnerável a modificações indevidas. Para mitigar esse risco, utiliza-se um único *hash* que contém os valores e um filtro conhecido como *Strong Parameters*, que permite filtrar os parâmetros e evitar que um usuário mal-intencionado envie campos extras ou substitua dados privados. Para implementar o *Strong Parameters*, é necessário adicionar um método privado no final do escopo da classe, no arquivo *app/controllers/posts_controller.rb*, conforme apresentado na Listagem 4.100. Além disso, na Linha 11, no método `create`, o argumento é alterado para o método privado `post_params`.

```ruby
class PostsController < ApplicationController
  def index
    @post = Post.all
  end
  def show
    @post = Post.find(params[:id])
  end
  def new
    @post = Post.new
  end
  def create
    @post = Post.new(post_params)
    if @post.save
      redirect_to root_path
    else
      render :new, status: :unprocessable_entity
    end
  end

  private
    def post_params
      params.require(:post).permit(:title, :body)
    end
end
```

Listagem 4.100: Adicionando *Strong Parameters*

Outra etapa do processo de implementação do método `Create` é a validação dos dados de entrada fornecidos pelo usuário. Para isso, o `Rails` oferece um recurso chamado *validations*, que verifica o objeto do modelo antes de salvar os dados no banco de dados. Caso a validação falhe, o salvamento é abortado e mensagens de erro correspondentes são exibidas ao usuário. A implementação desse recurso, por meio do arquivo *app/models/post.rb*, é ilustrada na Listagem 4.101. Na Linha 2, valida-se a variável `title`, exigindo pelo menos um caractere. De forma análoga, na Linha 3, a validação exige que o campo `body` contenha pelo menos 10 caracteres.

```ruby
    class Post < ApplicationRecord
        validates :title, presence: { message: "O post deve conter um
            título" }
        validates :body, presence: { message: "O conteúdo do post deve
             conter no mínimo 10 caracteres" }, length: { minimum: 10,
             message: "O conteúdo do post deve conter no mínimo 10
            caracteres" }
    end
```

Listagem 4.101: Adicionando *Validations*

Naturalmente, para que os testes de validação tenham efeito visual, o arquivo *app/views/posts/new.html.erb* também precisa ser modificado para exibir mensagens de erro para os campos *title* e *body*, conforme demonstrado nas Linhas 8 e 16

da Listagem 4.102. Isso é feito pelo método `full_messages_for`, que retorna a mensagem de erro correspondente.

```
1  |<%=| render "page_structure" |%>|
2  <h1>Novo Post</h1>
3
4  |<%=| form_with model: @post do |form| |%>|
5    <div>
6      |<%=| form.label "Título" |%>|<br>
7      |<%=| form.text_field :title |%>|
8      |<%| @post.errors.full_messages_for(:title).each do |message| |%>|
9        <div>|<%= message %>|</div>
10     |<% end %>|
11   </div>
12
13   <div>
14     |<%=| form.label "Conteúdo" |%>|<br>
15     |<%=| form.text_area :body |%>|<br>
16     |<%| @post.errors.full_messages_for(:body).each do |message| |%>|
17       <div>|<%= message %>|</div>
18     |<% end %>|
19   </div>
20
21   <div>
22     |<%=| form.submit "Salvar" |%>|
23   </div>
24 |<% end %>|
```

Listagem 4.102: Adicionando método *full_messages_for* ao formulário de criação de novo *post*

Para finalizar a implementação do método *Create*, é necessário fornecer ao usuário uma forma de acessar o formulário de criação de um novo *post*. Isso é feito ao adicionar um *link* no arquivo *app/views/posts/index.html.erb*, que direciona o usuário para a página de criação, conforme mostrado na Linha 4 da Listagem 4.103. Com a criação de novos *posts*, a página inicial passará a exibir o conteúdo conforme o exemplo apresentado na Figura 4.19.

```
1  |<%=| render "page_structure" |%>|
2  <h1>Blog</h1>
3
4  |<%=| link_to "Criar novo Post", new_post_path |%>|
5
6  <ul>
7    |<% @posts.each do |post| %>|
8      <li>
9          |<%=| link_to post.title, post |%>|
10     </li>
11   |<% end %>|
12 </ul>
```

Listagem 4.103: Adicionando *link* para criação de um novo *post* na página inicial



Figura 4.19: Fragmento da página inicial do projeto após a adição do *link* de criação de novo *post*

Para implementar o método *Update* do CRUD, o processo é dividido em etapas, como no método *Create*. Inicialmente, adicionam-se as ações *edit* e *update* no *controller*, no arquivo *app/controllers/posts_controller.rb*, logo abaixo da ação *create*, conforme demonstrado na Listagem 4.104. Na Linha 25, a ação *edit* busca no banco de dados o *post* correspondente ao *id* fornecido como argumento e o armazena na variável de instância *@post*. Já na Linha 24, os dados do formulário são validados e, caso a atualização seja bem-sucedida, a ação redireciona o usuário para a página de exibição do *post* editado; caso contrário, exibe o formulário novamente, acompanhado de uma mensagem de erro.

```ruby
class PostsController < ApplicationController
  def index
    @post = Post.all
  end
  def show
    @post = Post.find(params[:id])
  end
  def new
    @post = Post.new
  end
  def create
    @post = Post.new(post_params)
    if @post.save
      redirect_to root_path
    else
      render :new, status: :unprocessable_entity
    end
  end
  def edit
    @post = Post.find(params[:id])
  end
  def update
    @post = Post.find(params[:id])
    if @post.update(post_params)
      redirect_to @post
    else
      render :edit, status: :unprocessable_entity
    end
  end

  private
    def post_params
      params.require(:post).permit(:title, :body)
    end
end
```

Listagem 4.104: Criando ações *edit* e *update* no *controller*

O formulário de atualização do método *edit* é análogo ao do método *new*, inclusive em termos de código, por meio da criação do novo arquivo *app/views/posts/_form.html.erb*, cujo conteúdo é apresentado na Listagem 4.105. A única diferença em relação ao arquivo *app/views/posts/new.html.erb* é a substituição da variável de instância *@post* por *post*, conforme uma boa prática recomendada pelo próprio *framework*. Dessa forma, o formulário se torna uma *view* compartilhada, ou seja, um *partial*, o que impacta diretamente nos arquivos *app/views/posts/new.html.erb* e *app/views/posts/edit.html.erb*, como ilustrado nas Listagens 4.106 e 4.107.

```
|<%=| form_with model: post do |form| |%>|
  <div>
    |<%=| form.label "Título" |%>|<br>
    |<%=| form.text_field :title |%>|
    |<%| post.errors.full_messages_for(:title).each do |message| |%>|
      <div>|<%= message %>|</div>
    |<% end %>|
  </div>

  <div>
    |<%=| form.label "Conteúdo" |%>|<br>
    |<%=| form.text_area :body |%>|<br>
    |<%| post.errors.full_messages_for(:body).each do |message| |%>|
      <div>|<%= message %>|</div>
    |<% end %>|
  </div>

  <div>
    |<%=| form.submit "Salvar" |%>|
  </div>
|<% end %>|
```

Listagem 4.105: *Partial* do formulário de edição e criação de um *post*

```
|<%=| render "page_structure"  |%>|
<h1>Novo Post</h1>

|<%=| render "form", post: @post |%>|
```

Listagem 4.106: Atualizando página de criação de novo *post* utilizando *partial*

```
|<%=| render "page_structure"  |%>|
<h1>Editar Post</h1>

|<%=| render "form", post: @post |%>|
```

Listagem 4.107: Atualizando página de edição de *post* utilizando *partial*

Para dar acesso à funcionalidade de atualização do *post*, é adicionado um *link* para edição na página de visualização, no arquivo *app/views/posts/show.html.erb*, conforme apresentado na Listagem 4.108. O *link* para retornar à página inicial também é alterado para o uso do *link_to*, conforme mostrado na Linha 7. A página de visualização deve corresponder ao exemplo apresentado na Figura 4.20.

```
1 |<%=| render "page_structure"  |%>|
2 <h1>|<%= @post.title %>|</h1>
3
4 <p>|<%= @post.body %>|</p>
5
6 <ul>
7   <li>|<%=| link_to "Voltar para página inicial", root_path |%>|</li>
8   <li>|<%=| link_to "Editar post", edit_post_path(@post) |%>|</li>
9 </ul>
```

Listagem 4.108: Adicionando *link* para edição do *post*



Figura 4.20: Fragmento da página de visualização de um *post*

Para finalizar a implementação das operações CRUD, é implementada a operação *Delete* para a exclusão de um recurso. Para isso, são necessárias apenas uma rota e uma ação no *controller*. Com o uso de `resources`, o `Rails` já mapeia automaticamente a rota *DELETE /posts/:id* para a ação *destroy*. Assim, no arquivo *app/controllers/posts_controller.rb*, abaixo da ação *update*, é adicionada a ação *destroy*, como mostrado na Listagem 4.109. Na Linha 31, a ação busca no banco de dados o *post* correspondente ao *id* passado como argumento; na Linha 32, chama-se o método *destroy* para excluir o *post* do banco de dados; e, na Linha 33, o usuário é redirecionado para a página inicial.

```
class PostsController < ApplicationController
  def index
    @post = Post.all
  end
  def show
    @post = Post.find(params[:id])
  end
  def new
    @post = Post.new
  end
  def create
    @post = Post.new(post_params)
    if @post.save
      redirect_to root_path
    else
      render :new, status: :unprocessable_entity
    end
  end
  def edit
    @post = Post.find(params[:id])
  end
  def update
    @post = Post.find(params[:id])
    if @post.update(post_params)
      redirect_to @post
    else
      render :edit, status: :unprocessable_entity
    end
  end
  def destroy
    @post = Post.find(params[:id])
    @post.destroy
    redirect_to root_path, status: :see_other
  end

  private
    def post_params
      params.require(:post).permit(:title, :body)
    end
end
```

Listagem 4.109: Criando ação *destroy* no *controller*

Para disponibilizar a opção de exclusão ao usuário, adiciona-se um *link* denominado "Excluir post" no arquivo *app/views/posts/show.html.erb*, conforme mostrado na Listagem 4.110, na Linha 9. Esse *link* executa o método *delete* e envia ao usuário uma confirmação da exclusão do *post*. A nova página de visualização do *post* deverá corresponder ao exemplo apresentado na Figura 4.21.

```
1  |<%=| render "page_structure"  |%>|
2  <h1>|<%= @post.title %>|</h1>
3
4  <p>|<%= @post.body %>|</p>
5
6  <ul>
7    <li>|<%=| link_to "Voltar para página inicial", root_path |%>|</li>
8    <li>|<%=| link_to "Editar post", edit_post_path(@post) |%>|</li>
9    <li>|<%=| link_to "Excluir post", post_path(@post), data: {
10                     turbo_method: :delete,
11                     turbo_confirm: "Você tem certeza que deseja
                           excluir esse post?"
12                   } |%>|</li>
13 </ul>
```

Listagem 4.110: Adicionando *link* para exclusão do *post*



Figura 4.21: Fragmento da página de visualização de um *post* após adição do *link* de exclusão

Por fim, para estilizar a página *index*, cria-se um arquivo de estilo estático *app/assets/stylesheets/index.css* com o código apresentado na Listagem 4.111. Esse código exemplifica a estilização da página, podendo ser ajustado e expandido conforme as necessidades de personalização.

```
1 body {
2     background-color: #c4e6db;
3 }
```

Listagem 4.111: Estilização da página *index*

Para aplicar a estilização na página, insere-se o código da Linha 1 no topo da página *app/views/posts/index.html.erb*, conforme mostrado na Listagem 4.112. Para estilizar as demais páginas da aplicação, segue-se o mesmo padrão. Dessa forma, o projeto "*Blog* Simples" resultará conforme ilustrado na Figura 4.22.

```
|<%=| stylesheet_link_tag 'index.css' |%>|

|<%=| render "page_structure" |%>|
<h1>Blog</h1>

|<%=| link_to "Criar novo Post", new_post_path |%>|

<ul>
  |<% @posts.each do |post| %>|
    <li>
        |<%=| link_to post.title, post |%>|
    </li>
  |<% end %>|
</ul>
```

Listagem 4.112: Adicionando arquivo de estilização na página inicial



Figura 4.22: Fragmento da página inicial estilizada

4.11 Comparação dos *frameworks*

Com o objetivo de apresentar um exemplo prático unificado que utilize os três *frameworks* descritos, foi projetada a funcionalidade "Cadastrar atividade" (Figura 4.23) em um sistema fictício para a gestão de atividades de extensão universitária. Essa funcionalidade permite o cadastro de novas atividades de extensão por meio de um formulário, no qual o aluno anexa a documentação comprobatória de sua participação e pode salvar como rascunho antes da submissão definitiva. Além disso, o formulário de cadastro inclui campos como o título da atividade, a quantidade de horas de participação e as datas de início e término. O banco de dados `SQLite` foi utilizado em todos os projetos, pois não requer configuração manual, sendo o banco padrão nos *frameworks* quando não há especificação de outro banco de dados.

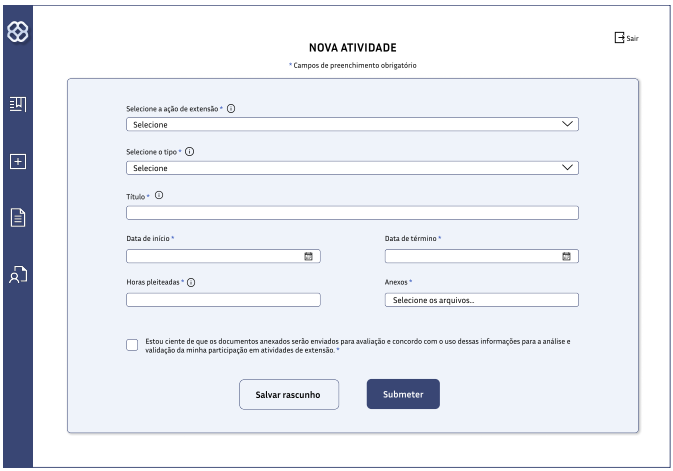


Figura 4.23: Tela “Cadastrar atividade”

4.11.1 Implementação das funcionalidades utilizando Ruby on Rails

Na implementação das funcionalidades “Cadastrar” e “Submeter” utilizando o *framework* `Ruby on Rails`, foram criadas três rotas, conforme ilustrado na Listagem 4.113. A rota `root` exibe a *view index*, que exibe em tela as atividades cadastradas, como a tela “Minhas atividades”, apresentada na prototipagem do *software*. Já as demais rotas são responsáveis, respectivamente, por exibir o formulário de cadastro, conforme mostrado na tela de cadastro de atividade da prototipagem, e por enviar os dados preenchidos no formulário.

```
1 Rails.application.routes.draw do
2   resources :atividades, only: [:index, :show]
3   root 'atividades#index'
4   get "/cadastrar", to: "atividades#new"
5   post "/cadastrar", to: "atividades#create"
6 end
```

Listagem 4.113: Rotas da aplicação - Rails

Em seguida, no *controller*, conforme apresentado na Listagem 4.114, foram definidas cinco ações: a ação `index`, que filtra as atividades de acordo com a situação em que estão; a ação `show`, que busca a atividade no banco de dados e renderiza a página correspondente; a ação `new`, que exibe o formulário de cadastro de uma nova atividade; a ação `create`, que cria a atividade com base nas informações preenchidas no formulário; e, por fim, a ação `atividade_params`, que é usada como parâmetro no momento de salvar a atividade, permitindo a correspondência correta entre os dados e o banco de dados.

```ruby
class AtividadesController < ApplicationController
  def index
    @submetidas = Atividade.where(situacao: 'Em análise')
    @integralizadas = Atividade.where(situacao: 'Aprovada')
    @indeferidas = Atividade.where(situacao: 'Indeferida')
  end
  def show
    @atividade = Atividade.find(params[:id])
    render 'atividades/exibir-atividade'
  end
  def new
    @atividade = Atividade.new
    render 'atividades/cadastrar-atividade'
  end
  def create
    @atividade = Atividade.new(atividade_params)
    if params[:atividade][:acao_extensao] == 'Curso'
      @atividade.tipo = params[:atividade][:tipo_curso]
    elsif params[:atividade][:acao_extensao] == 'Evento'
      @atividade.tipo = params[:atividade][:tipo_evento]
    else
      @atividade.tipo = nil
    end
    if @atividade.save
      redirect_to root_path, notice: 'Atividade cadastrada com sucesso!'
    else
      render 'atividades/cadastrar-atividade'
    end
  end
  def atividade_params
    params.require(:atividade).permit(:acao_extensao, :tipo, :horas, :anexo)
  end
end
```

Listagem 4.114: Controller da aplicação - Rails

As atividades estão dispostas na página inicial separadamente em:

- Submetidas: aguardando aprovação pela banca avaliadora;
- Integralizadas: aprovadas pela banca avaliadora;
- Indeferidas: reprovadas pela banca avaliadora.

O arquivo *model* do projeto define a obrigatoriedade dos campos, além de estabelecer a data de submissão da atividade como a data correspondente ao dia em que a atividade é enviada, conforme mostrado na Listagem 4.115. Para simular as atividades "Integralizadas" e "Indeferidas", utilizou-se o comando `rails console` para alterar os campos `situacao` e `data_aprovacao` diretamente no banco de dados.

```ruby
class Atividade < ApplicationRecord
    has_one_attached :anexo
    validates :acao_extensao, presence: true
    validates :horas, presence: true, numericality: { only_integer:
        true, greater_than: 0 }
    before_create :set_data_submissao
    private
    def set_data_submissao
        self.data_submissao ||= Date.today
    end
end
```

Listagem 4.115: Model da aplicação - Rails

Além disso, na página *index*, é possível visualizar os detalhes de cada tarefa ao clicar sobre a atividade desejada, sendo então redirecionado para a página de visualização da atividade, que exibe as informações cadastradas durante a submissão da atividade, como mostrado na tela "Visualizar atividade" apresentada na prototipagem do *software*. No *link* "Baixar anexo", o usuário pode realizar o *download* da documentação comprobatória de participação na atividade de extensão enviada pelo discente. Ao clicar no *link* "Voltar", o usuário é redirecionado para a página inicial.

Ainda na página inicial, ao clicar no *link* "Cadastrar atividade", o usuário é redirecionado para a página do formulário de cadastro de uma nova atividade. Nesse formulário, o primeiro campo corresponde à Ação de Extensão, definida de acordo com o documento de regulamento das atividades de extensão utilizado na etapa de levantamento de requisitos. As opções disponíveis são: Programa, Projeto, Curso e Evento. Excepcionalmente, as ações Curso e Evento possuem subcategorias. As subcategorias da ação *Curso* são: Formação inicial e Formação continuada, enquanto as subcategorias da ação *Evento* incluem: Congresso, Seminário, Encontro e Simpósio. Assim, ao selecionar uma dessas ações, é exibido um novo campo denominado *tipo*.

Por fim, para armazenar os documentos comprobatórios de participação do aluno enviados por anexo, foi utilizado o *Active Storage*, que facilita o *upload* de arquivos. O *Active Storage* armazena os arquivos localmente no diretório *storage*, garantindo um processo de *upload* mais seguro e eficiente.

4.11.2 Implementação das funcionalidades utilizando Laravel

Para a implementação das funcionalidades utilizando o *framework* `Laravel`, foram criadas cinco rotas, conforme apresentado na Listagem 4.116. A rota principal `/` exibe a página inicial da aplicação. Em seguida, a rota `/cadastrar` utiliza dois métodos: *create*, responsável por exibir o formulário de cadastro, e *store*, que recebe os dados preenchidos no formulário e os salva no banco de dados. A rota `atividades/id` exibe a atividade identificada pelo parâmetro `id`. Por fim, a rota

`download/id` é dedicada ao *download* do anexo enviado no cadastro da atividade, utilizando o parâmetro `id` para localizar o arquivo no banco de dados.

```
<?php
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\AtividadeController;

Route::get('/', [AtividadeController::class, 'index'])->name('
    atividades.index');
Route::get('/cadastrar', [AtividadeController::class, 'create'])->name
    ('atividades.create');
Route::post('/cadastrar', [AtividadeController::class, 'store'])->name
    ('atividades.store');
Route::get('/atividades/{id}', [AtividadeController::class, 'show'])->
    name('atividades.show');
Route::get('/download/{id}', [AtividadeController::class, 'download'])
    ->name('atividade.download');
```

Listagem 4.116: Rotas da aplicação - Laravel

No *controller*, conforme apresentado na Listagem 4.117, foram definidas cinco ações: a ação `index`, que organiza as atividades de acordo com sua exibição nas *views*, filtrando-as entre submetidas, integralizadas e indeferidas; a ação `create`, que exibe a *view* de cadastro contendo o formulário; a ação `store`, que recebe a requisição de cadastro de uma nova atividade, inicialmente verificando se o arquivo foi anexado e, em seguida, cadastrando a nova atividade no banco de dados; a ação `show`, que exibe uma atividade a partir do `id` enviado como argumento; e, por fim, a ação `download`, responsável pelo *download* do arquivo anexado no cadastro da atividade, acessado ao clicar no botão “Baixar anexo” na página de exibição da atividade.

```php
<?php
class AtividadeController extends Controller {
public function index() {
    $submetidas = Atividade::where('situacao', 'Em análise')->get();
    $integralizadas = Atividade::where('situacao', 'Integralizada')->
        get();
    $indeferidas = Atividade::where('situacao', 'Indeferida')->get();
    return view('index', compact('submetidas', 'integralizadas', '
        indeferidas')); }
public function create() { return view('cadastrar-atividade'); }
public function store(Request $request) {
    if ($request->hasFile('anexo')) {
        $anexoPath = $request->file('anexo')->store('anexos', 'public'
            );
    } else {
        return redirect()->back()->withErrors(['anexo' => 'O arquivo
            deve ser enviado.']); }
    $atividade = Atividade::create([
        'acao_extensao' => $request->input('acao_extensao'),
        'tipo' => $request->input('tipo-curso') ?? $request->input('
            tipo-evento'),
        'horas' => $request->input('horas'),
        'anexo' => $anexoPath ]);
    return redirect('/')->with('success', 'Atividade cadastrada com
        sucesso!'); }
public function show($id) {
    $atividade = Atividade::findOrFail($id);
    return view('exibir-atividade', compact('atividade')); }
public function download($id) {
    $atividade = Atividade::findOrFail($id);
    if ($atividade->anexo) {
        $filePath = storage_path('app/public/' . $atividade->anexo);
        if (file_exists($filePath)) {
            return response()->download($filePath);
        } else {
            return redirect()->back()->with('error', 'Anexo não
                encontrado.'); }}
    return redirect()->back()->with('error', 'Anexo não disponível.');
    }
}
```

Listagem 4.117: Controller da aplicação - Laravel

Em seguida, no arquivo *model*, apresentado na Listagem 4.118, são definidos os campos presentes no banco de dados e, por padrão, determina-se que o campo `situacao` seja preenchido com o valor "Em análise". Além disso, a função *boot* é utilizada para adicionar automaticamente a data de submissão e a data de aprovação, sendo que a data de submissão corresponde ao momento em que a atividade é cadastrada e a data de aprovação corresponde à data em que a atividade foi avaliada pela banca.

```php
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
class Atividade extends Model {
    use HasFactory;
    protected $fillable = [
        'acao_extensao',
        'tipo',
        'horas',
        'anexo',
        'data_submissao',
        'situacao',
        'data_avaliacao'
    ];
    protected $attributes = [
        'situacao' => 'Em análise',
    ];
    public static function boot() {
        parent::boot();
        static::creating(function ($model) {
            $model->data_submissao = now();
        });
        static::updating(function ($model) {
            if ($model->isDirty('situacao') && $model->situacao === '
                Integralizada') {
                $model->data_avaliacao = now();
            }
        });
    }
}
```

Listagem 4.118: Model da aplicação - Laravel

Semelhante à implementação utilizando `Rails` (Seção 4.11.1), as atividades na página inicial estão separadas entre as categorias de "submetidas", "integralizadas" e "indeferidas". Para isso, foi utilizado o comando *php artisan tinker* para modificar manualmente o campo `situacao` das atividades, a fim de simular as atividades "Integralizadas" e "Indeferidas".

4.11.3 Implementação das funcionalidades utilizando Django

Na implementação utilizando `Django`, existem dois tipos de arquivos de rotas: o *urls.py* na pasta padrão com o nome do projeto e o *urls.py* do app criado, chamado "atividades". Na Listagem 4.119, é apresentado o arquivo de rotas da pasta padrão, que define a rota da página *admin* do `Django` e a rota `atividades/`, correspondente à página inicial do projeto.

```
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path('admin/', admin.site.urls),
    path('atividades/', include('atividades.urls')),
]
```

Listagem 4.119: urls.py da pasta padrão da aplicação - Django

O arquivo *admin.py*, responsável por definir a configuração da interface administrativa do `Django`, é apresentado na Listagem 4.120. Nele, o modelo *Atividade* é importado e a tabela é dividida em cinco colunas: `acao_extensao`, `get_tipo`, `horas`, `data_submissao`.

```
from django.contrib import admin
from .models import Atividade
class AtividadeAdmin(admin.ModelAdmin):
    list_display = ('acao_extensao', 'get_tipo', 'horas', '
        data_submissao')
    list_filter = ('acao_extensao', 'horas', 'data_submissao')
    def get_tipo(self, obj):
        if obj.acao_extensao == "Curso":
            return obj.tipo_curso or "-"
        elif obj.acao_extensao == "Evento":
            return obj.tipo_evento or "-"
        return "-"
    get_tipo.short_description = "Tipo"
admin.site.register(Atividade, AtividadeAdmin)
```

Listagem 4.120: Admin da aplicação - Django

Já a Listagem 4.121 apresenta o arquivo de rotas do app "atividades". A rota principal `/` exibe a *view* que lista todas as atividades cadastradas, enquanto a rota `cadastrar/` exibe o formulário de cadastro de uma nova atividade. A rota `<int:id>/` exibe a atividade correspondente ao `id` passado como parâmetro de rota.

```python
from django.urls import path
from . import views
app_name = 'atividades'
urlpatterns = [
    path('', views.lista_atividades, name='lista_atividades'),
    path('cadastrar/', views.cadastrar_atividade, name='
        cadastrar_atividade'),
    path('<int:id>/', views.exibir_atividade, name='exibir_atividade')
        ,
]
```

Listagem 4.121: Rotas da aplicação - Django

No arquivo *views.py*, foram definidas três *views*, conforme apresentado na Listagem 4.122. A *view* "lista_atividades" é responsável por filtrar as atividades por situação e por renderizar o *template* "index.html". A *view* "cadastrar_atividade" renderiza o formulário, valida a resposta do cadastro e salva as informações no banco de dados. Por fim, a *view* "exibir_atividade" recebe como argumento o `id` da atividade e renderiza o *template* "exibir-atividade.html" com os dados da atividade requisitada.

```python
from django.shortcuts import render, get_object_or_404, redirect
from .models import Atividade
from .forms import AtividadeForm
def lista_atividades(request):
    submetidas = Atividade.objects.filter(situacao='Em análise')
    integralizadas = Atividade.objects.filter(situacao='Integralizada'
        )
    indeferidas = Atividade.objects.filter(situacao='Indeferida')
    context = {
        'submetidas': submetidas,
        'integralizadas': integralizadas,
        'indeferidas': indeferidas
    }
    return render(request, 'atividades/index.html', context)
def cadastrar_atividade(request):
    if request.method == "POST":
        return redirect("atividades:lista_atividades")
    return render(request, "atividades/form.html")
def exibir_atividade(request, id):
    atividade = get_object_or_404(Atividade, pk=id)
    return render(request, 'atividades/exibir-atividade.html', {'
        atividade': atividade})
```

Listagem 4.122: Views da aplicação - Django

Finalmente, no arquivo *models.py*, é definido o modelo de dados da aplicação, especificando os campos da tabela a serem armazenados no banco de dados, conforme mostrado na Listagem 4.123. O modelo inclui campos como *acao_extensao*, *anexo* e *situacao*, entre outros. Além disso, o modelo garante que apenas valores válidos possam ser atribuídos aos campos da tabela.

```
from django.db import models
class Atividade(models.Model):
    ACAO_EXTENSAO_CHOICES = [
        ('Programa', 'Programa'),
        ('Projeto', 'Projeto'),
        ('Curso', 'Curso'),
        ('Evento', 'Evento'),
    ]
    TIPO_CURSO_CHOICES = [
        ('Formação inicial', 'Formação inicial'),
        ('Formação continuada', 'Formação continuada'),
    ]
    TIPO_EVENTO_CHOICES = [
        ('Congresso', 'Congresso'),
        ('Seminário', 'Seminário'),
        ('Encontro', 'Encontro'),
        ('Simpósio', 'Simpósio'),
    ]
    SITUACAO_CHOICES = [
        ('Em análise', 'Em análise'),
        ('Integralizada', 'Integralizada'),
        ('Indeferida', 'Indeferida'),
    ]
    acao_extensao = models.CharField(max_length=20, choices=
        ACAO_EXTENSAO_CHOICES)
    tipo_curso = models.CharField(max_length=30, choices=
        TIPO_CURSO_CHOICES, blank=True, null=True)
    tipo_evento = models.CharField(max_length=30, choices=
        TIPO_EVENTO_CHOICES, blank=True, null=True)
    horas = models.PositiveIntegerField()
    anexo = models.FileField(upload_to='anexos/', blank=True, null=
        True)
    data_submissao = models.DateField(auto_now_add=True)
    situacao = models.CharField(max_length=50, default="Em análise")

    def __str__(self):
        return f"{self.acao_extensao} - {self.horas}h"
```

Listagem 4.123: Model da aplicação - Django

4.12 Pontos de Comparação

A partir da implementação das funcionalidades, foi possível realizar comparações entre os *frameworks* utilizando os seguintes pontos:

- **Curva de aprendizado:** Do ponto de vista da linguagem de programação em que os *frameworks* foram desenvolvidos, visto que a dificuldade da linguagem impacta a curva de aprendizado, o `Django` apresenta a melhor curva de aprendizado devido à facilidade e à ampla utilização da linguagem `Python`. Em seguida, temos o `Laravel`, que é popular devido à linguagem `PHP`. Por último, o `Rails`, desenvolvido com `Ruby`, é menos popular que os demais;

- **Tamanho dos arquivos:** O tamanho dos arquivos de um *framework* pode influenciar na manutenção a longo prazo. Ao comparar o tamanho final do projeto em cada um dos *frameworks*, o `Django` apresenta o menor tamanho, com 446 KB, seguido do projeto em `Rails` com 14,9 MB e, por fim, o projeto em `Laravel` com 727 MB;

- **Verbosidade do framework:** A verbosidade reflete a simplicidade do *framework*. Nesse sentido, o `Rails` e o `Laravel` adotam uma abordagem mais concisa, enquanto o `Django` prioriza clareza e legibilidade, exigindo mais detalhes na escrita do código. Assim, a escolha da tecnologia quanto à verbosidade deve considerar um equilíbrio entre clareza e manutenibilidade a longo prazo.

- **Documentação de apoio:** As documentações dos *frameworks* `Django` e `Rails` contêm tutoriais auxiliares para a implementação de um pequeno projeto inicial, a fim de que o desenvolvedor conheça o básico do *framework*, sendo a documentação do `Django` a mais clara e didática. Já a documentação do `Laravel` não segue uma ordem lógica de implementação de um projeto para iniciantes da linguagem, dificultando a estruturação de um projeto inicial; no entanto, apresenta exemplos genéricos para tópicos presentes na documentação. Além disso, o idioma padrão das três documentações comparadas é o inglês, mas a documentação do `Django` possui tradução para o português brasileiro, facilitando o uso a desenvolvedores não fluentes em inglês.

- **Comunidades para tirar dúvidas:** Os *frameworks* utilizados possuem comunidades dedicadas ao estudo da linguagem e disponíveis para tirar dúvidas, com `Rails` e `Django` se destacando por possuírem uma comunidade brasileira dedicada.

- **Camadas de segurança:** os principais ataques em uma aplicação web são SQL Injection, XSS (*Cross-Site Scripting*) e CSRF (*Cross-Site Request Forgery*). O SQL Injection é um ataque que consiste em inserir código malicioso em consultas SQL, o que pode destruir o banco de dados. O XSS é um ataque em que o invasor injeta código malicioso na aplicação, que será executado na máquina do usuário. Já o CSRF é um ataque de envio malicioso de solicitações

Tabela 4.2: Comparação da camada de segurança entre os *frameworks*

Segurança	Rails	Laravel	Django
SQL Injection	ORM Active Record fornece métodos de consulta que evita SQL Injection	Eloquent ORM que fornece métodos de consulta que evita SQL Injection	Evita SQL Injection separando o código de consulta dos parâmetros de consulta
XSS	Tem prevenção XSS automática com helpers	Tem prevenção XSS automática com Blade	Tem prevenção XSS automática com templates
CSRF	Proteção por meio do módulo RequestForgeryProtection	Proteção por meio da geração automática de tokens por sessão	Proteção integrada

não autorizadas de um usuário para um *site*, que se aproveita da confiança do navegador de um usuário autenticado. Assim, utilizando como critérios de segurança os principais ataques contra aplicações web, o desempenho dos *frameworks* é apresentado na Tabela 4.2.

Após a análise das camadas de segurança dos três *frameworks*, pode-se afirmar que todos oferecem boa segurança contra os principais ataques em uma aplicação web; porém, o `Django` se destaca por oferecer proteção automática contra esses ataques.

- **Utilização no mercado:** segundo o Survey StackOverflow 2024[18], em uma pesquisa cujo objetivo é elencar as ferramentas e tecnologias mais usadas atualmente pelos desenvolvedores, para a pergunta "Em quais estruturas e tecnologias da web você realizou um extenso trabalho de desenvolvimento no ano passado?", de um total de 38.132 respostas, 11,4% dos desenvolvedores responderam o *framework* `Django`, 8,6% responderam `Laravel` e 5,2% responderam `Ruby on Rails`. Ainda na mesma pesquisa foi questionado "Em quais (tecnologias web) deseja trabalhar no próximo ano?", e para 5.010 respostas, 14,3% para `Django`, 4,8% para `Laravel` e 1,9% para `Ruby on Rails`. Além disso, segundo o Statista[19] em uma pesquisa que busca elencar os *frameworks Web* mais usados entre os desenvolvedores em todo o mundo em 2024, de um total de 48.503 entrevistados, 12% responderam `Django`, 7,9% responderam `Laravel` e 4,7% responderam `Ruby on Rails`. A partir dessas pesquisas, pode-se observar um maior uso do *frameworks* `Django` no mercado, seguido pelo `Laravel` e `Ruby on Rails`. Ademais, `Django` aponta uma maior perspectiva de crescimento para os próximos anos quando comparado aos outros dois *frameworks*.

[18] https://survey.stackoverflow.co/2024/
[19] https://www.statista.com/

Capítulo 5

Banco de dados para Web

Francisca Jannielly Garcia da Costa (UFC)

Carlos Elmano de Alencar e Silva (UFC)

Fernando Rodrigues de Almeida Júnior (UFC)

Fischer Ferreira (Unifei)

Este capítulo tem como principal objetivo apresentar uma visão geral sobre banco de dados para web. O capítulo discute os seguintes tópicos: Aplicações web (Seção 5.1), uma visão geral sobre bancos de dados (Seção 5.2), Sistemas de Gerenciamento de Banco de Dados (Seção 5.3), MySQL (Seção 5.4), MongoDB (Seção 5.5), Redis (Seção 5.6) e Mapeamento objeto-relacional (Seção 5.7).

5.1 Aplicações Web

Criada pelo Departamento de Defesa dos Estados Unidos da América (EUA) em 1960, a *Advanced Research Projects Agency Network* (ARPANET) foi a primeira rede de computadores, cuja principal função era facilitar a comunicação em territórios em zona de guerra. Logo mais, na década de 70, surgiram os primeiros protocolos de comunicação, como o *Transmission Control Protocol* (TCP) e o *Internet Protocol* (IP), e foi a partir dessa época que a rede passou a ser chamada de Internet. O cientista da computação Tim Berners-Lee foi o responsável por desenvolver o primeiro buscador de informações na Internet. O navegador permitia o acesso a informações variadas por meio de um conjunto de hiperlinks. Esse mecanismo foi o pontapé inicial para o que se conhece atualmente como web (CONSORTIUM, 2023).

Aplicações web são *softwares* que são executados diretamente em um navegador web, permitindo que o usuário interaja e consuma o conteúdo por meio de uma interface gráfica do usuário (GUI). Estas aplicações podem ser executadas em qualquer dispositivo com um navegador web e uma conexão com a Internet. Sendo assim, é possível acessá-la por meio de computadores pessoais, notebooks, celulares e tablets.

Atualmente, o desenvolvimento de aplicações web é dividido em duas vertentes, chamadas de *stacks*: *stack front-end* e *stack back-end* (MOZILLA, 2023b). O *front-end* é o lado visual da aplicação, ou seja, a parte com a qual o usuário interage. Existe uma variedade de elementos que fazem parte do desenvolvimento *front-end*, como *layouts*, botões, *cards*, menus, formulários, textos e animações. As tecnologias-base desta *stack* são *HyperText Markup Language* (HTML), *Cascading Style Sheets* (CSS) e *JavaScript* (JS). O HTML é uma linguagem de marcação usada para criar a estrutura da aplicação. O CSS é usado para estilizar a estrutura criada com HTML, atribuindo cor, formato e outras propriedades de estilo. Já o JS, por sua vez, é responsável por tornar a estrutura dinâmica e facilitar a interação com a página. Existem outras linguagens usadas na web para *front-end*, além de uma infinidade de extensões e *frameworks* que facilitam a criação de aplicações (MILETTO; BERTAGNOLLI, 2014).

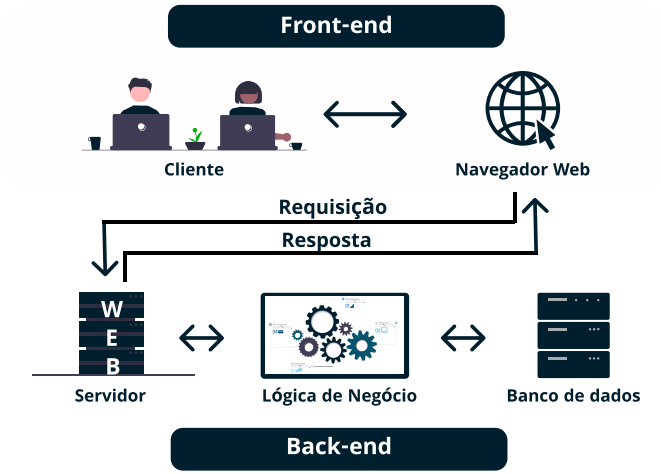
O *back-end* é o lado lógico da aplicação e é invisível ao usuário final. Geralmente, é a parte da infraestrutura em que são desenvolvidos os serviços a serem consumidos pelo *front-end* e na qual há comunicação e manipulação de informações quem vêm de bancos de dados. A interação entre a *stack front-end* e *back-end* se dá por meio de uma API, por meio da qual é possível estabelecer uma comunicação entre sistemas

e serviços seguindo as lógicas de negócio e aplicação previstas na especificação do projeto. A comunicação em uma API é feita a partir de um conjunto de métodos, cada um com uma função específica, sendo que os métodos mais populares do protocolo HTTP (MOZILLA, 2023c) são:

- *GET*: utilizado para solicitar um ou mais recursos;
- *POST*: envia uma informação para ser armazenada;
- *PUT*: envia uma atualização para um recurso já armazenado;
- *DELETE*: remove uma informação/recurso armazenado.

Assim como no *front-end*, há uma diversidade de ferramentas, linguagens e seus *frameworks* usados no desenvolvimento de serviços *back-end*, como o próprio `JavaScript e TypeScript` com `Node.js`, `Java` com `Spring Boot`, `C#` com `.NET`, `Python` com `Django`, além de tecnologias de bancos de dados, como bancos relacionais e não relacionais (IMASTER, 2023).

A Figura 5.1 apresenta um diagrama funcional de toda a infraestrutura envolvida em aplicações web, do *front-end* ao *back-end*.



Fonte: Adaptado de (DICODING, 2023)

Figura 5.1: Comunicação de aplicações web

Pode-se observar no diagrama da Figura 5.1 que enquanto o *front-end* se localiza no lado do cliente, o *back-end* fica no lado do servidor, de tal forma que o cliente interage com o *front-end* da aplicação por meio do navegador, que executa localmente no dispositivo do usuário. O *front-end*, por sua vez, interage com o

back-end por meio da API. Todo esse fluxo bidirecional de requisições e respostas baseia-se no protocolo HTTP (MOZILLA, 2023c).

5.2 Uma visão geral sobre bancos de dados

Banco de dados pode ser descrito como todo e qualquer conjunto de informações que representam um domínio específico, armazenadas eletronicamente em um ou mais dispositivos e que podem estar organizadas de maneira relacional ou não relacional. Essas informações podem variar de um simples dado a estruturas de maior complexidade, como, por exemplo, arquivos de áudio e vídeo. O armazenamento destes dados pode ocorrer de maneira estática ou dinâmica; assim, estarão disponíveis para o usuário visualizar e também para serem manipulados e gerenciados (REDIS, 2023b), (MONGODB, 2023), (MYSQL, 2023). A fim de garantir que a manipulação e gerenciamento dos dados ocorram de maneira precisa e eficiente, é necessário levar em consideração a forma como os dados são organizados. Foi com esse intuito que surgiu o processo de modelagem de dados.

5.2.1 Modelagem de dados

A modelagem de dados é uma etapa indispensável no processo de criação de um banco de dados, pois consiste em uma sequência de passos que visam identificar o contexto em que os dados estão inseridos e desenvolver um modelo abstrato que os represente adequadamente e de forma funcional. Esse modelo serve não somente para definir como os dados são organizados, mas também para auxiliar a garantir a integridade e consistência, já que, com um modelo de dados bem definido, fica mais fácil validar as informações de entrada e saída.

Existe uma hierarquia de três níveis em que a modelagem de dados pode ser categorizada, que são: conceitual, lógico e físico (MACORATTI, 2023). A Figura 5.2 ilustra a organização do fluxo dos níveis de modelagem de dados.

- Nível Conceitual: esse nível é o primeiro, e trata-se do mais abstrato, pois é nele que será feita uma análise e levantamentos dos requisitos, a fim de definir as estruturas em que os dados estarão armazenadas e/ou relacionados;
- Nível Lógico: nesse nível define-se de fato o modelo de dados que será utilizado, e para isso, são levadas em consideração as informações levantadas e definidas no nível conceitual;
- Nível Físico: esse nível, como o próprio nome já diz, trata de aspectos físicos relacionados ao armazenamento de informações. Logo, é nele que são implementados o modelo de dados e as configurações definidas no nível lógico.

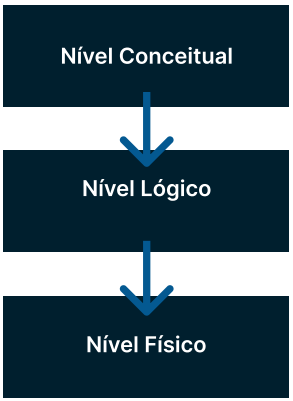


Figura 5.2: Fluxo dos níveis de modelagem

Fonte: Adaptado de (LEYENDECKER, 2023)

A nível conceitual, existe um modelo que é bastante difundido: o *Modelo Entidade Relacionamento* (MER). O MER, por estar no primeiro nível da hierarquia, trata da organização de um projeto de banco de dados de alto nível de abstração. Para compreender como esse modelo funciona, é ideal entender as definições para os conceitos de: entidade, relacionamento, atributos e cardinalidade.

- Entidade: uma entidade busca representar um conceito do mundo real, como, por exemplo, um cliente ou produto.
- Atributo: é uma característica que uma determinada entidade possui, então, se produto é uma entidade, seus atributos podem ser: identificador, nome, peso, data de validade, etc.
- Relacionamento: é o nome dado à associação entre duas ou mais entidades.
- Cardinalidade: é o indicativo de quantas instâncias de uma entidade estão associadas a outra, em um relacionamento. De maneira geral, a cardinalidade pode ser de três tipos:
  - Um para Um (1:1): onde uma instância de uma entidade só está relacionada a uma única instância de outra entidade em um relacionamento;
  - Um para Muitos (1:N): onde uma instância de uma entidade pode se relacionar com várias instâncias de outra entidade em um relacionamento;
  - Muitos para Muitos (N:M): onde várias instâncias de uma entidade podem estar relacionadas a várias instâncias de uma outra entidade em um relacionamento;

Existem alguns modelos visuais que auxiliam a compreender a organização dos dados no modelo escolhido. Um dos mais populares é o diagrama de Entidade-Relacionamento, conhecido como Diagrama ER. Esse modelo consiste em representar, por meio de símbolos, os elementos do modelo de dados escolhido e organizá-los em um diagrama. Os retângulos representam as entidades, os losangos representam os relacionamentos e as elipses representam os atributos das entidades. A Figura 5.3 apresenta o relacionamento entre duas entidades: "Cliente", que possui 3 atributos (CódigoCliente, EnderecoEmail e Nome) e "Produto", que possui 5 atributos (CódigoProduto, Nome, Peso, Preço e Validade). O relacionamento entre as duas entidades é denominado "Venda".

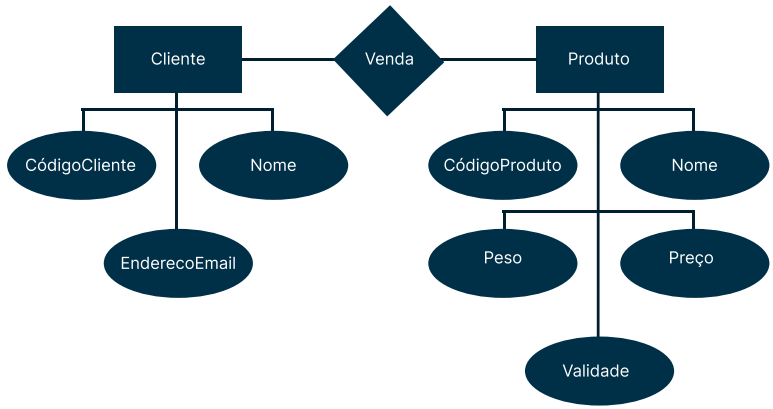


Figura 5.3: Representação do diagrama ER da relação de venda.

Ao nível lógico, em banco de dados, existem dois grandes grupos de modelos de dados: os relacionais e os não relacionais (GARCIA-MOLINA et al., 2008).

5.2.2 Modelo de Dados Relacional

Criado por volta de 1969, com o intuito de abstrair as informações físicas (CODD, 1969), o modelo de dados relacional é um dos padrões de banco de dados mais utilizados. Sua popularidade se deve ao seu padrão rígido de organização das informações armazenadas, que facilita questões de consistência e segurança (JOSE; ABRAHAM, 2020). Para aprender como esse modelo funciona, é de suma importância entender os seguintes conceitos:

- Relação: em banco de dados, as relações, ou tabelas, são estruturas usadas para organizar os dados. Considerando o MER, as tabelas estão fortemente vinculadas ao conceito de entidades.
- Tupla: em banco de dados, as tuplas são os registros de uma relação. Ou seja, as linhas de uma tabela;

- Coluna: em banco de dados, as colunas estão associadas às características ou atributos de uma relação (Entidade);
- Chave-Primária: refere-se a um ou mais atributos que identificam de forma única cada tupla dentro de uma tabela;
- Chave-Estrangeira: representa uma chave primária de outra tabela e serve para estabelecer uma ligação entre duas tabelas;
- Restrições de Integridade: são as regras impostas sobre um determinado conjunto de dados a fim de assegurar a segurança e integridade.

A Álgebra Relacional é uma linguagem baseada em conceitos matemáticos que permite a realização de operações em um conjunto de dados. As operações podem incluir: seleção, união, interseção, diferença ou produto cartesiano. A Figura 5.4 ilustra visualmente as operações que podem ser realizadas por meio da Álgebra Relacional, tais como seleção, união, diferença e interseção.

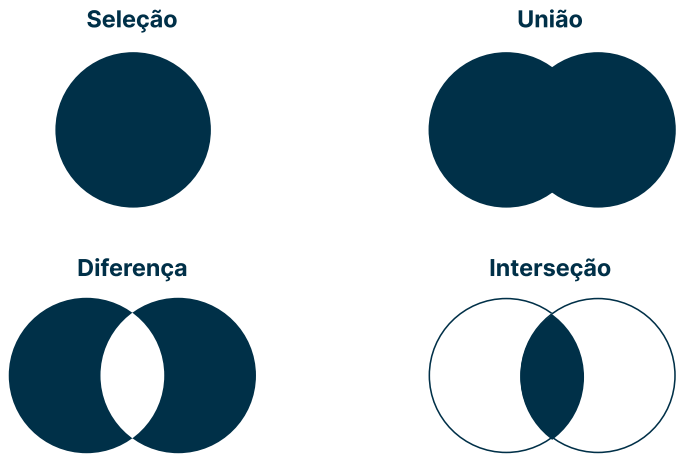


Figura 5.4: Operações de conjuntos

Fonte: Adaptado de (IONOS, 2018)

- Operação de seleção: seleciona as tuplas de uma relação (tabela) que satisfazem uma condição específica. Não deve ser confundida com projeção, que se refere à escolha de colunas.
- Operação de União: combina os dados de duas relações (tabelas), retornando todas as tuplas presentes em ambas. Para que a operação de união seja válida, as relações devem ter o mesmo número e o mesmo tipo de colunas.

- Operação de interseção: seleção de dados que pertencem simultaneamente aos conjuntos envolvidos.
- Operação de diferença: seleção de dados presentes em um conjunto, mas não no outro.
- Operação de Produto Cartesiano: combina todas as tuplas de uma relação com todas as tuplas de outra, resultando em uma nova relação com todas as possíveis combinações de tuplas. Não leva em consideração as dimensões das relações, apenas realiza a combinação entre elas.

Tendo como base as operações da Álgebra Relacional, foi criada uma linguagem de consulta de banco de dados chamada de *Structured Query Language* (SQL). Essa linguagem possui palavras reservadas que facilitam a realização de operações de união, seleção, interseção, diferença e produto cartesiano. Criado em 1970 pela IBM, o SQL tornou-se a linguagem padrão para bancos de dados relacionais e é frequentemente usado como modelo para outras linguagens de consulta (IBM, 2023a).

5.2.3 Modelo de Dados Não Relacional

Diante da heterogeneidade e do grande volume de dados provenientes da popularidade de aplicações da web, *Internet of Things* (IOT) e serviços em nuvem, bancos de dados relacionais acabaram por não ser uma alternativa suficiente para lidar com dados não estruturados. Essa problemática abriu margem para que organizações buscassem novas metodologias para solucionar esse entrave, e foi a partir disso que surgiram os Sistema de Gerenciamento de Banco de Dados (SGBD)s não relacionais, também conhecidos como *Not Only Structured Query Language* (NoSQL) (CELESTI et al., 2019). NoSQL é um termo que designa modelos de dados que não usam a linguagem SQL como principal ferramenta para realizar consultas e operações. Atualmente, existe uma variedade de SGBDs no mercado, e o que diferencia cada um deles é o modelo de dado ao qual ele procura atender (RAMAKRISHNAN; GEHRKE, 2008). Os SGBDs não relacionais podem ser classificados em 4 tipos: Orientado a Documentos, Chave e Valor, Baseado em Colunas, e Orientado a Grafos (KHASAWNEH et al., 2020).

5.2.4 Bancos de dados orientado a documentos

Um SGBD orientado a documentos armazena os dados em documentos no formato *Binary JSON* (BSON), que é uma variação do objeto JSON [1]. Ao contrário do documento textual, o BSON é armazenado em formato binário. Por ser flexível, essa forma de armazenamento possibilita um sistema de dados de maior escalabilidade, sendo ideal para dados não estruturados ou semiestruturados. A Listagem 5.1

[1] Trabalhando com JSON

é um documento que representa um estudante com os atributos: matrícula, nome, idade e as matérias nas quais o mesmo está matriculado.

```
{
        "matricula": "123",
        "nome": "Jannielly",
        "idade": 22,
        "materias": [
           {
              "id": 1,
              "nome": "Ciências de Dados",
              "quantidade_de_horas": 64
            }
        ]
}
```

Listagem 5.1: Representação de um documento em formato JSON

5.2.5 Bancos de dados de chave e valor

Um SGBD orientado a chave e valor, como o próprio nome diz, armazena seus dados em termos de chave (que é responsável por identificar a propriedade) e o valor (que é o conteúdo). Não existe um formato fixo para o valor, portanto, ele pode variar entre uma simples variável, uma lista ou um objeto *JavaScript Object Notation* (JSON). Os bancos de dados de chave e valor são reconhecidos por duas características importantes: seu modelo de acesso direto e sua estrutura otimizada, que juntas, possibilitam o rápido acesso aos valores e reduzem a necessidade de um processamento carregado. Esse tipo de armazenamento é voltado para aplicações que necessitam de maior velocidade na recuperação e na manipulação de dados. A Listagem 5.2 representa uma estrutura de chave valor. A chave possui o identificador da estrutura, que é "pessoa123"e o valor é um objeto que possui algumas propriedades da "pessoa123"como: nome, altura e peso.

```
Chave: "pessoa123"
Valor: {
    "nome": "Jannielly",
    "altura": 159,
    "peso": "65kg"
}
```

Listagem 5.2: Representação da estrutura chave e valor

5.2.6 Bancos de dados baseados em colunas

Um SGBD baseado em colunas segue um princípio parecido com o modelo relacional, porém, ao invés dos dados serem armazenados em linhas, são armazenados em colunas. Essa forma de persistência de dados possibilita operações com maiores

velocidades, já que as consultas que precisam de apenas algumas colunas podem ser executadas de maneira muito mais eficiente do que em bancos de dados relacionais tradicionais, que armazenam os dados por linha. Na Tabela 5.1, cada coluna representa as propriedades de um conjunto de pessoas, como nome, idade e sexo.

Tabela 5.1: Tabela colunar

Nome	Jannielly	Ana Lara	Klayver	Jander
Idade	22	20	22	21
Sexo	Feminino	Feminino	Masculino	Masculino

5.2.7 Baseado em grafos

Um SGBD baseado em grafos armazena suas informações em grafos. Um grafo é uma estrutura de dados composta por vértices e arestas. Essa estrutura é bastante difundida na área da computação, já que possibilita representar conexões entre objetos de maneira fácil de ser compreendida, sendo fortemente utilizada em aplicações que necessitam de uma sequência de ligações, como em redes sociais. Na Listagem 5.3 existe um nó central que representa uma pessoa chamada "Jannielly", a mesma possui 2 amigos conectados a ela por uma aresta do grafo.

```
{
      "nome": "Jannielly",
      "idade": 22,
      "amigos": [
        {
          "nome": "Ana Lara",
          "idade": 20
        },
        {
          "nome": "Gisely",
          "idade": 22
        }
      ]
}
```

Listagem 5.3: Representação de um grafo em formato JSON

5.3 Sistema de gerenciamento de banco de dados

SGBD é um software cujo objetivo principal é oferecer ao usuário um ambiente eficiente, organizado e seguro para realizar operações e transações com dados. Com ele, é possível gerenciar um banco de dados desde sua criação, facilitando a visualização e a manipulação e garantindo a consistência e a permanência de dados nele armazenados (GARCIA-MOLINA et al., 2008). Em busca de facilitar a realização de consultas e manipulações em bancos de dados relacionais e não relacionais, foi criada uma diversidade de SGBDs, cada um com um propósito diferente, sendo os mais populares (DBENGINES, 2023):

- `Oracle Server`: Lançado em 1980 e desenvolvido pela *Oracle Corporation*, o `Oracle Server` é um SGBD que atende ao modelo de dados relacional e é

de uso comercial. Suas principais características são o suporte a Transações ACID, multiplataforma, e a diversas linguagens de programação, como: `C, C Sharp, C++,` entre outras.

- `MySQL`: Lançado em 1995 e pertencente à *Oracle Corporation*, o `MySQL` é um SGBD que atende ao modelo de dados relacional e é open source. Suas principais características são o suporte a Transações ACID, multiplataforma, e a diversas linguagens de programação, como: `Ada, Java, JavaScript (Node.js)`, entre outras.

- `Microsoft SQL Server`: Lançado em 1989 e pertencente à *Microsoft Corporation*, o `Microsoft SQL Server` é um SGBD que atende ao modelo de dados relacional e de uso comercial. Suas principais características são o suporte a Transações ACID, aos sistemas operacionais Linux e Windows e a diversas linguagens de programação, como: `Delphi, Go, Java,` entre outras.

- `PostgreSQL`: Lançado em 1989 e pertencente à *PostgreSQL Global Development Group*, o `PostgreSQL` é um SGBD que atende ao modelo de dados relacional e é *open source*. Suas principais características são o suporte a Transações ACID, diversos sistemas operacionais e a uma variedade de linguagens de programação, como: `PHP, Python, Tcl,` entre outras.

- `MongoDB`: lançado em 2007 e desenvolvido pela *MongoDB Corporation*, o `MongoDB` é um SGBD que atende ao modelo de dados não relacional e é *open source*. Suas principais características são a sua orientação a documentos, Transações ACID de vários documentos com isolamento instantâneo, ter suporte a diversos sistemas operacionais, e a uma variedade de linguagens de programação, como: `Actionscript, C, C Sharp,` entre outras.

- `Redis`: lançado em 2009 e pertencente à *Redis project core team*, o `Redis` é um SGBD que atende ao modelo de dados não relacional e *open source*. Suas principais características são sua orientação a chave e valor, ter suporte a diversos sistemas operacionais, e a diversas linguagens de programação, como: `C, CSharp, C++,` entre outras.

- `Elasticsearch`: lançado em 2010 e pertencente à *Elastic*, o `Elasticsearch` é um SGBD que atende ao modelo de dados não relacional e é *open source*. Suas principais características são seu mecanismo de pesquisa e análise distribuído e *RESTful* [2] moderno, suporte a todos os sistemas operacionais com uma *Virtual Machine* (VM) `Java` e a diversas linguagens de programação, como: `.Net, Groovy, Java,` entre outras.

- `Cassandra`: lançado em 2008 e pertencente à *Apache Software Foundation*, o `Cassandra` é um SGBD que atende ao modelo de dados não relacional e é *open source*. Suas principais características são o seu fator de replicação

[2] *RESTful*

selecionável, suporte a diversos sistemas operacionais e a uma variedade de linguagens de programação, como: `C Sharp, C++, Clojure,` entre outras.

A qualidade de um SGBD depende de diversos fatores, por exemplo, o atendimento às limitações e especificidades do modelo de dados especificado para um dado projeto. Assim, existem algumas temáticas que são bastante difundidas quando se trata do modelo de SGBD a ser utilizado, e nas subseções seguintes são discutidas algumas dessas temáticas.

5.3.1 Transações ACID

A partir do momento em que os bancos de dados começaram a ser usados em áreas mais críticas, como saúde e economia (UFRN, 2023), tornou-se essencial que os SGBDs pudessem garantir a confiabilidade e integridade dos dados. Diante dessa necessidade, um grupo de pesquisadores da IBM criou o conceito de estrutura ACID, sigla que faz referência aos requisitos fundamentais incorporados por esse conceito:

- Atomicidade: garante que todas as operações da transação sejam executadas sem erro, e caso ocorra, as demais operações devem ser revertidas;
- Consistência: garante que todos os comandos que devem ser realizados em uma operação sejam concluídos, a fim de deixar o banco sempre válido para futuras transações;
- Isolamento: garante que nenhuma transação possa interferir no funcionamento de outra, de maneira isolada;
- Durabilidade: garante que os dados armazenados estejam sempre disponíveis, mesmo em caso de falha de *hardware* ou de *software*.

5.3.2 Teorema de CAP

O Teorema de CAP é outro quesito bastante difundido dentre os DBA's e desenvolvedores, já que com ele é possível escolher um conjunto de características e padrões do seu sistema de acordo com a necessidade. A sigla CAP representa três características: consistência, disponibilidade (*Availability*) e partição.

- Consistência: capacidade de garantir que eventuais réplicas de dados em um sistema, devem estar atualizados simultaneamente;
- Disponibilidade: capacidade do sistema de atender às necessidades e requisições a ele enviadas;
- Partição: a capacidade do sistema de tolerar o particionamento dos dados.

O teorema afirma que é impossível ter mais de duas das três definições acima e, diante disso, torna-se necessário analisar cuidadosamente os requisitos do sistema de acordo com cada necessidade. As possibilidades são (SANTOS, 2017):

- CA: o sistema está sempre disponível e as réplicas de dados são equivalentes entre si, mas sem tolerância a particionamento dos dados;
- AP: o sistema está sempre disponível e os dados podem ser particionados, porém, pode haver inconsistência entre réplicas de dados, já que podem estar em locais distintos.
- CP: a consistência entre réplicas de dados é garantida e os dados podem ser particionados, porém, há a possibilidade do sistema estar indisponível.

5.3.3 Cargas de trabalho e balanceamento de cargas

Cargas de trabalho (*workloads*) referem-se às tarefas a serem executadas, requisições a serem processadas ou dados a serem armazenados por sistemas de computação. Em aplicações web, as cargas de trabalho podem variar entre buscas em bancos de dados, manipulações, processamento, transações, dentre outras, estando diretamente relacionadas ao número de acessos e ao foco da aplicação. Por exemplo, no contexto de redes sociais, em geral, a maioria das cargas de trabalho está relacionada ao armazenamento e processamento de dados, visto que possuem uma grande quantidade de usuários, conteúdos e um alto tempo de atividade (VOLPATO, 2023). Por outro lado, em aplicações relacionadas à venda, o foco pode estar mais ligado às transações e consultas de dados, uma vez que lojas realizam vendas sobre uma gama de produtos e/ou serviços.

O balanceamento de cargas tem o intuito de otimizar a distribuição de recursos e lidar de forma eficiente com as requisições de usuários e aplicações. É nesse processo que se analisa a natureza da aplicação envolvida e, a partir disso, identifica-se a melhor maneira de realizar as operações. Existem diversas maneiras de realizar o balanceamento e, como dito anteriormente, tudo dependerá do foco e do tamanho da aplicação. No cenário de armazenamento de dados, as opções podem variar entre particionamento, distribuição geográfica e uso de algoritmos especialmente criados para otimização de cargas de trabalho.

Elevadas cargas de trabalho em SGBDs, associadas à falta ou ao mau planejamento do balanceamento de cargas, geram ineficiência no uso dos recursos disponíveis, acarretando congestionamento, que implica lentidão e, no limite, recusa de atendimento às requisições, resultando em baixa eficiência e qualidade do *software*.

5.4 Uma visão geral sobre MySQL

Criado em 1995 pelos engenheiros suecos, Michael Widenius e David Axmarka (MYSQL, 2023), o `MySQL` está entre os mais populares sistemas de geren-

ciamento de banco de dados do tipo relacional (DBENGINES, 2023). Em sua concepção inicial, foi criado para ser um SGBD de uso pessoal, mas devido a sua facilidade de uso, simplicidade e disponibilidade, além de ser um *software* de código aberto, logo foi comercializado e, em 2008, comprado pela *Oracle Corporation*, empresa à qual pertence até os dias de hoje.

O *software* está atualmente na versão 8.0 e desde sua criação foram implementadas diversas funcionalidades que justificam a sua popularidade atemporal (MYSQL, 2023). O `MySQL` possui suporte a *multithreading* e isso permite que diferentes *back-ends* acessem a mesma instância de um banco de dados e visualizem o mesmo conjunto de dados. No contexto de facilidades para uso em aplicações web, as características que o diferenciam de outros SGBD's são: suporte à multiplataformas, recursos de gerenciamento de segurança, escalabilidade, suporte à Transações ACID e suporte à computação em nuvem. Além disso, existem alguns objetos que permitem a execução de ações específicas em resposta a consultas ou eventos, que podem ser procedimentos, funções, eventos, gatilhos ou *views*.

5.4.1 Ferramenta de Gerenciamento do MySQL

Existem alguns programas e recursos fornecidos pela *Oracle* a fim de facilitar a vida dos usuários do `MySQL`, e o mais conhecido é o `MySQL Workbench`. O `MySQL Workbench` é um *software* de interface gráfica que possui diversas funcionalidades para simplificar e tornar intuitiva a implantação e gestão de bancos de dados ou servidores `MySQL` (MYSQL, 2023), sendo possível:

- Projetar: possibilita que o *Database Administrator* (DBA) projete, gere, configure e modele seu banco de dados de maneira visual. O *software* tem disponíveis vários tipos de modelagens em seu conjunto de ferramentas, que vão desde engenharia direta e reversa até mesmo modelos complexos de Entidade Relacionamento (ER);

- Desenvolver: fornece diversas ferramentas para fins de criação, otimização e execução de consultas, possuindo um editor SQL com algumas funcionalidades que facilitam o trabalho do desenvolvedor, tais como: preenchimento automático, histórico de execução, reutilização de código e realce de cores por sintaxe;

- Administrar: apresenta um menu com diversas funcionalidades ligadas diretamente à administração e configuração de bancos de dados e servidores, permitindo configurar servidores, administrar usuários, fazer *backup* e auditoria dos dados de maneira visual e simplificada.

Além dessas funcionalidades, o `MySQL Workbench` possui uma gama de ferramentas que permite melhorar o desempenho dos aplicativos `MySQL` a partir de indicadores e migrar facilmente um banco de dados de um outro SGBD relacional para o `MySQL`.

5.4.2 Arquitetura do MySQL

A arquitetura de um SGBD descreve como os componentes deste *software* estão relacionados e como os fluxos funcionam no fornecimento dos serviços. O `MySQL` é baseado na arquitetura cliente-servidor, cuja principal característica é a divisão de funções em duas partes: a do cliente e a do servidor. A Figura 5.5 esboça o funcionamento de uma arquitetura cliente-servidor. Nesse exemplo, três clientes, por meio de requisições, solicitam serviços ou recursos ao servidor, que as processa e devolve as respostas aos clientes.

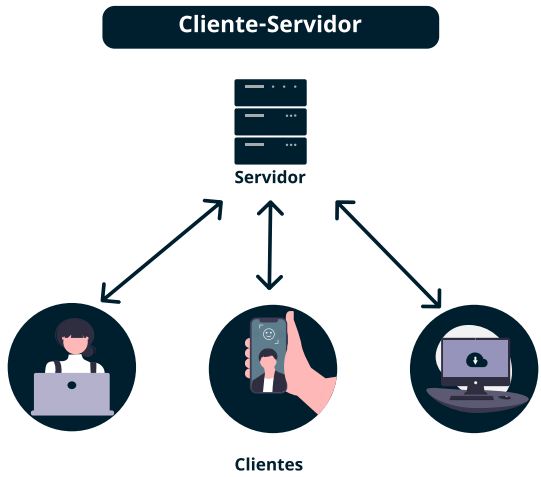


Figura 5.5: Ilustração da arquitetura cliente-servidor

Fonte: Adaptado de (WIKIPéDIA, 2023)

A camada cliente tem a função de fornecer uma interface amigável ao usuário, permitindo que este realize suas requisições ao servidor. É importante salientar que quando a consulta é realizada, existe um processo de validação de sintaxe e semântica, e caso tudo esteja correto, o resultado é obtido. Além disso, existe um conjunto de recursos e serviços nesta camada, que são: manipulação de conexão, autenticação e segurança.

- Manipulação de conexão: é o processo em que o cliente estabelece uma comunicação com o servidor por meio de uma *thread*, e toda a troca de informação é realizada por meio desta;
- Autenticação: é o processo realizado por parte do servidor para verificar se o cliente é válido, e geralmente essa verificação é realizada por meio de *login* e senha;

- Segurança: é o processo em que o servidor verifica se o usuário/cliente conectado tem permissão para realizar as consultas e manipulações solicitadas.

A camada servidor recebe as requisições do cliente, as processa e retorna à camada cliente o resultado demandado pelo usuário. E assim como na camada cliente, existe um conjunto de serviços que são realizados nesta camada, que são: tratamento de *thread, parser*, otimizador, cache de consulta, *buffer*, cache de metadados e cache da chave.

- Tratamento de *thread*: é onde acontece o processo de fornecer e assegurar a conexão cliente-servidor por meio de uma *thread*.
- Parser: é uma etapa que recebe a consulta a ser executada pelo SGBD e efetua uma análise semântica e sintática do código enviado pelo cliente. Caso esteja tudo correto, a consulta é separada em *tokens* que servirão para montar uma árvore de análise e a retornar para o próximo serviço;
- Otimizador: recebe como entrada a árvore de análise e realiza algumas modificações, se necessárias, a fim de melhorar a consulta;
- Cache de consulta: é um recurso que serve para verificar se a consulta que entrou para ser executada já foi realizada anteriormente. Caso já tenha sido, ela retorna a resposta sem a necessidade de passar pelas etapas de análise, *parse* e otimização;
- *Buffer* e *cache*: são espaços de memória em que estão armazenados os dados frequentemente solicitados e acessados. No entanto, enquanto o *buffer* é usado para armazenar os dados que foram recentemente acessados, o cache armazena dados que provavelmente serão acessados em breve e para o armazenamento destes é levado em consideração as consultas realizadas anteriormente;
- Cache de metadados: é um espaço de memória usado para rastrear informações de estruturas de armazenamento de dados, como bancos de dados e objetos. Ele contém informações como o nome das tabelas, colunas, índices e chaves primárias;
- Cache de chave: é um tipo de identificador para uma estrutura de dados que está dentro do cache de metadados. Ele é usado para recuperar informações específicas sobre a estrutura de dados armazenada no cache de metadados.

5.4.3 Instalação e Configuração

Com intuito de demonstrar como o `MySQL` pode ser utilizado, será mostrada uma sequência de passos para a instalação e configuração do mesmo. Como já dito anteriormente, o `MySQL` é um SGBD multiplataforma, ou seja, pode ser instalado em qualquer SO, seja Windows, Linux ou MacOS (MYSQL, 2023). Nesta

seção é realizada a instalação do `MySQL` Community e `MySQL` Workbench, além da configuração de um servidor `MySQL` numa máquina Windows.

1. Para efetuar o *download* do `MySQL` é preciso acessar o link [3] do site oficial. Ao acessar a página de *download*, é necessário escolher a versão do SO em que será instalado o SGBD e, após isso, clicar em *download* conforme mostrado na Figura 5.6.

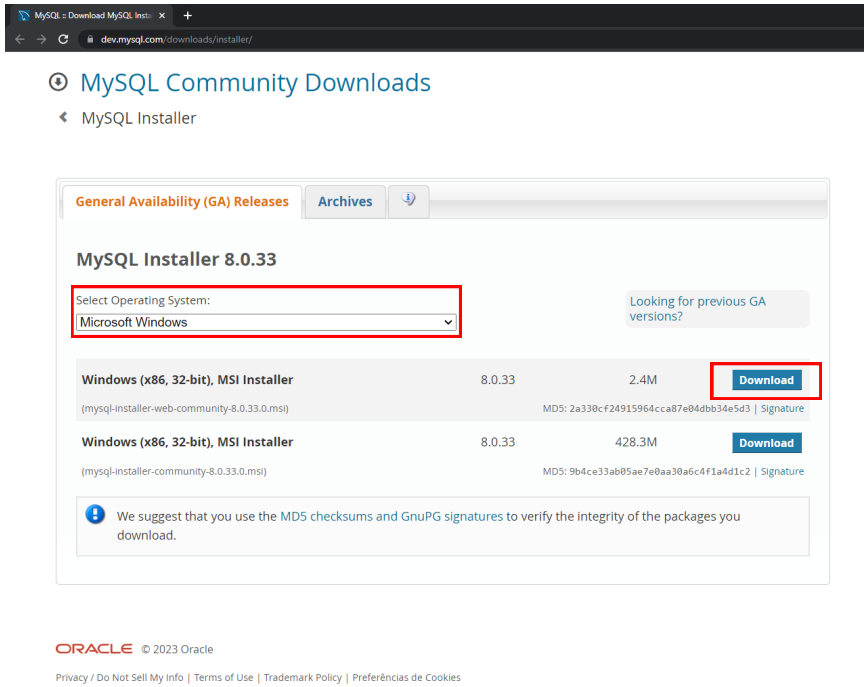


Figura 5.6: Página inicial para download do `MySQL`

2. A Figura 5.7 demonstra a página de *login* para a conta *Oracle*. O *login* é opcional. Para continuar o *download* basta clicar em *No thanks, just start my download* (Não, obrigado, apenas inicie meu download).

[3] Download `MySQL`



Figura 5.7: Página de login *Oracle account*

3. Após a conclusão do *download*, basta clicar no instalador baixado e iniciar o processo de instalação. A primeira etapa é a escolha do tipo de configuração. Recomenda-se clicar em *Custom*, para customizar os programas que serão instalados juntos ao `MySQL` e depois em *Next*.

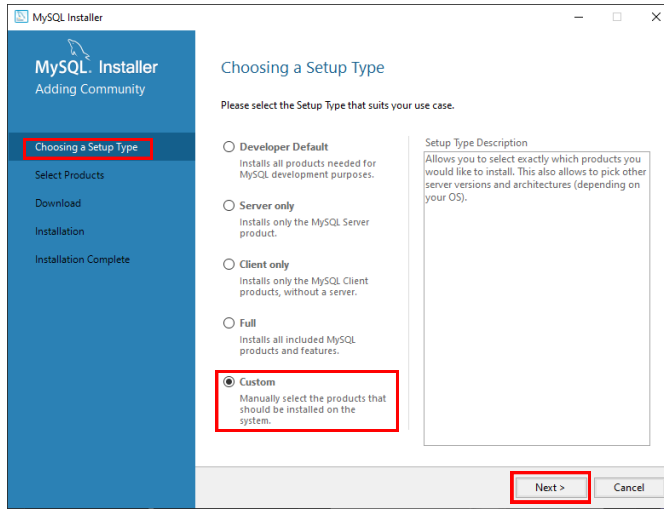


Figura 5.8: Tela de escolha do tipo de instalação do `MySQL`

4. A Figura 5.9 demonstra a etapa em que ocorre a seleção dos programas e utilitários a serem instalados junto com o `MySQL`. Recomenda-se que iniciantes escolham as versões mais recentes do `MySQL Server` e do `MySQL Workbench` . Em seguida, deve-se seguir a sequência de cliques: *Next* `->` *Execute* `->` *Next*.

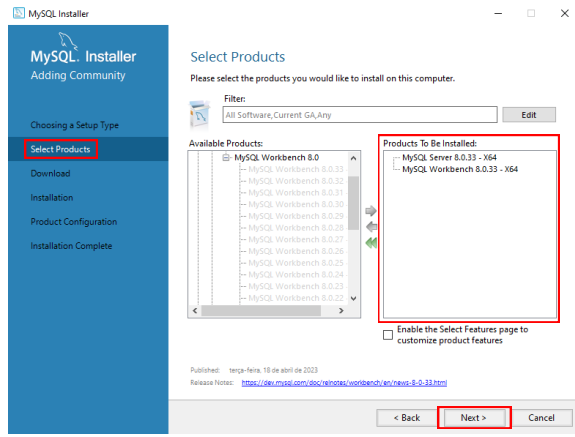


Figura 5.9: Tela de escolha dos utilitários do `MySQL`

5. A Figura 5.10 ilustra a etapa de configuração de rede. Nesta, não é necessário modificar nada, apenas clicar em *Next*.

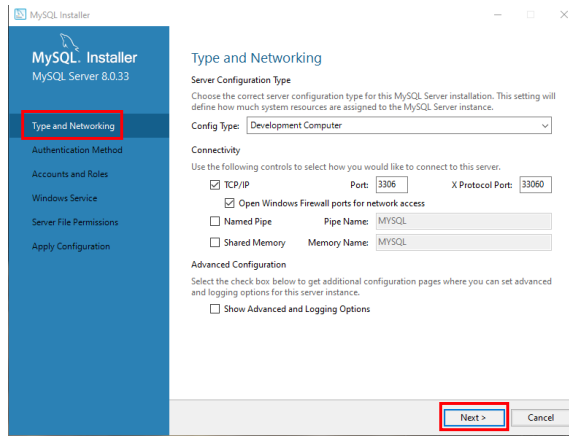


Figura 5.10: Tela de configurações de rede do `MySQL`

6. Na etapa de *Authentication Method*, cuja tela é mostrada na Figura 5.11, são realizadas as configurações de autenticação. Nesta, o recomendado é marcar a

primeira opção *Use strong Password Encryption Authentication* e, após, clicar em *Next*.

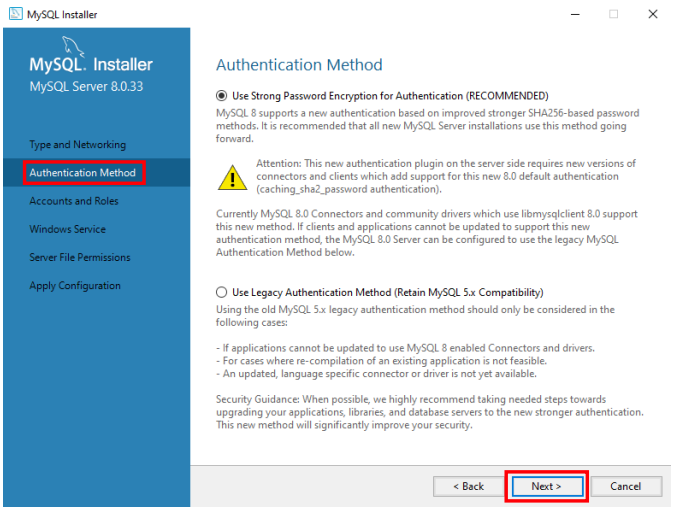


Figura 5.11: Tela de configurações de autenticação do `MySQL`

7. Na etapa de *Accounts and Roles*, exemplificada pela Figura 5.12, são realizadas as configurações de usuário e senha. Nesta etapa é recomendado criar uma senha para o usuário *root*e, em seguida, clicar em *Next*. Nas próximas etapas não é necessário modificar nada, apenas confirmar as informações e continuar clicando em *Next* e *Execute* até finalizar.

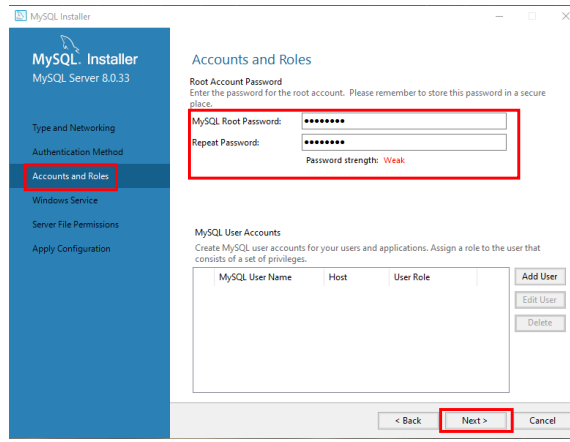


Figura 5.12: Tela de configuração de contas e usuários do `MySQL`

8. Com o `MySQL Workbench` em execução, a primeira ação é criar um servidor para o banco de dados. Para isto, basta clicar no ícone de "mais"ao lado de `MySQL` *Connections*, conforme destacado na Figura 5.13.

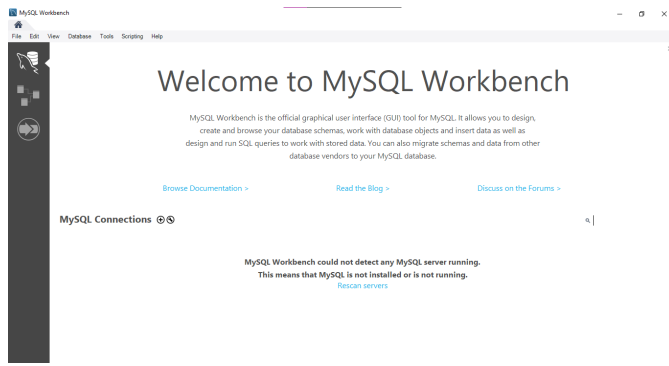


Figura 5.13: Tela inicial `MySQL` Workbench

9. Para configurar o servidor basta preencher as lacunas de *Connection Name* e *HostName*, destacadas na Figura 5.14. Como o banco de dados será local, *HostName* deve ser preenchido com *localhost* e a senha será a chave já configurada no passo 7. Após preencher tudo, basta clicar em *Ok*.

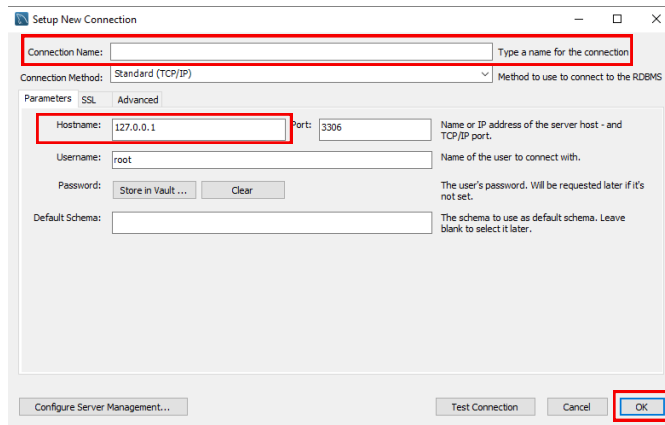


Figura 5.14: Tela de configuração do servidor

10. Para acessar o servidor criado, basta, na página inicial do `MySQL Workbench`, clicar no servidor e preencher os dados de *login*. A Figura 5.15 apresenta a tela inicial do `MySQL Workbench` com servidor criado.

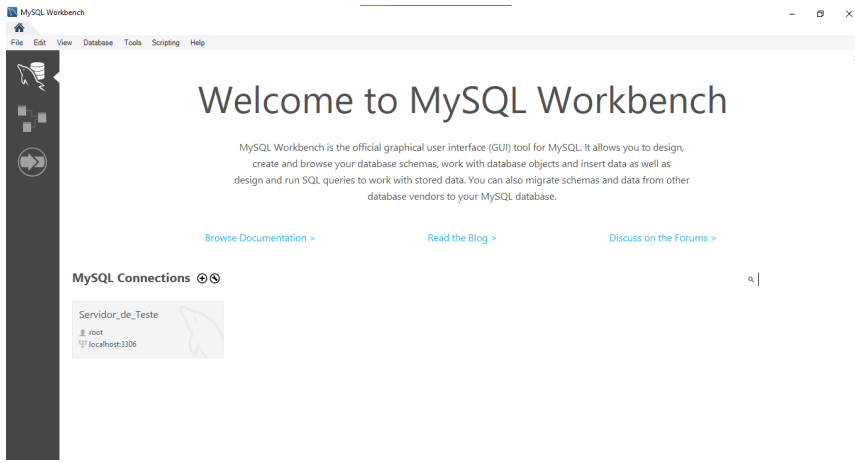


Figura 5.15: Tela inicial `MySQL` Workbench com o servidor criado

11. A Figura 5.16 ilustra a tela de gerenciamento e criação de banco de dados do `MySQL Workbench` . É no *script* em que são escritas as consultas e manipulações.

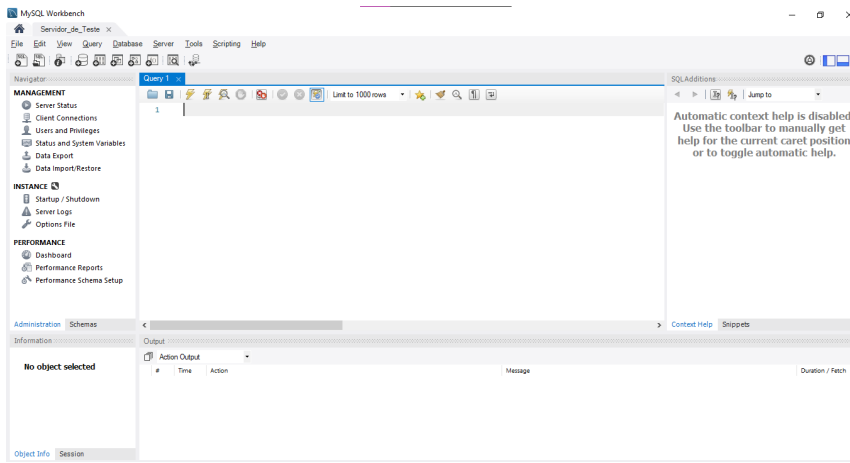


Figura 5.16: Tela de gerenciamento de bancos de dados do `MySQL` Workbench

5.4.4 Linguagem de Consulta

Assim como outros SGBDs, o `MySQL` usa uma linguagem de consulta para realizar suas buscas, manipulações e configurações, e esta linguagem é o SQL. Como já citado na Seção 5.2, o SQL é uma linguagem de consulta projetada para bancos de dados relacionais e que tem como base as operações da Álgebra Relacional. A linguagem é dividida em cinco blocos de comandos, conforme mostrado na Figura 5.17, no qual cada subdivisão possui um propósito, que será demonstrado na sequência.

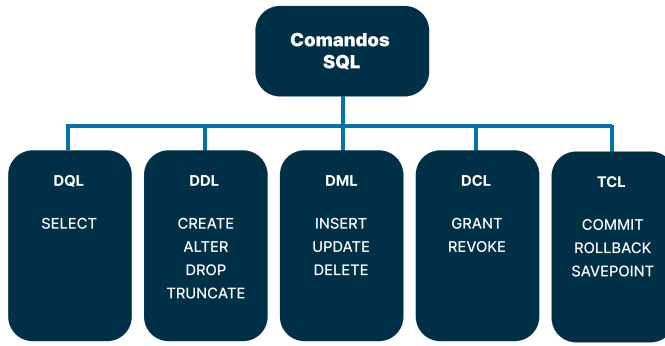


Figura 5.17: Comandos SQL

Fonte: Adaptado de (DICODING, 2023)

***Data Query Language* (DQL)**: é o subconjunto da linguagem SQL responsável por definir comandos para consulta de dados armazenados em um banco de dados. Este conjunto inclui apenas o comando *SELECT*, que é usado para resgatar dados de tabelas e outras fontes de dados. A Listagem 5.4 exemplifica a seleção da coluna "alunos"dentro da tabela "estudantes".

```
1 SELECT aluno FROM estudantes;
```

Listagem 5.4: Exemplo de uso do *SELECT*

Como o SQL é uma linguagem voltada para bancos de dados relacionais, há algumas operações e cláusulas que aproveitam essas relações para obter resultados mais precisos com o *SELECT*. Uma dessas cláusulas é o *JOIN*, que permite combinar colunas ou entidades com base em critérios específicos. Existem quatro variações principais de *JOIN*, cada uma com seu propósito e resultado (MYSQL, 2023). A seguir, será introduzida uma breve explicação sobre cada uma delas, e as Tabelas 5.2 e 5.3 representam duas entidades a fim de facilitar o entendimento dos exemplos a seguir:

Tabela 5.2: Tabela que representa a entidade *empresas*

empresas			
IdEmpresa	**IdCliente**	**Nome**	**Cidade**
1	1	Empresa1	Hidrolândia
2	3	Empresa2	Sobral
3	3	Empresa3	Ipueiras
4	4	Empresa4	Santa Quitéria

Tabela 5.3: Tabela que representa a entidade *clientes*.

clientes		
IdCliente	**NomeCliente**	**Telefone**
1	Jannielly	(11) 11111-1111
2	Klayver	(22) 22222-2222
3	Ana Lara	(33) 33333-3333
4	Maria Rita	(44) 44444-4444

INNER JOIN: essa cláusula seleciona somente os registros que correspondem as condições passadas no bloco de comando, ou seja, devolve como resposta da consulta apenas as informações que possuem os valores comuns nas tabelas envolvidas. A Listagem 5.5 representa a junção das informações das Tabelas 5.2 e 5.3 apenas onde as linhas corresponderem a condição da diretiva *ON*. A Tabela 5.4 ilustra o resultado da operação.

```
1 SELECT *
2 FROM empresas
3 INNER JOIN clientes
4 ON empresas.IdCliente = clientes.IdCliente;
```

Listagem 5.5: Exemplo de uso do *INNER JOIN*

CROSS JOIN: essa cláusula realiza o produto cartesiano entre tabelas, ou seja, dada duas tabelas o resultado será um produto cartesiano entre as tabelas envolvidas. A Listagem 5.6 representa o produto cartesiano entre as tabelas *clientes* e *empresas*. A Tabela 5.5 mostra o resultado da operação.

Tabela 5.4: Resultado do *INNER JOIN*

IdEmpresa	**IdCliente**	**Nome**	**Cidade**	**IdCliente**	**NomeCliente**	**Telefone**
1	1	Empresa1	Hidrolândia	1	Jannielly	(11) 11111-1111
2	3	Empresa2	Sobral	3	Ana Lara	(33) 33333-3333
3	3	Empresa3	Ipueiras	3	Ana Lara	(33) 33333-3333
4	4	Empresa3	Ipueiras	4	Maria Rita	(44) 44444-4444

Tabela 5.5: Resultado do *CROSS JOIN*

IdEmpresa	IdCliente	Nome	Cidade	IdCliente	NomeCliente	Telefone
1	1	Empresa1	Hidrolândia	1	Jannielly	(11) 11111-1111
2	3	Empresa2	Sobral	1	Jannielly	(11) 11111-1111
3	3	Empresa3	Ipueiras	1	Jannielly	(11) 11111-1111
4	4	Empresa4	Santa Quitéria	1	Jannielly	(11) 11111-1111
1	1	Empresa1	Hidrolândia	2	Klayver	(22) 22222-22222
2	3	Empresa2	Sobral	2	Klayver	(22) 22222-22222
3	3	Empresa3	Ipueiras	2	Klayver	(22) 22222-22222
4	4	Empresa4	Santa Quitéria	2	Klayver	(22) 22222-22222
1	1	Empresa1	Hidrolândia	3	Ana Lara	(33) 33333-3333
2	3	Empresa2	Sobral	3	Ana Lara	(33) 33333-3333
3	3	Empresa3	Ipueiras	3	Ana Lara	(33) 33333-3333
4	4	Empresa4	Santa Quitéria	3	Ana Lara	(33) 33333-3333
1	1	Empresa1	Hidrolândia	4	Maria Rita	(44) 44444-4444
2	3	Empresa2	Sobral	4	Maria Rita	(44) 44444-4444
3	3	Empresa3	Ipueiras	4	Maria Rita	(44) 44444-4444
4	4	Empresa4	Santa Quitéria	4	Maria Rita	(44) 44444-4444

```
SELECT *
FROM empresas
CROSS JOIN clientes;
```

Listagem 5.6: Exemplo de uso do *CROSS JOIN*

LEFT JOIN: essa cláusula retorna todas as linhas da tabela que estão à esquerda e as linhas da tabela à direita que obedecerem à diretiva ON. Caso não haja nenhuma correspondência na tabela da direita com quaisquer tuplas na tabela da esquerda, tais tuplas aparecerão no resultado com valores *NULL* para os atributos da tabela da direita. A Listagem 5.7 representa o uso do *LEFT JOIN* que retorna todas as linhas da tabela *empresas* e as linhas da tabela *clientes* que obedecerem à condição na diretiva *ON*. A Tabela 5.6 mostra o resultado da operação.

```
SELECT *
FROM empresas
LEFT JOIN clientes
ON empresas.IdCliente = clientes.IdCliente;
```

Listagem 5.7: Exemplo de uso do *LEFT JOIN*

RIGHT JOIN: essa cláusula retorna todas as linhas da tabela que estão a direita e as linhas da tabela a esquerda que obedecerem à diretiva ON. Caso não haja nenhuma correspondência na tabela da esquerda com quaisquer tuplas na tabela da direita, tais tuplas aparecerão no resultado com valores *NULL* para os atributos da tabela da esquerda. A Listagem 5.8 representa o uso do *RIGHT JOIN* que retorna todas as linhas da tabela empresas e as linhas da tabela clientes que obedecem à condição na diretiva *ON*. Como resultado da operação pode-se obter a Tabela 5.7.

Tabela 5.6: Resultado do *LEFT JOIN*

IdEmpresa	IdCliente	Nome	Cidade	IdCliente	NomeCliente	Telefone
1	1	Empresa1	Hidrolândia	1	Jannielly	(11) 11111-1111
2	3	Empresa2	Sobral	3	Ana Lara	(33) 33333-3333
3	3	Empresa3	Ipueiras	3	Ana Lara	(33) 33333-3333
4	4	Empresa4	Santa Quitéria	4	Maria Rita	(44) 44444-4444

Tabela 5.7: Resultado da operação com *RIGHT JOIN*

IdEmpresa	IdCliente	Nome	Endereço	IdCliente	NomeCliente	Telefone
1	1	Empresa1	Hidrolândia	1	Jannielly	(11) 11111-1111
NULL	NULL	NULL	NULL	2	Klayver	(22) 22222-22222
2	3	Empresa2	Sobral	3	Ana Lara	(33) 33333-3333
3	3	Empresa3	Ipueiras	3	Ana Lara	(33) 33333-3333
4	4	Empresa4	Santa Quitéria	4	Maria Rita	(44) 44444-4444

```
1 SELECT *
2 FROM empresas
3 RIGHT JOIN clientes
4 ON empresas.IdCliente = clientes.IdCliente;
```

Listagem 5.8: Exemplo de uso do *RIGHT JOIN*

Além das operações comuns e das junções, existem recursos avançados que podem ser utilizados dentro de um sistema de banco de dados, como as *PROCEDURES* e os *TRIGGERS*. As *PROCEDURES* são conjuntos de comandos que são executados de forma sequencial. Caso uma linha de comando não seja executada corretamente, as operações anteriores não são revertidas e a execução da *PROCEDURES* é encerrada. A Listagem 5.9 representa a estrutura de uma *PROCEDURE*.

```
1 CREATE PROCEDURE nome_dado_a_procedure
2 BEGIN
3     -- comandos SQL aqui
4 END;
```

Listagem 5.9: Exemplo de uso da *PROCEDURES*

Os *TRIGGERS* são objetos que estão diretamente relacionados a uma ou mais entidades. Estes objetos servem como "ouvintes"de um determinado evento e realizam alguma operação quando identificam o evento especificado. A Listagem 5.10 representa uma *TRIGGER* que observa quando o evento de *INSERT* na tabela *clientes* é executado. Quando tal evento acontecer, a ação de adicionar este novo cliente a tabela *clienteEmpresa* é executada.

```
CREATE TRIGGER adc_cliente_empresa
AFTER INSERT ON clientes
FOR EACH ROW
BEGIN
    INSERT INTO clienteEmpresa (id_cliente, nome_cliente)
    VALUES (NEW.id_cliente, NEW.nome_cliente);
END;
```

Listagem 5.10: Exemplo de uso da *TRIGGER*

***Data Definition Language* (DDL)**: é o subconjunto da linguagem SQL responsável por fazer a definição do banco de dados. Este conjunto inclui quatro comandos: *CREATE, ALTER, DROP e TRUNCATE.*
CREATE: usado para criar estruturas dentro de banco de dados. A Listagem 5.11 representa a criação de uma tabela chamada "estudantes".

```
CREATE TABLE Estudantes (
ID INT PRIMARY KEY AUTO_INCREMENT,
Nome VARCHAR(255),
Idade INT,
Endereco VARCHAR(255);
```

Listagem 5.11: Exemplo de uso do *CREATE*

ALTER: usado para alterar estruturas no banco de dados. A Listagem 5.12 apresenta uma alteração na tabela "estudantes". Nesse caso, está sendo adicionada uma nova coluna chamada "telefone".

```
ALTER TABLE estudantes ADD telefone VARCHAR(20);
```

Listagem 5.12: Exemplo de uso do *ALTER*

DROP: usado para apagar estruturas dentro de banco de dados. A Listagem 5.13 representa a deleção da estrutura da tabela "estudantes".

```
DROP TABLE estudantes;
```

Listagem 5.13: Exemplo de uso do *DROP*

TRUNCATE: usado para excluir os dados de estruturas dentro de banco de dados de forma permanente. A Listagem 5.14 representa uma alteração na tabela estudantes. Nesse caso todos os dados da tabela "estudantes"estão sendo deletados.

```
TRUNCATE TABLE estudantes;
```

Listagem 5.14: Exemplo de uso do *TRUNCATE*

***Transaction Manipulation Language* (DML)**: é o subconjunto da linguagem SQL responsável por manipular as informações de banco de dados. Este conjunto inclui três comandos: *INSERT, UPDATE e DELETE.*
INSERT: usado para inserir novos dados em uma ou mais tabelas. A Listagem 5.15 demonstra a inserção de um novo dado a tabela de "estudantes".

```
INSERT INTO estudantes (matricula, nome_completo, telefone)
VALUES (1234, Jannielly, 8899999999);
```

Listagem 5.15: Exemplo de uso do *INSERT*

UPDATE: usado para atualizar dados em uma ou mais tabelas. A Listagem 5.16 demonstra a atualização do atributo "nome"na tabela de "estudantes"com base na condição *WHERE.*

```
UPDATE estudantes
SET nome = 'Francisca Jannielly Garcia da Costa'
WHERE matricula = 66666;
```

Listagem 5.16: Exemplo de uso do *UPDATE*

DELETE: usado para deletar dados de uma estrutura. Diferentemente do *DROP* que apaga a estrutura e seus dados ou do *TRUNCATE* que deleta os dados de maneira permanente, o comando *DELETE*, permite a recuperação dos dados. A Listagem 5.17 representa a exclusão dos dados da tabela "estudantes".

```
DELETE FROM estudantes;
```

Listagem 5.17: Exemplo de uso do *DELETE*

***Data Control Language* (DCL)**: é o subconjunto da linguagem SQL relacionado à segurança do banco de dados. Este conjunto inclui 2 comandos: *GRANT* e *REVOKE.*
GRANT: usado para conceder permissão de acesso ou de manipulação a um objeto do banco de dados a um usuário específico. A Listagem 5.18 representa a permissão para o usuário "Priscila"consultar a tabela "estudantes".

```
GRANT SELECT ON estudantes TO Priscila;
```

Listagem 5.18: Exemplo de uso do *GRANT*

REVOKE: usado para revogar a permissão de acesso ou de manipulação de um objeto do banco de dados a um usuário. A Listagem 5.19 representa a permissão para o usuário "Jander"consultar a tabela "estudantes".

```
1 REVOKE SELECT ON estudantes FROM Jander;
```

Listagem 5.19: Exemplo de uso do *REVOKE*

***Transaction Control Language* (TCL)**: é o subconjunto da linguagem SQL relacionado às transações em um banco de dados. Este conjunto inclui três comandos: *COMMIT, ROLLBACK e SAVEPOINT*.
COMMIT: usado para garantir que as alterações foram realizadas em uma transação, tornando-as persistentes no banco de dados. A Listagem 5.20 representa a transação de deletar o estudante com a matrícula 12345, neste caso, o *COMMIT* garante que as alterações da transação foram realizadas.

```
1 BEGIN TRANSACTION
2     DELETE FROM estudantes
3     WHERE matricula = 12345;
4 COMMIT;
```

Listagem 5.20: Exemplo de uso do *COMMIT*

ROLLBACK: usado para reverter uma transação, ou seja, ao usar *ROLLBACK* é recuperado o estado do banco de dados até o último *COMMIT* que foi realizado. A Listagem 5.21 representa o *ROLLBACK* para reverter uma determinada transação dado que houve um erro.

```
1 -- suponha que aqui houve um erro
2 ROLLBACK;
```

Listagem 5.21: Exemplo de uso do *ROLLBACK*

SAVEPOINT: usado para definir um ponto na transação em que os dados e alterações são salvas. Este comando é bem útil para garantir o versionamento dos dados. A Listagem 5.22 representa a configuração de um ponto de salvamento dentro de uma transação, ou seja, tudo feito antes está armazenado e denominado de "*ponto1*".

```
1  BEGIN TRANSACTION;
2     GRANT SELECT ON estudantes TO Priscila;
3     SAVEPOINT ponto1;
4     DELETE FROM clientes WHERE id = 2;
5 COMMIT
```

Listagem 5.22: Exemplo de uso do *SAVEPOINT*

Tabela 5.8: Comparação Banco de dados relacionais vs. Banco de dados não relacionais (MongoDB)

Banco de Dados Relacional	Banco de Dados Não Relacional
Base de dados	Base de dados
Tabela	Coleção
Registro	Documento
Coluna	Atributo

5.5 Uma visão geral sobre MongoDB

MongoDB é um SGBD orientado a documentos e é o mais utilizado quando se trata de banco de dados não relacional para aplicações web (DBENGINES, 2023). Criado em 2007, pela então empresa *10gen*, hoje chamada de MongoDB, teve como primeira versão o modelo de plataforma em nuvem para banco de dados, e sua idealização deu-se pela necessidade de trabalhar com os diferentes tipos de dados que a empresa lidava (MONGODB, 2023). Tendo o banco de dados relacional como exemplo de comparação, podemos entender que os dados são armazenados conforme mostra a Tabela 5.8:

No MongoDB, um documento é uma estrutura composta por pares de campo e valor, semelhante a um objeto JSON. Seus campos podem ser preenchidos por diversos tipos de dados, que podem ser desde um simples valor inteiro até um vetor de outros documentos. A Listagem 5.23 apresenta um exemplo de arquivo no formato especificado pelo MongoDB.

```
{
    "nome": "Francisca Jannielly Garcia da Costa",
    "idade":22,
    "escolaridade": "Ensino Superior incompleto",
    "caracteristicasFisicas":{
        "altura": 159,
        "peso":65,
        "raca":"Preta"
    }
 }
```

Listagem 5.23: Representação de um documento em MongoDB

O formato JSON oferece vantagens para o uso em documentos MongoDB. A primeira vantagem é que o JSON é o formato nativo para armazenamento de dados na maioria das linguagens de programação. Além disso, essa estrutura possibilita que algumas operações e técnicas sejam aplicadas, como junções e polimorfismo fluente (MONGODB, 2023).

Diferentemente do MySQL, que usa SQL para realizar consultas e operações com os dados do banco de dados, o MongoDB possui sua própria linguagem de consulta baseada em JavaScript. Embora possuam similaridades, em alguns aspectos a linguagem de consulta do MongoDB e a SQL se diferenciam totalmente, já que no MongoDB a sintaxe é baseada em objetos JSON e não em Tabelas.

5.5.1 Ferramenta de Gerenciamento do MongoDB

O MongoDB *Compass* é um *software* com interface gráfica que permite realizar operações e configurações em bancos de dados e servidores MongoDB. Este *software* é completamente gratuito e está disponível para execução em macOS, Windows e distribuições Linux. Assim como outros softwares de gerenciamento de banco de dados, o MongoDB *Compass* permite criar e gerenciar seu banco de dados de maneira simplificada, seja em sua máquina local ou em servidores na nuvem (MONGODB, 2023).

Caso possua um banco de dados do tipo relacional, com o MongoDB *Compass* é possível realizar a transferência de seu banco de dados para um modelo não relacional do tipo MongoDB, para isto, basta exportar seus dados em arquivos do tipo JSON e/ou *Comma-separated values* (CSV) e importar para o MongoDB *Compass* e realizar a configuração. Com os dados importados, o banco de dados está pronto para uso, permitindo realizar buscas configuradas, inserções, atualizações e deleções.

O pipeline de agregação do MongoDB possibilita a realização de operações de agregação e de consultas avançadas. Estes recursos permitem extrair e manipular os dados armazenados no banco de dados de forma eficiente. Para realizar as consultas e agregações, o MongoDB *Compass* possibilita que o usuário do MongoDB possa executar seus comandos diretamente em um ambiente JavaScript interativo, conhecido tecnicamente como (*shell*).

5.5.2 Instalação e Configuração

Com o intuito de demonstrar como o MongoDB pode ser utilizado, nesta seção é apresentada uma sequência de passos para sua instalação e configuração. Como já dito anteriormente, o MongoDB é um SGBD multiplataforma, ou seja, pode ser instalado em qualquer SO, seja Windows, Linux ou MacOS (MONGODB, 2023). Para esta seção será usado o MongoDB *Community Server*, será realizada a instalação do MongoDB *Compass* e a configuração do seu primeiro servidor numa máquina Windows.

1. Para fazer o *download* do MongoDB é preciso acessar o link [4] do site oficial. Ao acessar a página de *download* será possível escolher o SO em que ocorrerá a instalação. Após a escolha, basta clicar em *download* para baixar o MongoDB. A Figura 5.18 representa a tela inicial do site oficial do MongoDB.

[4] *Página de download do MongoDB*

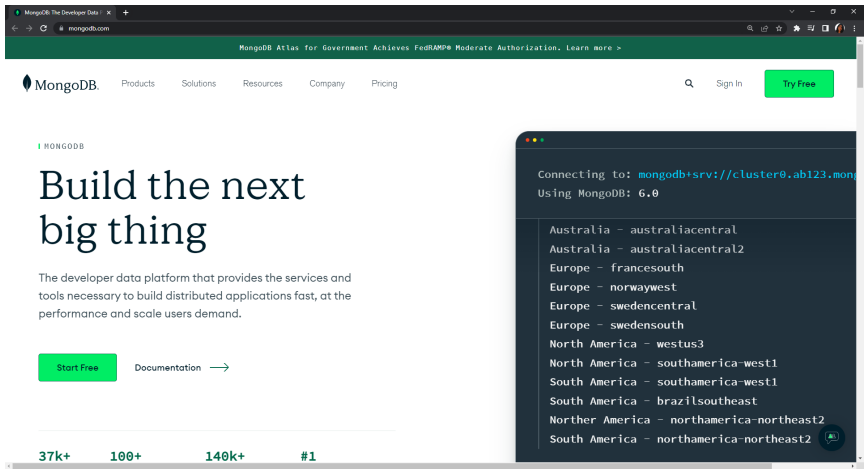


Figura 5.18: Página inicial `MongoDB`

2. Para ir para a página de *download* do `MongoDB` é necessário acessar, no menu superior, a aba *products* e clicar em *Community Server*, conforme mostra a Figura 5.19.

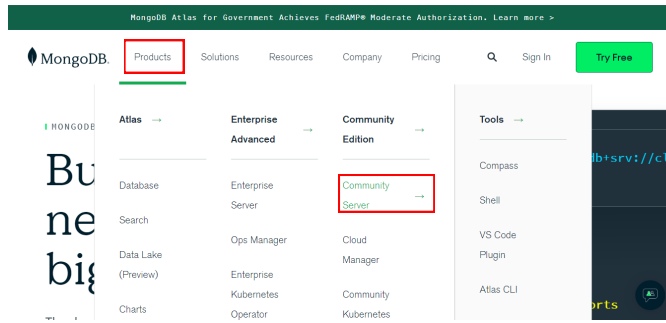


Figura 5.19: Página inicial do `MongoDB` com ênfase no menu

3. A Figura 5.20 representa a página de *download* do `MongoDB`. No final da página, é necessário selecionar a versão do `MongoDB`, o SO e o formato do instalador. Após preencher os três campos, basta clicar em *download*.

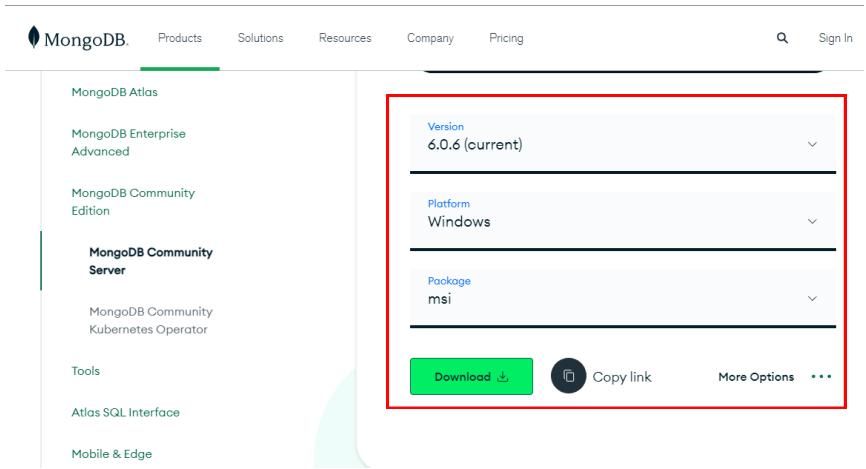


Figura 5.20: Página de download do `MongoDB Server`

4. Quando o *download* finalizar, basta clicar no instalador. A Figura 5.21 mostra a primeira tela do processo de instalação. Nesta, basta clicar em *Next*.

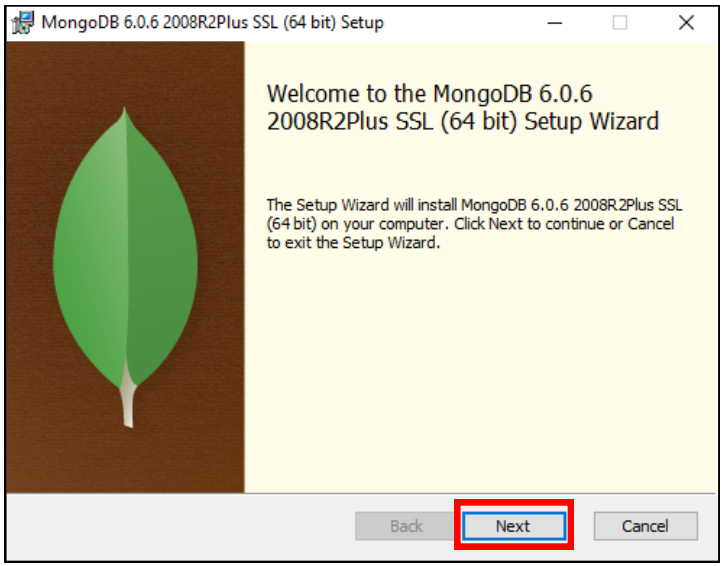


Figura 5.21: Primeiro passo da instalação do `MongoDB`

5. A Figura 5.22 representa a segunda tela do processo de instalação do `MongoDB`. Nesta tela, estão os termos de instalação. Para aceitar, basta marcar a caixinha *I accept the terms in the license Agreement*.

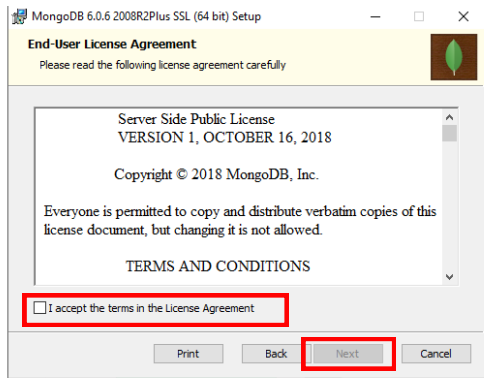


Figura 5.22: Tela de termos e licenças do `MongoDB`

6. A Figura 5.23 mostra a etapa de escolha do tipo de configuração. Nesta tela, haverá duas opções: a versão completa, que irá instalar todas as ferramentas disponíveis no `MongoDB`; e a versão customizada, na qual o usuário pode baixar apenas as ferramentas de que necessita. Para este trabalho, a escolha será pela versão completa.

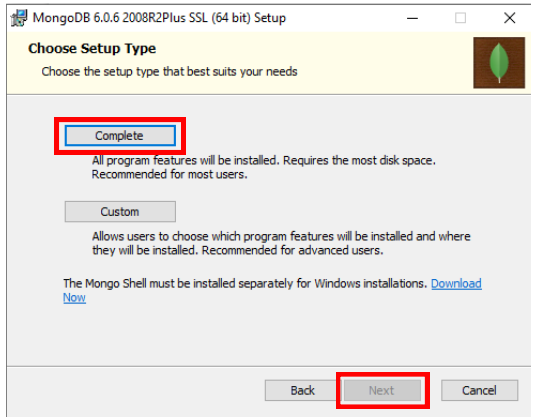


Figura 5.23: Tela de escolha do tipo de instalação do `MongoDB Server`

7. A Figura 5.24 representa a tela na qual é possível realizar as configurações do serviço Mongo, como adição de usuário, configuração de rede e local de instalação. Nesta etapa, basta clicar em *Next*.

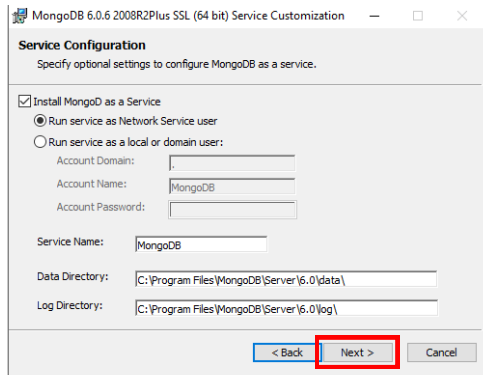


Figura 5.24: Tela de configuração de rede e usuário para o `MongoDB Server`

8. A etapa mostrada na Figura 5.25 é de bastante importância, pois é nela que será confirmada a instalação do `MongoDB` *Compass*, ferramenta já introduzida nesta seção. Após confirmar a instalação, basta clicar em *Next* e, em seguida, em *Finish*.

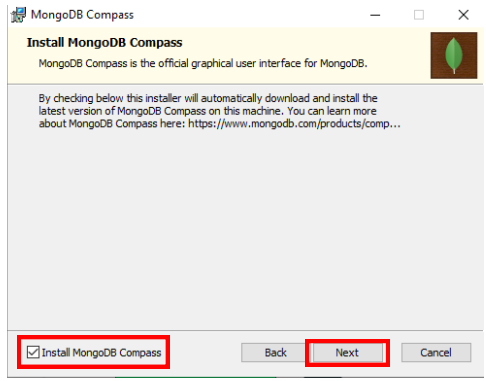


Figura 5.25: Tela de instalação do `MongoDB Compass`

9. A página inicial do `MongoDB` *Compass* dá acesso ao login no servidor no qual está o banco de dados. Para casos de uso de um servidor local, o caminho será: "`MongoDB`://localhost:27017/". Para conectar, basta clicar em *Connect*. A Figura 5.26 representa a tela de início do `MongoDB` *Compass*.

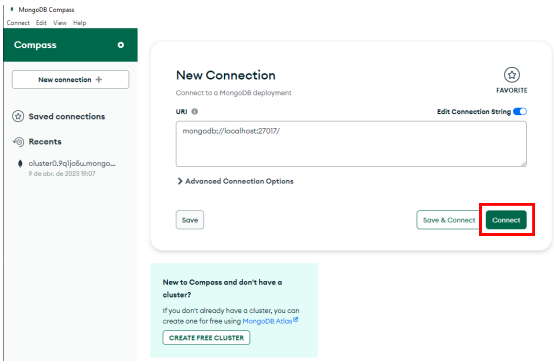


Figura 5.26: Tela inicial do `MongoDB Compass`

10. A Figura 5.27 mostra o ambiente do servidor conectado. Nesta página é possível criar bancos de dados, realizar consultas e fazer configurações de performance. A opção *create database* possibilita a criação de um banco de dados `MongoDB`.

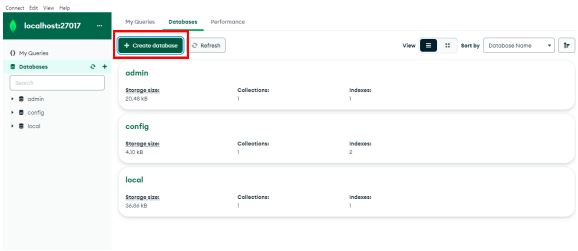


Figura 5.27: Tela de gerenciamento dos bancos de dados no `MongoDB Compass`

11. A Figura 5.28 mostra a tela de criação de um banco de dados usando o `MongoDB` *Compass*. Para criar, basta preencher os campos de *Database Name* e *Collection Name* e, em seguida, clicar em *Create Database*.

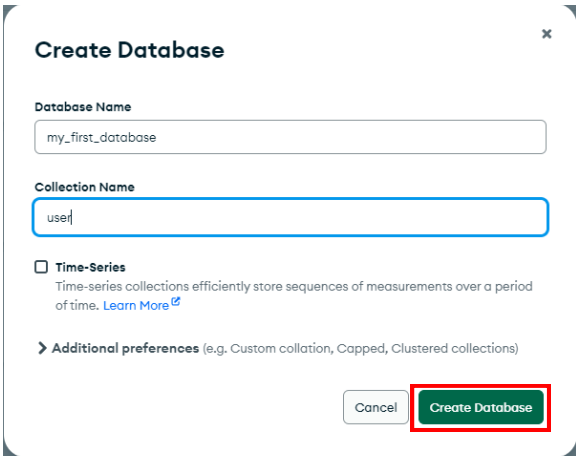


Figura 5.28: Tela de criação de um banco de dados no `MongoDB Compass`

12. A tela representada pela Figura 5.29 dá acesso ao banco de dados criado. Nessa tela, é possível importar, criar, manipular e consultar dados.

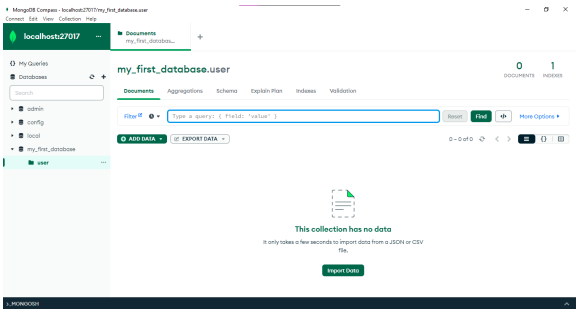


Figura 5.29: Tela de gerenciamento do banco de dados criado

5.5.3 Linguagem de Consulta

O `MongoDB` tem suporte a diversas linguagens de programação, como `C Sharp, JavaScript e C++`, mas a linguagem comumente utilizada é a própria linguagem de consulta, baseada em `JavaScript`. Como a SQL já foi introduzida na Seção 2.4, nesta seção apresenta-se uma comparação entre as duas linguagens, como ferramenta didática, para mostrar como funcionam as consultas e cláusulas no `MongoDB`.

O banco de dados utilizado nas explicações a seguir é denominado "db". Para criar uma nova coleção, que funciona como uma tabela em um banco de dados relacional, usa-se a função *createCollection()*. A Listagem 5.24 ilustra a criação de uma coleção de "users".

```
db.createCollection(users)
```

Listagem 5.24: Exemplo de uso do comando *createCollection*

Para listar todas as coleções associadas ao banco de dados, usa-se o comando *show*. A Listagem 5.25 mostra todas as coleções associadas aos *users*.

```
show collections
```

Listagem 5.25: Exemplo de uso do comando *show*

Para adicionar um documento à coleção *users*, que seria como adicionar uma linha em uma tabela, usa-se a função *insert()*. A listagem 5.26 representa a adição dos atributos "nome", "idade"e "sexo"à tabela *users*.

```
db.users.insert({
    "name": "Jannielly",
    "idade": 22,
    "sexo": "Feminino"
})
```

Listagem 5.26: Exemplo de uso do comando *insert*

A função *insertMany()* realiza a inserção de mais de um documento por vez. A Listagem 5.27 representa a adição múltipla dos atributos "nome", "idade"e "sexo"à coleção *users*.

```
db.users.insertMany({
    "name": "Jannielly",
    "idade": 22,
    "sexo": "Feminino"
},
{
    "name": "Ana Lara",
    "idade": 20,
    "sexo": "Feminino"
})
```

Listagem 5.27: Exemplo de uso do comando *insertMany*

O `MongoDB` gera automaticamente uma chave de identificação para cada novo item adicionado ao banco de dados a qual pode ser utilizada para respectivas manipulações. Por exemplo, Para deleção de um documento pode-se utilizar a função *remove()*. A Listagem 5.28 alustra a remoção de um documento utilizando a função remove() e o respectivo pelo seu identificador.

```
db.users.remove({_id:xxxx1111})
```

Listagem 5.28: Exemplo de uso do comando *remove()*

Um outro exemplo de uso do identificador é a edição de um documento usa-se por meio da função *update()*, como demonstrado na Listagem 5.29 no qual apenas o atributo idade é modificado.

```
db.users.update({_id:xxxx1111}, {$set: {idade: 23})
```

Listagem 5.29: Exemplo de uso do comando *update*

Para filtrar usa-se a diretiva *find()*. A Listagem 5.30 representa a seleção de todos os documentos da coleção *users*.

```
db.users.find()
```

Listagem 5.30: Exemplo de uso do comando *find*

Para filtragens seletivas dentro de uma coleção basta fornecer à função *find()* com uma condição. A Listagem 5.31 demonstra como selecionar os documentos da coleção *users* cujo atributo idade tenha o valor "22".

```
db.users.find({"idade": 22})
```

Listagem 5.31: Exemplo de uso do comando *find* com uma condição

Para deletar uma coleção usa-se a função *drop()*. A Listagem 5.32 representa a deleção da coleção *users*.

```
db.users.drop()
```

Listagem 5.32: Exemplo de uso do comando *drop*

No `MongoDB` não existe nenhuma cláusula que faça exatamente o quê o comando *join* faz, mas existe uma função que pode realizar uma operação similar, que é o comando *$lookup*. Supondo que existam as duas coleções, *users* e *products*. Como demonstrado nas Listagens 5.33 e 5.34.

```
{
    "_id": ObjectId("z8z8z8z8z8z8"),
    "username": "Jannielly",
    "age": 22
},
{
   "_id": ObjectId("a9a9a9a9a9a"),
    "username": "Ana Lara",
    "age": 20
}
```

Listagem 5.33: Coleção *users*

```
{
    "_id": ObjectId("sadhsa"),
    "user_id": ObjectId("z8z8z8z8z8z8"),
    "Produto_Name": "Computador"
},
{
    "_id": ObjectId("60a02e063f457f2ec02c1e12"),
    "user_id": ObjectId("a9a9a9a9a9a"),
    "Produto_Name": "Smartphone"
}
```

Listagem 5.34: Coleção *products*

A Listagem 5.35 demonstra como realizar a junção das coleções *users* e *products* por meio comando por meio do comando *$lookup* da função *aggregate()*. *users* e *products*.

```
 db.products.aggregate([
{
$lookup: {
  from: "users",
  localField: "user_id",
  foreignField: "_id",
  as: "user_info"
}
}
```

Listagem 5.35: Exemplo de uso do comando *aggregate*

O resultado dessa operação é a junção das duas coleções com base no identificador, cujo resultado é ilustrado na Listagem 5.36.

```
 {
    "_id": ObjectId("sadhsa"),
    "user_id": ObjectId("z8z8z8z8z8z8"),
    "Produto_Name": "Computador",
    user_info: [
    {
       "_id": ObjectId("z8z8z8z8z8z8"),
       "username": "Jannielly",
       "age": 22
    }
    ]
},
{
    "_id": ObjectId("sadhsa"),
    "user_id": ObjectId("z8z8z8z8z8z8"),
    "Produto_Name": "Computador",
    user_info: [
    {
       "_id": ObjectId("a9a9a9a9a9a"),
       "username": "Ana Lara",
       "age": 20
    }
    ]
}
```

Listagem 5.36: *aggregate*, *users* e *products*

Existem outras funções e variações das funções já apresentadas, cada uma atendendo a uma determinada especificidade. Para consultar outros comandos ou cláusulas é recomendado acessar a documentação do `MongoDB` no site oficial [5].

5.6 Uma visão geral sobre Redis

Criado em 2009 por Salvatore Sanfilippo, `Redis` é um SGBD de armazenamento em memória que trabalha com dados organizados por chave e valor (REDIS, 2023a). As chaves são *strings* que identificam o conteúdo guardado no valor, que, por sua vez, pode variar entre os seguintes tipos de dados: *strings*, *hashes,* listas, conjuntos, conjuntos ordenados, *bitmaps* e *hiperloglogs*. A Listagem 5.37 representa o armazenamento em *hash* da entidade "pessoa"em um SGBD `Redis`.

[5] *Documentação do `MongoDB`*

```
HMSET pessoa:1 nome "Francisca Jannielly Garcia da Costa"
idade 22 escolaridade "Ensino Superior Incompleto"
altura 159 peso 65 raca "Preta"
```

Listagem 5.37: Exemplo de uma estrutura *hash* no `Redis`

A principal motivação para o seu surgimento foi fornecer armazenamento de dados com alto desempenho e baixa latência, visando atender às necessidades de aplicações em tempo real, como jogos *online, chat, streaming de vídeo* , análise de dados em tempo real, entre outros.

Diferente do `MySQL` e do `MongoDB`, o `Redis` não é multiplataforma, isso significa dizer que ele não tem suporte para todas as variações de sistemas operacionais, rodando apenas em sistemas Unix, como MacOS e distribuições Linux. Apesar desta restrição, a partir do Windows 10, podemos rodar o `Redis` sem nenhum problema usando a ferramenta *Windows Subsystem for Linux* (WSL) (REDIS, 2023a).

O WSL é o subsistema Windows para Linux. Essa ferramenta permite realizar a execução de um ambiente Linux diretamente no seu Windows. O seu uso é de grande valia, já que essa ferramenta garante uma ponte entre os SOs Windows e Linux e assim, o desenvolvedor pode executar programas Linux sem a necessidade de realizar *dual-boot* ou instalar máquinas virtuais (MICROSOFT, 2023a).

5.6.1 Ferramenta de Gerenciamento

O `Redis` não possui ferramenta visual para realizar operações e consultas no banco de dados que seja *open source* e oficial, como `MySQL` *Workbench* ou `MongoDB` *Compass*. Portanto, para utilizar o `Redis`, serão apresentadas operações diretamente via linha de comando, tal qual será apresentado na subseção de linguagem de consulta e na subseção de instalação e configuração.

5.6.2 Arquitetura do Redis

Assim como os dois outros SGBDs apresentados, a arquitetura do `Redis` também é baseada na arquitetura cliente-servidor, isso significa que há uma separação entre o cliente (usuários), que solicita os dados, e o servidor (`Redis`), que é responsável por gerenciar, processar e armazenar esses dados.

5.6.3 Instalação e Configuração

Com intuito de demonstrar como o `Redis` pode ser utilizado, a seguir é apresentada uma sequência de passos para a instalação e configuração do mesmo. Como já dito anteriormente, o `Redis` é um SGBD que roda apenas em sistemas do tipo Unix, ou seja, pode ser instalado em qualquer SO Linux ou MacOS(REDIS, 2023a). Para esta seção, será realizada a instalação do WSL e `Redis` Server, além da configuração de um servidor numa máquina Windows 10.

Para executar o `Redis` Server em uma máquina Windows, é necessário primeiro realizar a habilitação do WSL. Para tal fim, basta abrir o *Windows powershell* e colar o comando da Listagem 5.38. Após a habilitação, é recomendado reiniciar a máquina.

```
Enable-WindowsOptionalFeature -Online
-FeatureName Microsoft-Windows-Subsystem-Linux
```

Listagem 5.38: Comando de instalação do WSL

Com o WSL habilitado, é possível executar um ambiente Linux diretamente da máquina Windows. No entanto, é necessário instalar a distribuição Linux que será utilizada. Recomenda-se baixar a versão mais recente do SO. Para este trabalho, será utilizado o Ubuntu 20.04.6 LTS. O qual pode ser facilmente instalado a partir da *Microsoft Store*, como mostrado na Figura 5.30.

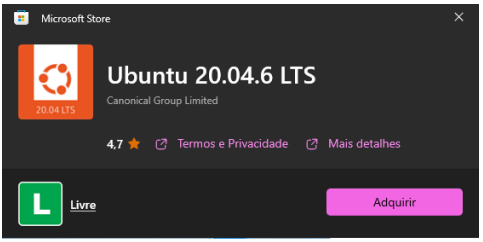


Figura 5.30: Microsoft Store - Ubuntu

Com o Ubuntu instalado, basta executá-lo e realizar as configurações necessária, dentre as quais, criar um usuário e uma senha para o sistema. A Figura 5.31 mostra o terminal do *Ubuntu* para configurações de segurança.

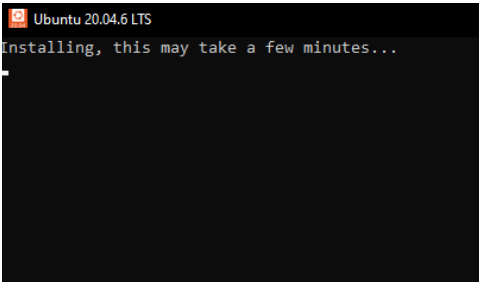


Figura 5.31: Terminal do Ubuntu

Após as configurações iniciais, é recomendado atualizar o sistema, o que é feito por meio dos comandos apresentados na Listagem 5.39. Após instalado e atualizado

o SO, o Redis Server pode finalmente ser instalado por meio do comando mostrado na Listagem 5.40.

```
sudo apt-get update
sudo apt-get upgrade
```

Listagem 5.39: Comando de atualização do gerenciador de pacotes Linux

```
sudo apt-get install Redis-server
```

Listagem 5.40: Comando de instalação do `Redis` Server

Com o `Redis` Server instalado, recomenda-se dar *restart* no servidor com o comando mostrado na Listagem5.41. A Listagem 5.42 mostra o comando que habilita o servidor para ser utilizado. Após habilitado, o `Redis` *Server* estará pronto para a realização de armazenamentos, consultas e manipulações de dados, como mostra a Figura 5.32.

```
sudo service Redis-server restart
```

Listagem 5.41: *Restart Server*

```
Redis-cli
```

Listagem 5.42: Comando para habilitar o `Redis`-Cli



Figura 5.32: `Redis`-Cli

5.6.4 Linguagem de Consulta

O `Redis` tem suporte a diversas linguagens de programação, como *C Sharp*, *JavaScript* e *Python*, mas a mais comumente utilizada é a sua própria linguagem de

consulta, que é baseada em linguagem shell para Linux. A seguir, são apresentados os principais comandos básicos, mas para se aprofundar na linguagem de consulta própria do Redis é importante ler a documentação disponível (REDIS, 2023a).

Como já dito, o `Redis` é um banco de dados baseado em chave e valor. Para setar um valor a em uma chave, usa-se o comando *set*. Caso o valor seja uma *string*, ele deve ser colocado entre aspas. A Listagem 5.56 mostra uma chave chamada "usuário1"sendo setada com um objeto JSON de três atributos.

```
set usuario1
'{"id":"x1x2x3","nome":"Jannielly","idade"=22}'
```

Listagem 5.43: Uso do comando *set*

Para inserção de chaves múltiplas, usa-se o comando *mset*. A Listagem 5.44 mostra três valores armazenados em uma única operação.

```
mset usuario1
'{"id":"x1x2x3","nome":"Jannielly","idade"=22}'
usuario2 '{"id":"x4x5x6","nome":"Klayver","idade"=22}'
usuario3 '{"id":"x7x8x9","nome":"Douglas","idade"=21}'
```

Listagem 5.44: Uso do comando *mset*

Para resgatar um valor, deve-se usar o comando *get* seguido da chave à qual está o valor desejado.A Listagem 5.45 mostra um exemplo de uso do comando *get*, no qual é resgatado o valor referente a chave ´´usuario1".

```
get usuario1
```

Listagem 5.45: Uso do comando *get*

Para o resgate de múltiplas chaves, usa-se o comando *mget*. A Listagem 5.46 mostra um exemplo de uso do comando *mget*, no qual três chaves são acessadas em uma única operação.

```
mget usuario1  usuario2  usuario3
```

Listagem 5.46: Uso do comando *mget*

Para o armazenamento de uma lista, existem duas opções. A primeira é *lpush*, que adiciona os valores da esquerda para direita. Já a segunda é o *rpush*, que segue o sentido inverso, ou seja, armazena os valores da direita para esquerda. As Listagens 5.47 e 5.48 representam a criação de uma lista de usuários, em uma estrutura JSON usando as duas opções supradescritas.

```
lpsuh lista_de_usuarios

'{"id":"x7x8x9","nome":"Douglas","idade"=21}'
'{"id":"x4x5x6","nome":"Klayver","idade"=22}'
'{"id":"x1x2x3","nome":"Jannielly","idade"=22}'
```

Listagem 5.47: Uso do comando *lpush*

```
rpush lista_de_usuarios
'{"id":"x7x8x9","nome":"Douglas","idade"=21}'
'{"id":"x4x5x6","nome":"Klayver","idade"=22}'
'{"id":"x1x2x3","nome":"Jannielly","idade"=22}'
```

Listagem 5.48: Uso do comando *rpush*

Para recuperar um valor de uma lista, usa-se o comando *lrange*, acompanhado de três parâmetros: o nome da lista, a posição inicial e a posição final. O `Redis` pode realizar a contagem reversa, ou seja, em vez de fornecer no terceiro parâmetro a posição final absoluta desejada, pode-se informar a posição final relativa desejada, utilizando o valor -1 para a última posição, o valor -2 para a penúltima posição e assim sucessivamente. A Listagem 5.49 representa o resgate dos valores de uma lista de usuários, na qual o valor 0 indica a posição do primeiro termo a ser mostrado, e o -2 indica que a lista será mostrada até o penúltimo termo.

```
lrange lista_de_usuarios 0 -2
```

Listagem 5.49: Uso do comando *lrange*

Pode-se também utilizar uma estrutura de *hash* para armazenar os valores. Para isso, usa-se o comando *hmset*, seguido de uma chave e dos atributos e respectivos valores. A Listagem 5.50 representa o armazenamento de um "usuário"no formato *hash*.

```
hmset usuario1 nome Jannielly idade 22 sexo Feminino
```

Listagem 5.50: Uso do comando *hmset*

Para o resgate de uma estrutura *hash*, usa-se o comando *hmgetall*, seguido do nome da estrutura. A Listagem 5.51 mostra o resgate de todos os atributos da estrutura *hash* chamada "usuario1".

```
hmgetall usuario1
```

Listagem 5.51: Uso do comando *hmgetall*

Para o resgate de um atributo específico em uma estrutura *hash*, usa-se o comando *hmget*, seguido do nome da estrutura e do atributo. A Listagem 5.52 representa a operação de resgate do atributo 'nome' da estrutura *hash* criada com o nome de chave "usuario1".

```
hmget usuario1 nome
```

Listagem 5.52: Uso do comando *hmget*

Para deletar uma chave do banco, usa-se o comando *del*, seguindo da chave a ser deletada. A Listagem 5.53 exemplifica o comando para fazer a deleção da chave "usuario1".

```
del usuario1
```

Listagem 5.53: Uso do comando *del*

Ao iniciar o `Redis`, um banco de dados já é criado por padrão. Para criar um novo banco ou selecionar um já existente, usa-se o comando *select*, seguido do número do banco desejado. A Listagem 5.54 exemplifica o comando. O `Redis` possui um limite padrão de 16 bancos de dados em um único servidor, numerados de 0 a 15 (REDIS, 2023a).

```
select usuario1
```

Listagem 5.54: Uso do comando *select*

Para verificar as chaves existentes no banco de dados, basta usar o comando *keys* * como demonstrado na Listagem 5.55.

```
keys *
```

Listagem 5.55: Uso do comando *keys* *

Com o `Redis` é possível criar uma chave temporária. O comando *expire* permite que seja definido um tempo para uma determinada chave expirar. Na Listagem 5.56 a chave "usuario"é criada. Na Listagem 5.57 é definido um tempo de 60s para a chave "usuario"expirar. Com o comando *ttl* (Time to Live), é possível observar o tempo restante para uma chave expirar, conforme mostrado na Listagem 5.58.

```
set usuario "Jannielly"
```

Listagem 5.56: Uso do *set*

```
expire usuario 60
```

Listagem 5.57: Uso do *expire*

```
ttl usuario
```

Listagem 5.58: Uso do *ttl*

Essa funcionalidade é de grande utilidade para otimização do banco de dados, já que com seu uso é possível realizar o gerenciamento de sessões, realizar controles de acesso temporário ou limpeza automática de dados obsoletos.

Em casos de acesso a chaves que ainda não foram setadas no banco ou deletadas, o valor de retorno será *nil*, que equivale a *null*.

5.7 Mapeamento objeto-relacional

Object Relational Mapping (ORM) é uma técnica que possibilita a comunicação entre a camada lógica e o banco de dados utilizado na aplicação. Essa técnica realiza o mapeamento e abstração de complexidades das operações de banco de dados, facilitando a manipulação, e persistência dos dados por meio de uma espécie de tradução entre a linguagem de programação e a estrutura da linguagem de consulta usada no banco de dados. Além de realizar o mapeamento entre objetos e tabelas, existem recursos que fazem do ORM uma técnica tão utilizada. Com ele, podemos gerar esquemas de estruturas automaticamente, gerenciar modelos de dados, acessar e gerenciar cache de objetos e ter controle sobre as transações efetuadas, todos esses recursos sem a necessidade de geração de códigos em linguagens de consulta, como SQL (CHEN, 2016).

Apesar das vantagens, também existem desvantagens em se utilizar um ORM em sua aplicação, tais como perda de desempenho e performance, bloqueios gerados por plataformas de gerenciamento de bancos de dados, além do aumento da complexidade da aplicação. Existe uma variedade de ORM, geralmente um para cada linguagem de programação, e cada qual com suas especificidades, como, por exemplo, o `Entity Framework` usado no `C Sharp/.NET`.

5.7.1 Entity Framework

`Entity Framework` é um ORM com suporte da `Microsoft` usado em aplicações web do tipo `ASP.NET`, sendo um componente que fornece ferramentas e recursos

para a realização do mapeamento entre aplicação e bancos de dados relacionais. Com o `Entity Framework` é possível realizar operações que vão desde um *Create, Read, Update e Delete* (CRUD) simples, até operações mais complexas, como *join* entre tabelas, rastreamento de mudanças, gerenciamento e otimização de consultas e consultas *Language Integrated Query* (LINQ).

O `Entity Framework` possui suporte ao chamado *"code first"*. Essa técnica consiste em realizar a criação, via código, dos modelos (entidades) e todas as suas configurações. O *code first* permite que os desenvolvedores foquem seu tempo na lógica do desenvolvimento enquanto o `Entity Framework`, após os modelos criados e lógica concluída, possa realizar as configurações de banco de dados. Assim como todos os ORMs, o `Entity Framework` possui comandos próprios para realizar a criação de instâncias, configurações e realizar consultas. Cada mudança que é feita no banco de dados, seja estrutural ou atualização de dados, é registrada pelas *migrations*. *Migration* ou migração, é uma abordagem que tem intuito de controlar as versões e moderar as mudanças realizadas no banco de dados de maneira a facilitar o trabalho em grupo e evitar possíveis perdas ou falhas.A Figura 5.33 apresenta o fluxo de comunicação em aplicações que usam ORM.

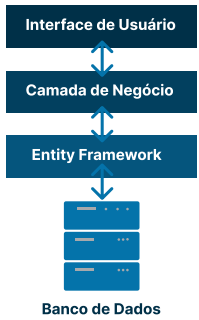


Figura 5.33: Comunicação usando `Entity Framework`

Fonte: Adaptado de (MATEUS, 2023)

Apesar de ser uma ferramenta que possui diversas funcionalidades, algumas limitações diminuem a popularidade do mesmo, tais como o fato de não ter suporte para *multithread* em um único contexto e sua perda de desempenho em aplicações que exigem uma complexidade mais alta nas consultas e aplicações (MICROSOFT, 2023b).

Capítulo 6

Back-end

Klayver Ximenes Carmo (UFC)

Antonio Josefran de Oliveira Bastos (UFC)

Fischer Ferreira (Unifei)

Tabela 6.1: Métodos HTTP

Método	Ação	Exemplo de uso
GET	Ler	Capturar recursos de um sistema
POST	Criar	Inserir um recursos em um sistema
PUT	Atualizar	Atualizar dados de um recurso em um sistema
DELETE	Apagar	Apagar recursos em um sistema

Este capítulo tem como principal objetivo fornecer uma visão geral de algumas tecnologias de *back-end* que podem ser utilizadas em aplicações para web: `Node.js`, `Django DRF` e `ASP.NET`. Assim, o capítulo está organizado da seguinte forma: A Seção 6.1 apresenta uma visão geral sobre API e API RESTful. A Seção 6.2 apresenta uma ferramenta chamada Insomnia para utilizar os exemplos de API apresentados no texto. Finalmente, as Seções 6.3, 6.4 e 6.5 discutem os fundamentos e apresentam exemplos práticos de `Node.js`, `Django DRF` e `ASP.NET`, respectivamente.

6.1 API e API RESTful

A API (*Application Programming Interface*), ou Interface de Programação de Aplicação, é uma interface de programação que permite a comunicação entre sistemas e aplicações. Isto é feito por meio de um conjunto de protocolos e ferramentas para o desenvolvimento de *software* e aplicativos. No contexto do desenvolvimento web, a API é exposta por um servidor que permite a comunicação e interação de clientes externos e em diferentes plataformas, como em *sites*, aplicativos *mobile* ou *softwares desktop*. Seu desenvolvimento permite criar uma camada de abstração entre o *front-end* e o *back-end*, facilitando a integração entre os serviços.

API *RESTFul* é um tipo específico de API que segue as restrições e princípios arquiteturais definidos pela arquitetura *REST*. A arquitetura *REST* baseia-se nos protocolos HTTP e *Hypertext Transfer Protocol Secure* (HTTPS), utilizados na comunicação entre cliente e servidor na maioria dos serviços de aplicação (VALES; BRADA, 2020). A nomenclatura *REST* refere-se à independência das solicitações anteriores de um cliente ao servidor, ou seja, um cliente pode fazer solicitações de recursos em qualquer ordem, sem estado e de forma isolada das outras requisições (AMAZON, 2023a).

A Tabela 6.1 apresenta os métodos HTTP mais utilizados para manipular os recursos disponíveis no servidor, tornando a arquitetura REST mais escalável e flexível devido à popularização e padronização dos verbos HTTP. Geralmente, as respostas de uma API são no formato JSON, *Extensible Markup Language* (XML) ou HTML e são transmitidas por meio de métodos HTTP. Uma API RESTFul pode ser construída em diferentes tecnologias *back-end*, como `Node.js` com o *framework* `Express`, `Django` REST e `ASP.NET` web API, no qual cada um desses *frameworks* possui suas próprias vantagens e desvantagens.

A Figura 6.1 mostra como funciona o processo do modelo arquitetural de uma API REST, onde um usuário é capaz de realizar a comunicação com o servidor por meio de uma API fazendo requisições utilizando os métodos HTTP. O servidor então retorna uma resposta HTTP ao cliente, podendo, por exemplo, ser no formato JSON, XML ou HTML. O formato da resposta dependerá da requisição do cliente ou das restrições do servidor.

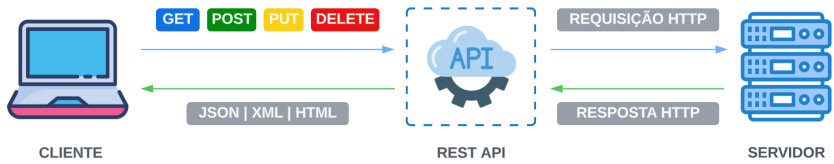


Figura 6.1: Modelo arquitetural de uma API REST

6.1.1 Persistência dos dados e configuração do banco de dados

No desenvolvimento de aplicações web, é de extrema importância o armazenamento das informações compartilhadas e trafegadas. Nesse contexto, os bancos de dados desempenham um papel fundamental no gerenciamento dela. Além da persistência, eles garantem a integridade, a segurança e a disponibilidade dos dados. Existem diferentes tipos de bancos de dados, cada um projetado para atender a necessidades específicas de armazenamento. Em geral, podem ser divididos em duas categorias principais: bancos de dados relacionais e não relacionais. Neste capítulo, será utilizado um banco de dados baseado no modelo relacional. Este modelo organiza dados em linhas e colunas, nas quais as tabelas e as relações entre elas são definidas previamente.

O banco de dados relacional a ser utilizado será o `MySQL`[1], que utiliza a linguagem SQL para consulta e interação com os dados. Os procedimentos de instalação do *Workbench* `MySQL` seguirão as etapas de instalação em uma máquina com sistema operacional *Windows*. O primeiro passo é acessar o site oficial de *download*[2] e clicar na opção de download, como destacado em vermelho na Figura 6.2. Os próximos passos para instalação devem seguir o recomendado pela documentação oficial[3].

[1] ⟨https://www.mysql.com/⟩
[2] ⟨https://dev.mysql.com/downloads/workbench/⟩
[3] ⟨https://dev.mysql.com/doc/workbench/en/wb-installing-windows.html#wb-windows-standalone⟩

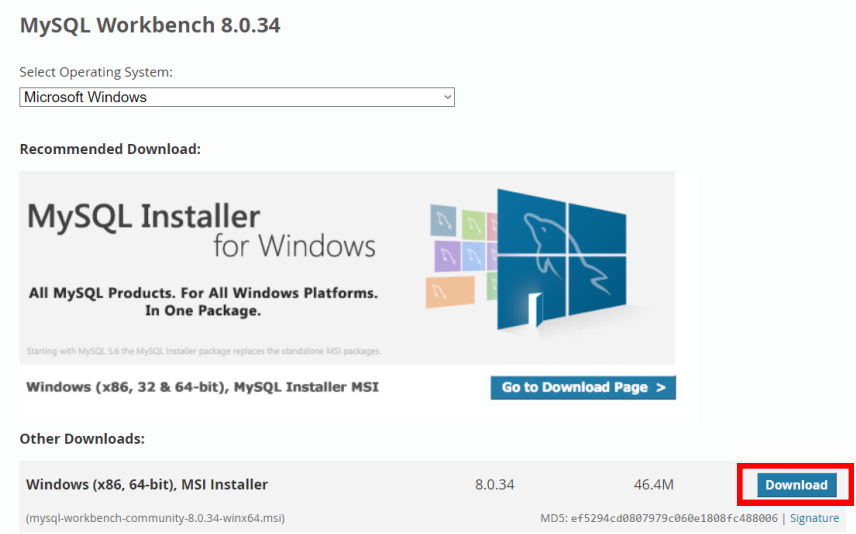


Figura 6.2: Página de *download* do *Workbench* `MySQL`

Com o *Workbench* já instalado, é necessário criar uma conexão para permitir a transferência de dados entre o aplicativo e o banco de dados. Na página inicial do programa instalado, é preciso clicar no botão de "+" na área de conexões `MySQL`, com isso, abrirá uma janela para configuração da conexão, conforme apresenta a Figura 6.3. Nela, é possível definir o nome da conexão e outras configurações, como o usuário, o *host* e a porta de conexão. Com a conexão já definida, é necessário criar o banco de dados/*schema*. A Listagem 6.1 apresenta a *query* SQL que deve ser executada no editor de texto da conexão.

```
1 CREATE SCHEMA "nome_do_banco_de_dados";
```

Listagem 6.1: *Query* SQL para criar um banco de dados

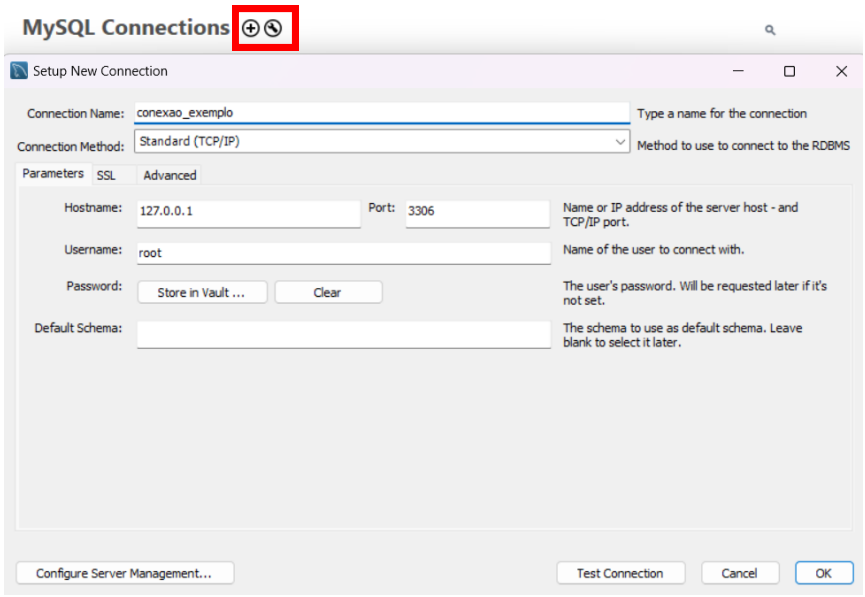


Figura 6.3: Página de configuração do *Workbench* `MySQL`

Os bancos a serem utilizados neste trabalho terão os seguintes padrões de nomenclatura:

- "crud_nodejs": Banco relacionado ao exemplo na tecnologia `Node.js`.
- "crud_drf": Banco relacionado ao exemplo na tecnologia *Django REST Framework* (DRF).
- "crud_aspnet": Banco relacionado ao exemplo na tecnologia `.NET`.

6.2 Insomnia

No desenvolvimento de APIs, são de suma importância o teste e a documentação do projeto. Existem algumas ferramentas que possibilitam a realização dessas atividades, como `Postman`[4] e `Insomnia`[5]. Neste capítulo, será utilizada a ferramenta `Insomnia` para testar as rotas por meio de solicitações HTTP. Ela pode ser obtida para instalação no site oficial. A ferramenta `Insomnia` contém várias funcionalidades que ajudam nas configurações para testes das requisições, como a possibilidade de criar ambientes de desenvolvimento, por meio da declaração de variáveis específicas para cada ambiente, como *Uniform Resource Locator* (URL)s que variam de acordo

[4] ⟨https://www.postman.com/⟩
[5] ⟨https://insomnia.rest/⟩

com o ambiente de trabalho (desenvolvimento, teste e produção). A mesma também permite a importação e exportação de uma coleção de requisições para uma equipe de desenvolvedores, sem necessidade de criar um novo ambiente de testes a cada uso.

Nesta seção será apresentada uma série de passos à fim de demonstrar como a ferramenta `Insomnia` pode ser instalada. O procedimentos seguirão as etapas para a instalação em uma máquina com sistema operacional *Windows*. O primeiro passo é acessar o site oficial de *download*[6], clicar em "*Download Insomnia for Windows*" e esperar que o programa seja iniciado. A Figura 6.4 apresenta a tela inicial da página de *download* da ferramenta para consumo de API.

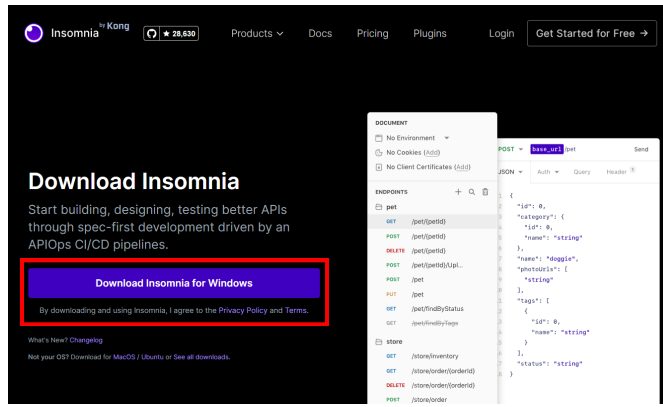


Figura 6.4: Página inicial de *download* da ferramenta `Insomnia`

Com a finalização do *download*, o próximo passo é clicar no instalador para iniciar a instalação. Uma tela para este procedimento será aberta com a logo da ferramenta `Insomnia`. Após a finalização do processo, a janela inicial do programa será aberta, e nela estarão disponíveis as opções de ferramentas que podem ser utilizadas. A Figura 6.5 apresenta a estrutura da página inicial da ferramenta.

[6] ⟨https://insomnia.rest/download⟩

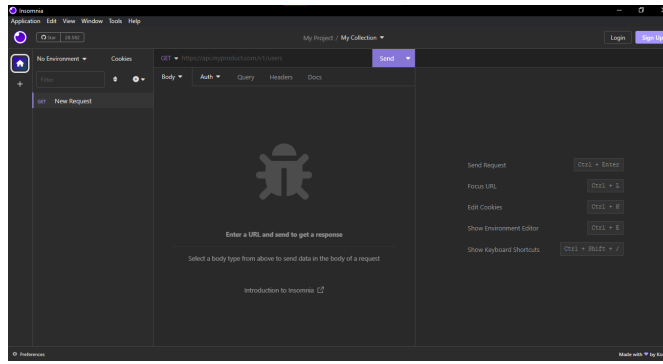


Figura 6.5: Painel inicial da ferramenta `Insomnia`

A Figura 6.6, adaptada de (INSOMNIA, 2023), mostra um exemplo de uso da ferramenta. Na parte superior é inserida a URL a ser testada, juntamente com o método HTTP desejado. Abaixo da URL é possível inserir algumas configurações conforme o teste a ser realizado, como anexar um corpo à requisição, autenticação, cabeçalhos, entre outros. No lado direito da figura, é apresentado um código de resposta 200, indicando o sucesso da requisição, bem como o tempo e o tamanho da requisição. Logo abaixo, é possível verificar o conteúdo da resposta retornada pela API. No exemplo, foi utilizada uma requisição à API do `GitHub`, buscando os dados de um usuário, na qual é possível ver informações como *login*, *id* e outras.

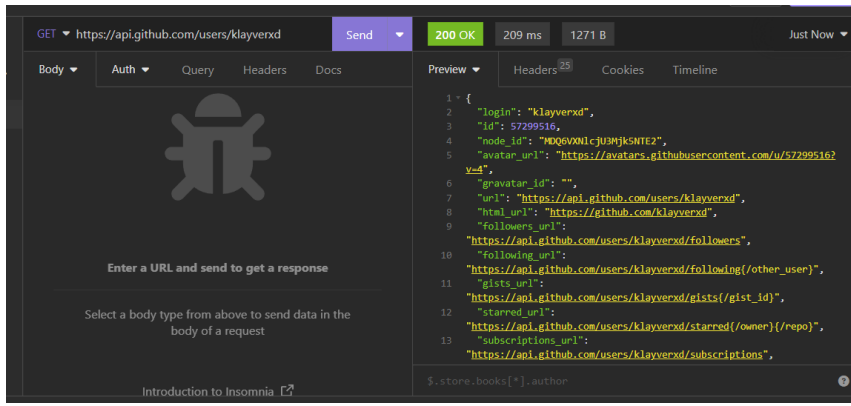


Figura 6.6: Interface da ferramenta `Insomnia`

6.3 Introdução ao Node.js

O `Node.js`[7] é uma plataforma para desenvolvimento de *software back-end* criada em 2009 por Ryan Dahl. Sua criação teve como intuito ser uma alternativa para as tecnologias da época, já que as existentes até então, possuíam deficiência em aplicações web em tempo real. Além disso, o `Node.js` é baseado em *JavaScript*, sendo construído sobre a *engine JavaScript* do Google para o Chrome, chamada V8. Por usar *JavaScript* como linguagem de desenvolvimento, o `Node.js` permite o desenvolvimento de aplicações na mesma linguagem, tanto no lado do servidor quanto no lado do cliente, além de oferecer uma grande quantidade de módulos e bibliotecas para um desenvolvimento mais rápido e facilitado.

Como um ambiente de execução *JavaScript* assíncrono orientado a eventos, o `Node.js` é projetado para desenvolvimento de aplicações escaláveis de rede. Sua versatilidade permite que seja utilizado em vários contextos de desenvolvimento, como em aplicações em tempo real, na construção de jogos, plataformas de *streaming* e ambientes escaláveis, como serviços de *back-end* e servidores.

Ao iniciar um projeto, temos vários pontos positivos para a escolha inicial do `Node.js`, como sua arquitetura não bloqueante, que permite o processamento simultâneo de várias solicitações de entrada e saída, tornando a plataforma altamente escalável e adequada para aplicações em tempo real. Por outro lado, também são encontradas algumas desvantagens, como o uso em aplicações que demandam alto uso de Unidade Central de Processamento (CPU), como aplicativos de renderização 3D ou de processamento de imagens. Esses pontos negativos podem surgir porque o `Node.js` é construído em torno de uma abordagem assíncrona e orientada a eventos. Portanto, para casos que exigem um grande processamento síncrono, seria mais adequada a utilização de outra plataforma de desenvolvimento.

Grandes empresas de tecnologia utilizam o `Node.js` em seus serviços, como *LinkedIn*, *Netflix* e *Paypal*, além de várias *startups* e desenvolvedores de *software* independentes. Além de sua flexibilidade e escalabilidade, o `Node.js` oferece facilidade de uso para manipulação de vídeos, arquivos, criação de *Web services* e outros. Sua ampla disponibilidade de módulos e pacotes de código aberto também a torna uma opção viável para a maioria dos casos de uso, sendo uma das melhores escolhas para a construção e a solução de aplicações eficientes. Essas grandes possibilidades reforçam o amplo uso da tecnologia e sua capacidade de atender às demandas do mercado.

6.3.1 JavaScript engine V8

O V8[8] é um mecanismo de alto desempenho desenvolvido pela *Google*, com código aberto e escrito em *C++*. Ele é usado no Chrome, no `Node.js`, entre outros projetos para a interpretação e execução de código *JavaScript*(V8, 2022). O motor `V8` serve para que o `Node.js` execute o código *JavaScript* por meio do *Just in Time*

[7] ⟨https://nodejs.org⟩
[8] ⟨https://v8.dev/⟩

(JIT), fazendo a compilação do *JavaScript* em *bytecode* e convertendo para uma linguagem de baixo nível em tempo de execução, para que seja entendida pelos processadores, em vez de interpretá-lo em tempo real (KRYLOV et al., 2020). A Figura 6.8 mostra como funciona essa sequência de processos, desde a análise e conversão à execução do código na CPU.

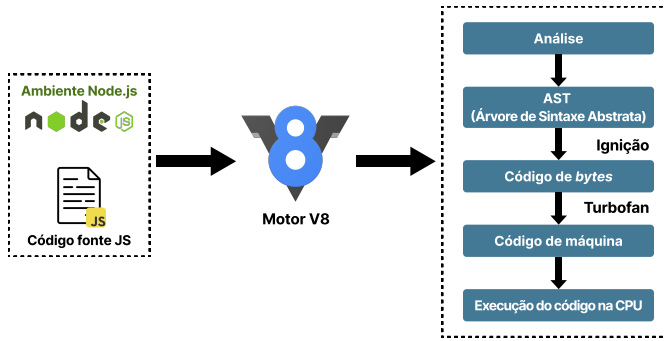


Figura 6.7: Funcionamento do motor V8

Além da compilação e execução do código *JavaScript*, o `V8` é responsável pela manipulação, alocação de memória e coleta de lixo dos objetos que não são utilizados. Esse gerenciamento de coleta de lixo é uma das principais funcionalidades que contribuem para o desempenho do `Node.js`. Ao liberar a memória ocupada por objetos desnecessários, o V8 evita vazamentos de memória e ajuda a manter a aplicação em execução de forma eficiente. Assim, a coleta de lixo é fundamental para garantir um melhor desempenho e maior eficiência em cenários de alta carga de trabalho e processamento intensivo (V8, 2022).

6.3.2 JavaScript Libuv (Eventloop)

A biblioteca `Libuv`[9] é responsável pela assincronicidade do modelo de entradas e saídas não bloqueantes. Seu desenvolvimento teve como objetivo o uso em aplicações `Node.js`, mas atualmente também é utilizada por `Luvit`[10] (sistema de execução assíncrono para a linguagem de programação Lua), `Julia`[11] (linguagem de programação que utiliza a biblioteca em seu sistema de execução, chamado *Julia's Event IO*), `uvloop`[12] (biblioteca assíncrona para Python) e outros. A `Libuv` é uma biblioteca de código aberto que fornece uma camada de abstração sobre o sistema operacional, permitindo que os programas executem operações de entrada e saída simultaneamente com o gerenciamento de eventos, processos e *threads*. Seu

[9] ⟨https://docs.libuv.org/en/v1.x/⟩
[10] ⟨https://luvit.io/⟩
[11] ⟨https://julialang.org/⟩
[12] ⟨https://github.com/MagicStack/uvloop⟩

funcionamento baseia-se no conceito de *loop* de eventos, que é iniciado quando o `Node.js` é iniciado (LIANG et al., 2017).

A Figura 6.8, adaptada de (LIBUV, 2014), apresenta a ordem das operações centrada no *loop* de eventos. O ciclo inicia quando o `Node.js` recebe uma requisição, o que dispara um evento por meio de sua arquitetura orientada a eventos. O conjunto de eventos gera uma fila de requisições, que são solicitações de diversos tipos feitas ao ciclo de eventos do `Node.js`. Uma condicional é inicialmente executada para verificar se a requisição é do tipo bloqueante; caso não seja, ela é rapidamente processada e retornada ao usuário. Quando uma tarefa bloqueante é requisitada no ciclo de eventos, ela é enviada internamente ao grupo de *threads* por meio do `Libuv`. Este conjunto de *threads* compartilhadas é necessário para que não seja preciso fazer o bloqueio do ciclo principal, tornando-se eficiente no consumo de recursos. Quando o processamento interno é concluído, é retornada uma *callback* ao ciclo de eventos, que, por sua vez, envia a resposta ao usuário (ROBERTS, 2014).

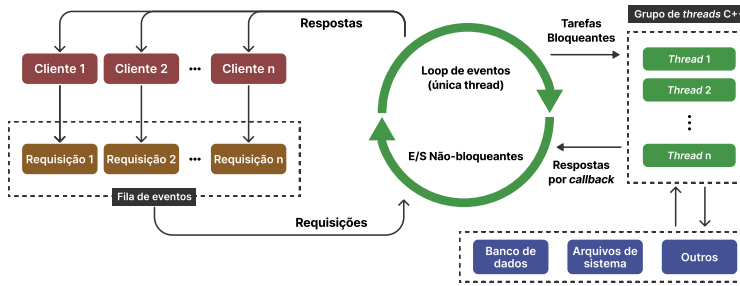


Figura 6.8: Diagrama de operações do *loop* de eventos

6.3.3 Diferença entre JavaScript para front-end e back-end

Embora a linguagem *JavaScript* possa ser utilizada tanto no *front-end* quanto no *back-end*, há diferenças importantes na implementação e nas funcionalidades ao desenvolver para a web e na construção de APIs. A Figura 6.9 mostra algumas tecnologias que são utilizadas para o desenvolvimento web, como *React*, *Vue* e *Angular*. No *back-end*, temos o `Node.js`, tecnologia em estudo utilizada na construção de servidores e APIs.

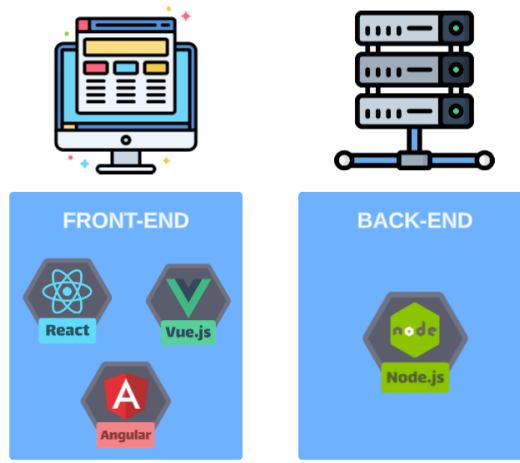


Figura 6.9: Diagrama de operações do *loop* de eventos

No desenvolvimento de aplicações para navegadores, o *JavaScript* é interpretado e executado em uma máquina virtual. Ele é responsável pela interatividade, pela manipulação da estrutura de uma página web, chamada de *Document Object Mode*l (DOM), pelo envio de requisições e por outras tarefas relacionadas ao *front-end*. Já no *back-end*, como no desenvolvimento em `Node.js`, o *JavaScript* é executado fora do navegador, permitindo o acesso a recursos do sistema, como arquivos e outros recursos típicos de um servidor. Dessa forma, o *JavaScript* é utilizado para construir aplicações *back-end* completas, como APIs e serviços web. No lado do cliente, onde o *JavaScript* é comumente utilizado, o navegador fornece a DOM para a manipulação dos objetos, o mesmo acontece para o `V8`, onde é fornecido pela *engine* todos os tipos de dados, operadores, objetos e funções especificados no padrão `ECMAScript` e `CommonJS`.

6.3.4 Ambiente de desenvolvimento para o Node.js

O `Node.js`[13] é uma plataforma de desenvolvimento *cross-plataform*, o que significa que ele pode ser executado em diferentes sistemas operacionais, como Windows, Linux e MacOS. Isso ocorre porque o `Node.js` é baseado em *JavaScript*, uma linguagem de programação interpretada pelo navegador ou pelo ambiente de execução do `Node.js`, e não depende de nenhuma plataforma específica.

Para ter um ambiente de desenvolvimento `Node.js`, é necessário instalá-lo no seu sistema operacional a partir do site oficial. Após a instalação, é possível verificar se tudo ocorreu de forma correta digitando "`node --version`" no terminal

[13] ⟨https://nodejs.org⟩

ou *prompt* de comando do sistema operacional. O comando exibirá a versão do `Node.js` instalada, confirmando se a instalação foi bem-sucedida.

Além disso, é possível utilizar o gerenciador de pacotes do `Node.js`, o *Node Package Manager* (NPM), que já vem instalado com o *software*. Para verificar se o NPM está instalado corretamente, basta digitar o comando "`npm --version`" no terminal ou *prompt* de comando. O comando exibirá a versão do NPM instalada, confirmando se a instalação ocorreu de forma bem-sucedida. O NPM é uma ferramenta muito útil para instalar e gerenciar pacotes e bibliotecas necessários ao desenvolvimento de aplicações `Node.js`.

6.3.5 Framework Express

No desenvolvimento de APIs RESTful, o `Node.js` possui o *framework Express*[14], que facilita a criação de aplicações e é um dos mais populares devido à sua facilidade, flexibilidade e eficiência. Ele fornece uma camada de abstração sobre o protocolo HTTP, reduzindo a complexidade e fornecendo uma maneira mais fácil de lidar com rotas, *middlewares* para a manipulação no processo de requisições e respostas, além de permitir a integração com outros módulos, como diversos bancos de dados e outros serviços.

A Listagem 6.2 apresenta um servidor simples em `Node.js` que utiliza o *framework Express*. Nele, é criada uma instância do módulo *Express* e definida uma rota com o método GET na raiz da aplicação. Quando o servidor é iniciado com o método listen() na porta 3000, ele fica disponível para receber solicitações HTTP e uma mensagem é exibida no console indicando que o servidor está rodando na porta especificada. Ao receber uma solicitação de um cliente com o método GET para a rota raiz, a função de *callback* associada é executada, enviando como resposta uma mensagem "Ola, mundo!" para o usuário.

```
1  const express = require('express')
2  const app = express()
3
4  app.get('/', (req, res) => {
5    res.send('Ola, mundo!')
6  })
7
8  app.listen(3000, () => {
9    console.log('Servidor rodando na porta 3000!')
10 })
```

Listagem 6.2: Exemplo de um servidor Express

[14] ⟨https://expressjs.com/⟩

6.3.6 Exemplo CRUD Node.js com `Express`

Uma das aplicações mais básicas no desenvolvimento de um projeto com banco de dados é a implementação das operações do CRUD (*Create*, *Read*, *Update* e *Delete*). Nesta seção, será apresentado um projeto que abrange todas as operações básicas de uma API, permitindo a criação, leitura, atualização e exclusão de dados. Para implementar o CRUD em `Node.js`, é necessário criar rotas que aceitem solicitações HTTP, como GET, POST, PUT e DELETE, e realizar as operações correspondentes no banco de dados. O `Node.js` possui muitas bibliotecas e ferramentas que podem ser utilizadas para implementar operações CRUD, como o `Express` que foi mencionado na Seção 6.3.5.

Para iniciar um projeto em `Node.js` é necessário executar o comando "`npm init -y`" , com isso será criado um arquivo "package.json" na raiz do projeto com as configurações padrões, local onde foi executado o comando. Este arquivo contém as informações do projeto, tais como nome, versão, autor, dependências, *scripts*, entre outras, ele é importante para gerenciar as dependências do projeto e também para compartilhar o mesmo com outros desenvolvedores, facilitando sua instalação e execução.

Para a estruturação do projeto, serão instaladas as bibliotecas que auxiliarão no desenvolvimento. A execução do comando "`npm install express mysql2 nodemon dotenv`" faz com que o gerenciador de pacotes instale as seguintes bibliotecas:

- `Express` (obrigatória): *Framework* que oferece recursos para a criação de aplicações web.
- `mysql2` (obrigatória): *Driver* do MySQL para `Node.js`, permitindo conectar-se a um banco de dados MySQL e executar consultas SQL.
- `nodemon` (opcional): Ferramenta que monitora as alterações no código-fonte e reinicia automaticamente o servidor sempre que uma alteração é detectada em tempo de desenvolvimento.
- `dotenv` (opcional): Um pacote que permite carregar variáveis de ambiente a partir de um arquivo .env.

A Listagem 6.3 apresenta um arquivo na raiz do projeto, nomeado de "index.js". Inicialmente, são importados os pacotes "express"e "dotenv"(Linhas 1 e 2). Na constante "app"(Linha 7), é instanciado o *framework Express*, assim como a configuração para o *parsing* de JSON (Linha 11) que será recebido nas requisições. As rotas do servidor são definidas no arquivo "userRouters.js"(apresentado na Listagem 6.5), onde é importada como "userRoutes" e definida para que todas as requisições feitas na raiz do servidor ("/") sejam tradadas neste módulo separadamente. Por fim, é feita a inicialização do servidor na porta definida nas variáveis de ambiente e é exibida uma mensagem no console para indicar que o servidor está em execução (Linhas 15 e 16).

```
import express from "express";
import dotenv from "dotenv";
dotenv.config();

import userRoutes from "./routes/userRouters.js";

const app = express();

const port = process.env.PORT || 5000;

app.use(express.json());

app.use("/", userRoutes);

app.listen(port, () => {
        console.log(`Server is listening on port ${port}`);
});
```

Listagem 6.3: Arquivo principal de um servidor CRUD

Para a comunicação com o banco de dados, será criada, na raiz do projeto, uma pasta de configurações do servidor, chamada "config". Nesta pasta será criado um arquivo chamado "database.js"(Listagem 6.4). Como demostrado na Listagem 6.4, será feita a instância (por meio do "mysql2.createConnection") (Linha 1) e a conexão com o banco de dados usando as variáveis de ambiente, informando o endereço, nome do usuário, senha e nome do banco de dados (Linhas 5 a 10). As credenciais de acesso ao banco de dados para a conexão seguem os passos apresentados na Seção 6.1.1. Além disso, por meio das Linhas 5 a 10, a conexão será exportada para que outros arquivos possam utilizá-la para realizar as operações CRUD no banco de dados.

```
import mysql2 from "mysql2";
import dotenv from "dotenv";
dotenv.config();

export const database = mysql2.createConnection({
        host: process.env.DB_HOST,
        user: process.env.DB_USER,
        password: process.env.DB_PASSWORD,
        database: process.env.DB_DATABASE,
});
```

Listagem 6.4: Arquivo de conexão com o banco de dados do CRUD

A defunção das rotas é feita no arquivo "userRouters.js" dentro da pasta *routes*, na raiz do projeto, seguindo a Listagem 6.5, onde o roteador é instanciado usando

o método “express.Router()”(Linha 8). Em seguida, definem-se os *endpoints* para os métodos GET, POST, PUT e DELETE. Quando um determinado *endpoint* é requisitado, é executada uma função de controle correspondente, disponível, nos arquivos da pasta *controllers* para cada verbo HTTP.

```
1  import express from "express";
2
3  import {lerUsuarios} from "../controllers/lerUsuarios.js";
4  import {adicionaUsuario} from "../controllers/adicionaUsuario.js";
5  import {atualizaUsuario} from "../controllers/atualizaUsuario.js";
6  import {deletaUsuario} from "../controllers/deletaUsuario.js";
7
8  const router = express.Router();
9
10 router.get("/", lerUsuarios);
11 router.post("/", adicionaUsuario);
12 router.put("/:id", atualizaUsuario);
13 router.delete("/:id", deletaUsuario);
14
15 export default router;
```

Listagem 6.5: Arquivo de rotas do CRUD

As funções de controle são feitas em arquivos separados correspondentes a cada método HTTP seguindo o seguinte padrão:

- Arquivo ”adicionaUsuario.js”: corresponde ao método POST.
- Arquivo ”atualizaUsuario.js”: corresponde ao método PUT.
- Arquivo ”deletaUsuario.js”: corresponde ao método DELETE.
- Arquivo ”lerUsuario.js”: corresponde ao método GET.

A estrutura final do projeto deverá ficar como mostrado na Figura 6.10.

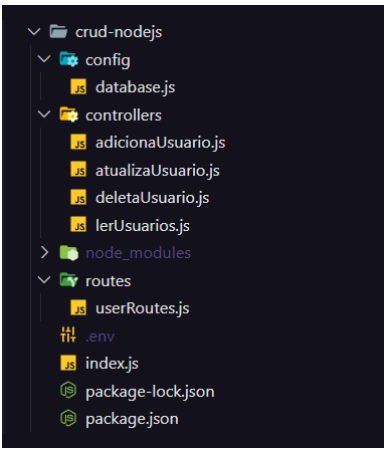


Figura 6.10: Funcionamento do motor V8

Os controladores de rota para a implementação do método GET são apresentados na Listagem 6.6. A função ("lerUsuarios") é responsável por processar solicitações de leitura de dados feitas por meio do verbo HTTP, realizar operação de seleção dos registros no banco de dados com a consulta *getQuery* no banco de dados *database* e enviar as respostas apropriadas aos clientes. Um exemplo de utilização do método GET pode ser observado na Figura 6.11. Nesse exemplo, a requisição ao servidor retornará todos os usuários cadastrados no banco de dados em formato JSON, com as informações de id, nome, *email* e idade.

```
1  import { database } from "../config/database.js";
2
3  export const lerUsuarios = (_, res) => {
4          const getQuery = "SELECT * FROM usuarios";
5
6          database.query(getQuery, (err, data) => {
7                  if (err) return res.json(err);
8
9                  return res.status(200).json(data);
10 });
11 };
```

Listagem 6.6: Arquivo controlador do método GET

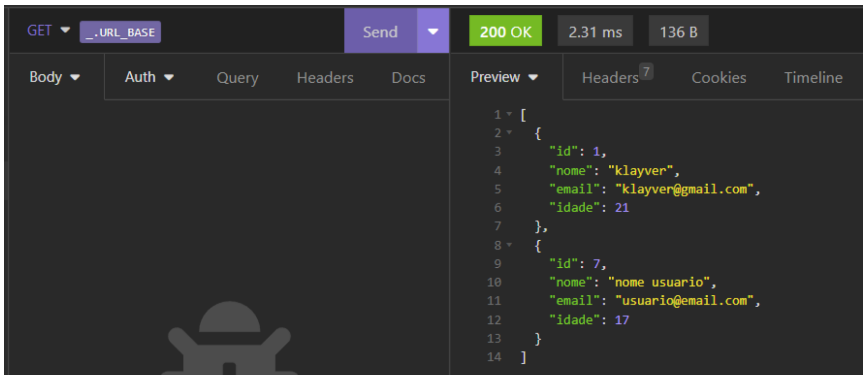


Figura 6.11: Utilização do método GET (API Node.js)

Na Listagem 6.7 é apresentada a função ("adicionaUsuario") necessária para processar solicitações de inserção de dados por meio do verbo HTTP PUT. Essa função insere o registro no banco de dados com a consulta *insertQuery* e envia as respostas apropriadas aos clientes. Para tal operação, também é possível validar os dados enviados pelo cliente para obter maior consistência. Um exemplo de como consumir o método POST pode ser visto na Figura 6.12. Nesse exemplo, a requisição ao servidor enviará um JSON com as informações de um novo usuário no corpo da requisição, incluindo nome, *email* e idade. O servidor então inserirá o novo registro de usuário no banco de dados e retornará uma mensagem de sucesso ao cliente.

```
import { database } from "../config/database.js";

export const adicionaUsuario = (req, res) => {
const insertQuery =
    "INSERT INTO usuarios(`nome`, `email`, `idade`) VALUES(?)";

const valores = [req.body.nome, req.body.email, req.body.idade];

database.query(insertQuery, [valores], err => {
    if (err) return res.json(err);

    return res.status(200).json("Usuário adicionado com sucesso!");
});
};
```

Listagem 6.7: Arquivo controlador do método POST

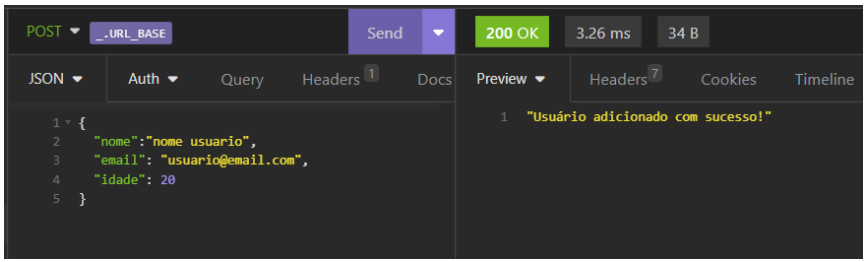


Figura 6.12: Utilização do método POST (API Node.js)

Na Listagem 6.8, é apresentada a função ("atualizaUsuario") necessária para processar solicitações de atualização de dados por meio do verbo HTTP PUT. Essa função é responsável por realizar a atualização do registro no banco de dados por meio da consulta *updateQuery* e enviar uma resposta de sucesso na alteração para o clientes. É possível realizar a validação dos dados enviados pelo cliente para garantir a consistência dos mesmos. Um exemplo de como consumir o método PUT pode ser observado na Figura 6.13. Nesse exemplo, a requisição feita ao servidor enviará um JSON com as informações de um usuário existente a ser atualizado, incluindo nome, *email* e idade, no corpo da requisição em formato JSON. O servidor então atualizará o registro correspondente no banco de dados de acordo com o id repassado como parâmetro na URL e retornará uma mensagem de sucesso ao cliente caso a operação tenha sido realizada corretamente.

```
import { database } from "../config/database.js";

export const atualizaUsuario = (req, res) => {
const updateQuery =
    "UPDATE usuarios SET `nome` = ?, `email` = ?, `idade` = ?
    WHERE `id` = ?";

const valores = [req.body.nome, req.body.email, req.body.idade];

database.query(updateQuery, [...valores, req.params.id], err => {
    if (err) return res.json(err);

    return res.status(200).json("Usuário atualizado com sucesso!");
});
};
```

Listagem 6.8: Arquivo controlador do método PUT

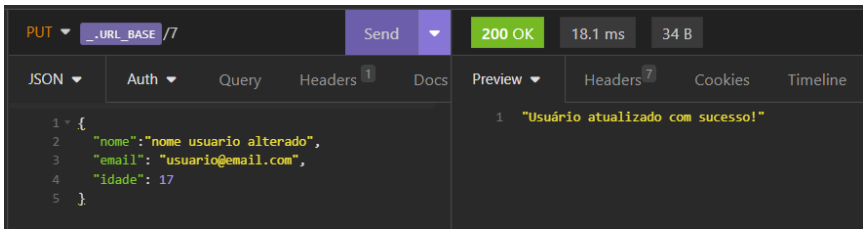


Figura 6.13: Utilização do método PUT (API Node.js)

Na Listagem 6.9, é apresentada a função ("deletaUsuario") responsável por processar solicitações de exclusão de dados por meio do verbo HTTP DELETE. Essa função é responsável por realizar a exclusão do registro correspondente no banco de dados por meio da consulta *deleteQuery* e enviar uma resposta de sucesso na remoção para o cliente. É possível realizar a validação dos dados enviados pelo cliente para garantir a consistência dos dados, como a verificação para saber se é um registro existente. Um exemplo de como consumir o método DELETE pode ser observado na Figura 6.14. Nesse exemplo, a requisição feita ao servidor enviará o "id" do usuário a ser excluído como parâmetro na URL. O servidor então excluirá o registro correspondente do banco de dados e retornará uma mensagem de sucesso ao cliente caso a operação tenha sido realizada corretamente.

```
import { database } from "../config/database.js";

export const deletaUsuario = (req, res) => {
const deleteQuery = "DELETE FROM usuarios WHERE `id` = ?";

database.query(deleteQuery, [req.params.id], err => {
    if (err) return res.json(err);

    return res.status(200).json("Usuário removido com sucesso!");
});
};
```

Listagem 6.9: Arquivo controlador do método DELETE

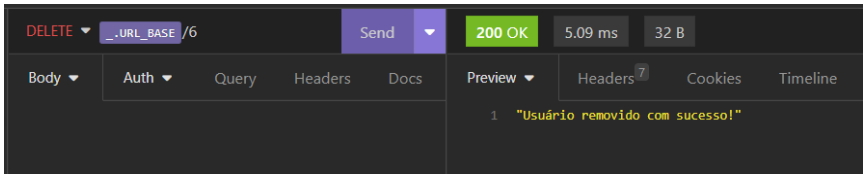


Figura 6.14: Utilização do método DELETE (API Node.js)

6.4 Introdução ao Django REST Framework

O `Django`[15] é um *framework* web *Python* criado em 2003 por Adrian Holovaty e Simon Willison que incentiva o desenvolvimento rápido e um *design* limpo e pragmático, com foco em escalabilidade rápida e flexível, velocidade de desenvolvimento e segurança. Inicialmente, foi criado para gerenciar um site jornalístico, mas em 2005 se tornou *open source* com licença *Berkeley Software Distribution* (BSD) (DJANGO, 2022). Muitos desenvolvedores escolhem este *framework* devido à sua organização e simplicidade de aprendizado e instalação. O `Django` também possui outras características atrativas para quem escolhe utilizá-lo no desenvolvimento de uma aplicação. Por exemplo, a velocidade no desenvolvimento, uma vez que oferece estruturas prontas para uso em várias tarefas comuns de desenvolvimento web, como autenticação de usuários e administração de conteúdo, além de ser um projeto gratuito de código aberto (AMAZON, 2023b).

Esta tecnologia possui vários conceitos no seu desenvolvimento, o que se destaca entre eles é o padrão MVT, que, atrelado à linguagem *Python*, permite a criação de aplicações com poucas linhas de código. Devido à sua ampla gama de recursos de desenvolvimento, várias empresas optaram por utilizá-lo em suas aplicações, incluindo a *Mozilla*, o *Instagram* e a *NASA*. Sua utilização abrange diversos cenários, como gerenciamento de servidores web e de bancos de dados, roteamento de URLs, processamento de dados e gerenciamento de eventos. Além disso, a comunidade de desenvolvedores em torno desta tecnologia é bastante ativa, o que contribui para a disponibilidade de recursos e de suporte.

Para a construção de APIs RESTful, foi criado em 2013 por Tom Christie, o `Django` REST Framework[16], que fornece algumas funcionalidades e ferramentas adicionais para o desenvolvimento de forma rápida, fácil e eficiente. Estão incluídos em algumas de suas funcionalidades os pacotes de política de autenticação (OAuth1a e OAuth2) e o suporte à serialização de dados, ORM e não ORM, que serão explicados na Seção 6.4.3. O DRF funciona como um complemento ao `Django`, sendo necessária a sua instalação para utilizar o REST framework (ALURA, 2023). Muitas empresas internacionais utilizam o DRF em suas aplicações, como *Mozilla*, *Red Hat*, *Heroku*, e *Eventbrite* (DRF, 2023).

6.4.1 Padrão MVT

O *Model-View-Template* é um padrão de *design* para desenvolvimento de aplicações web que separa as responsabilidades de cada componente da aplicação, facilitando o entendimento e a manutenibilidade. A Figura 6.15 apresenta todas as camadas do padrão MVT e como elas se relacionam, tanto entre si quanto com as requisições de um usuário.

[15] ⟨https://www.djangoproject.com/⟩
[16] ⟨https://www.django-rest-framework.org/⟩

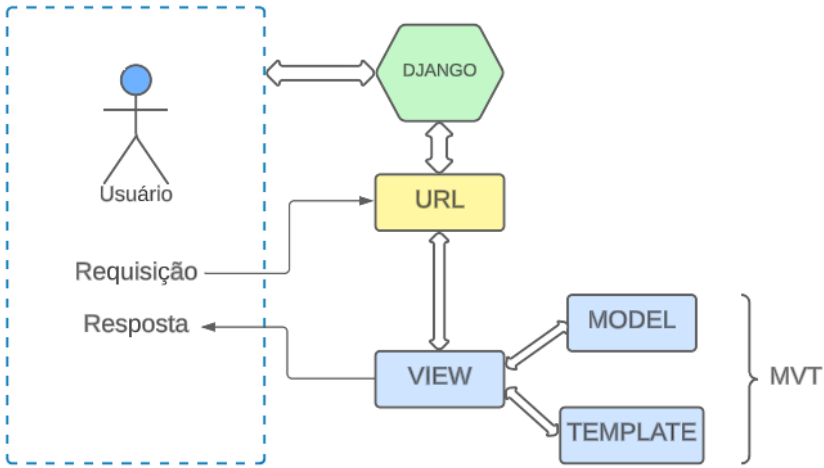


Figura 6.15: Representação modelo MVT

O Modelo atua como a interface dos seus dados, sendo responsável pela manutenção e pela estrutura lógica por trás de toda a aplicação, e é representado por um banco de dados (geralmente um banco de dados relacional como `MySQL` ou `PostgreSQL`), definindo como os dados serão armazenados, acessados e manipulados. A visualização é a interface do usuário vista no navegador quando é renderizado um site. Ela é representada por arquivos HTML/CSS/JS, sendo responsável pela lógica de negócios da aplicação e pela interação com o usuário. A Visualização recebe as solicitações do usuário, consulta o modelo e renderiza uma resposta para o usuário. O *Template* consiste em partes estáticas da saída HTML desejada, bem como em alguma sintaxe especial que descreve como o conteúdo dinâmico será inserido. Nele serão definidos a estrutura e o *layout* da página que é responsável por exibir os dados fornecidos pela visualização (GEEKSFORGEEKS, 2023).

6.4.2 Princípio *Don't Repeat Yourself* (DRY)

Citado no livro "Código Limpo" de Uncle Bob, o princípio DRY é direcionado à não duplicação de código, prezando pela reutilização e facilitando, principalmente, a manutenção e atualização (MARTIN, 2008). Durante o processo de desenvolvimento de uma aplicação, é comum encontrar dificuldades na escrita de um código abstrato. Quando se chega ao ponto de repetir determinados trechos de código, é recomendado verificar se é possível substituir o código repetido por um código mais abstrato, escrito e utilizado. Nesse sentido, a generalização envolve identificar os aspectos repetidos e, a partir disso, passar a escrever um código mais abstrato (HAOYU; HAILI, 2012).

A aplicação correta do princípio implica diretamente na modularização e reutilização, ou seja, uma modificação em um módulo não implica uma mudança em outro elemento logicamente não relacionado. O objetivo do reuso de pedaços de código é introduzir relações entre classes no paradigma *Object-oriented Programming* (POO), bem como melhorar a coesão e o acoplamento. Na POO, o DRY é implementado por meio da criação de classes e objetos que encapsulam comportamentos e propriedades comuns, evitando a repetição desnecessária de código (CABEZAS et al., 2020).

6.4.3 Serialização e deserialização

A serialização é o processo de conversão de dados para facilitar o consumo por parte de aplicações. No contexto do Django REST *Framework*, a serialização refere-se à conversão de objetos do *Python* em formatos de dados comuns, como XML, *YAML Ain't Markup Language* (YAML) ou o mais utilizado, JSON. O `Django` fornece um módulo de serialização padrão que pode ser usado para serializar dados em vários formatos, mas o DRF oferece recursos avançados de serialização que simplificam e agilizam a conversão de objetos *Python* em dados com formato compatível com a web.

Várias classes de serialização são oferecidas pelo DRF para personalizar a serialização de dados, como para modelos, listas e *hyperlinked*, de acordo com as necessidades da aplicação. Além de incluir um conjunto completo de campos de serialização personalizados para tipos de dados comuns, como datas e horas. O DRF também inclui a funcionalidade de validação de dados. O Serializador pode ser configurado para validar os dados de entrada e fornecer mensagens de erro detalhadas caso ocorra algum problema. Isso ajuda a garantir que a entrada do usuário seja sempre válida e consistente. A Listagem 6.10 apresenta um exemplo de serialização nos campos definidos, tomando como exemplo os dados de um comentário, como o *email*, conteúdo e data de criação.

```
1  from rest_framework import serializers
2
3  class CommentSerializer(serializers.Serializer):
4      email = serializers.EmailField()
5      content = serializers.CharField(max_length=200)
6      created = serializers.DateTimeField()
7
8  serializer = CommentSerializer(comment)
9  serializer.data
10
11 >> {'email': 'leila@example.com', 'content': 'foo bar', 'created': '
       2022-05-27T15:17:10.375877'}
```

Listagem 6.10: Exemplo serialização de dados

6.4.4 Ambiente de desenvolvimento para o Django REST Framework

No desenvolvimento em Django REST *Framework*, a escolha do sistema operacional é uma questão de preferência pessoal do desenvolvedor, pois o DRF pode ser executado em vários sistemas operacionais, desde que sejam compatíveis com a versão do *Python* utilizada para o desenvolvimento. Para criar um ambiente de desenvolvimento adequado, é necessário instalar o *Python*, que pode ser baixado a partir do site oficial[17]. O site disponibiliza tanto a versão mais atualizada quanto versões anteriores específicas. Após a instalação, é possível verificar a versão do *Python* instalada por meio do comando "`python --version`" no console do terminal de comando.

Após a instalação do *Python*, é recomendável criar um ambiente virtual para desenvolvimento, o que ajuda a evitar conflitos entre as dependências de projetos e as bibliotecas instaladas no sistema, além de permitir ter várias versões do *Python* instaladas para projetos específicos. A biblioteca "venv"já vem instalada com o *Python*, e para criar um ambiente virtual, basta executar o comando "`python -m venv myenv`" no terminal. Para ativar o ambiente virtual no sistema operacional Windows, por exemplo, é necessário executar o comando "`myenv\Scripts\activate`" no terminal.

A instalação do `Django` pode ser feita por meio do gerenciador de pacotes nativo do *Python*, o "pip", que já é incluído na instalação padrão do *Python*. Para instalar o *framework*, basta executar o comando 'pip install django' no terminal. Para criar um novo projeto no `Django`, basta executar o comando "`django-admin startproject nome_do_projeto`" no terminal.

6.4.5 Exemplo CRUD Django REST Framework

Contextualmente, a criação de uma API RESTful com o DRF envolve, primeiramente, a definição dos modelos de dados, que representam os objetos de uma aplicação a serem expostos pela API. Após a criação dos modelos, é preciso definir as operações que podem ser realizadas sobre esses objetos, com base na arquitetura REST, incluindo os métodos de leitura, criação, atualização e exclusão. Esta definição é feita por meio das visualizações do DRF, que especificam como os dados serão manipulados e retornados em resposta às solicitações HTTP. Para tornar os dados acessíveis pela API, é necessário converter os objetos do modelo em um formato serializado, como JSON ou outro formato compatível com a web. O DRF fornece uma camada de serialização, que permite definir como os objetos do modelo serão serializados e desserializados. Este processo é feito usando os serializadores do DRF, que transformam os objetos em dados serializados e vice-versa.

Para iniciar um projeto CRUD no DRF é necessário instalar, por meio do gerenciador de pacotes pip no terminal, os pacotes que serão utilizados com o co-

[17] ⟨https://www.python.org/downloads/⟩

mando "`pip install django djangorestframework mysqlclient`", onde serão instalados os seguintes pacotes:

- `django` (obrigatório): *Framework* que oferece recursos para a criação de aplicações web.
- `djangorestframework` (obrigatório): Extensão do DRF que simplifica o desenvolvimento de APIs RESTful.
- `mysqlclient` (obrigatório): *Driver* do MySQL para o Python, permite a conexão com um banco de dados MySQL e a execução de consultas SQL.

Para um projeto API RESTFul de controle de usuários, a inicialização é feita com o comando "`django-admin startproject crud_drf_cbv`". Com este comando será criada uma pasta como o nome do projeto dentro do diretório atual, isto evita o conflito com módulos externos. Além disso, um arquivo principal com nome "`manage.py`" também será gerado, contendo as informações primárias do projeto. Dentro do projeto `Django` criado anteriormente, será utilizado o comando "`python manage.py startapp usuarios`" para iniciar um aplicativo com nome "usuarios", que ajudará na modularização do projeto. A estrutura do projeto e aplicação deverá seguir o modelo da Figura 6.16, onde a aplicação gerada fica dentro do projeto principal.

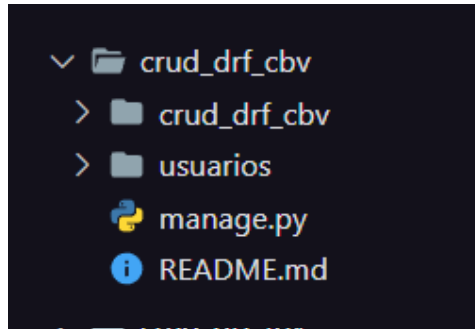


Figura 6.16: Estrutura de arquivos inicial

Com a inicialização do projeto e aplicação feitas, é preciso fazer as configurações iniciais. Na raiz do projeto, no arquivo "`settings.py`", será inserido o "rest_framework" e a aplicação (neste exemplo o nome "usuarios") na variável "INSTALLED_APPS". Este processo é necessário para a indicação ao `Django` que a aplicação faz parte do projeto e que deve ser carregada quando o mesmo for iniciado. O "rest_framework" define a dependência do projeto em relação ao DRF, em que o `Django` verificará se o DRF está instalado no ambiente *Python* antes de iniciar o projeto, exibindo um erro caso o DRF não esteja instalado. Além disso, o `Django` carregará automaticamente os arquivos de configuração do DRF, como as definições

de roteamento de URL. A estrutura da variável “INSTALLED_APPS” deve ser semelhante a Listagem 6.11.

```
INSTALLED_APPS = [
    # outras importações
    "rest_framework",
    "usuarios",
]
```

Listagem 6.11: Estrutura da variável ”INSTALLED_APPS”

No mesmo arquivo, na variável “DATABASES”, serão definidas as configurações de conexão com o banco de dados. Nela serão definidos os parâmetros de conexão, como nome, usuário, senha, *host*, porta e *engine*. Os dados do banco de dados para a conexão seguem os passos apresentados na Seção 6.1.1. Esta configuração de conexão deve seguir o modelo da Listagem 6.12.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

Listagem 6.12: Configuração do banco de dados

Após a configuração do projeto e conexão com o banco de dados, é feita a criação do modelo que a API utilizará para executar as operações CRUD. No `Django`, um modelo é utilizado para gerar as tabelas do banco de dados, colunas, relacionamentos entre as tabelas e várias restrições, além de fornecer modelos já prontos, como os de autenticação. Dentro do arquivo “models.py”, localizado na raiz do diretório da aplicação criada nos passos anteriores, será criada uma classe “Usuario”. A Listagem 6.13 apresenta a estrutura da mesma com a definição dos campos de nome, *email* e idade. A classe Meta é usada para definir metadados adicionais sobre o modelo, em que pode ser especificado, por exemplo, o nome da tabela do banco de dados a ser criada, que terá o nome “usuarios_cbv”. O método “__str__” é definido para representar o objeto do modelo como uma *string*. Neste caso, é retornado o nome do usuário.

```python
from django.db import models

class Usuario(models.Model):
    nome = models.CharField(max_length=255)
    email = models.EmailField(max_length=45)
    idade = models.IntegerField()

    class Meta:
        db_table = 'usuarios_cbv'

    def __str__(self):
        return self.nome
```

Listagem 6.13: Configuração modelo Usuário

O `Django` inclui um painel administrativo que permite gerenciar o conteúdo de uma aplicação. Esse painel é exibido automaticamente ao criar um projeto. No arquivo "`admin.py`", na raiz da aplicação "usuarios", é feita a configuração de registro dos modelos, conforme a Listagem 6.14. Ele é responsável por registrar o modelo "Usuario", criado na Listagem 6.13 para que seja gerenciado por meio do painel administrativo. Isso significa que, ao acessar o painel, haverá uma seção para gerenciamento de usuários, onde será possível listar, criar, atualizar e deletar registros da tabela "usuarios", apresentada na Figura 6.21.

```python
from django.contrib import admin
from .models import Usuario

admin.site.register(Usuario)
```

Listagem 6.14: Configuração modelo Usuário na página admin

Como explicado na Seção 6.4.3, é necessário um processo de serialização para fazer a conversão de objetos ou listas de objetos retornados pelo banco de dados em tipos de dados compatíveis com a estrutura web, como o JSON. Para estas configurações será feito um arquivo na raiz da aplicação nomeado de "serializers.py". Neste projeto foi utilizado o "ModelSerializer" para fazer a serialização, como apresenta a Listagem 6.15. Na classe "Meta"é definido o modelo com o qual o serializador deve trabalhar, e são especificados quais campos do modelo serão serializados. Neste caso, o serializador retornará um JSON contendo o id, nome, *email* e idade de um usuário, correspondentes aos campos do modelo criado na Listagem 6.13.

```
from rest_framework import serializers
from .models import Usuario

class UsuarioSerializer(serializers.ModelSerializer):
    class Meta:
        model = Usuario
        fields = ['id', 'nome', 'email', 'idade']
```

Listagem 6.15: Configuração serializador Usuário

As solicitações HTTP no DRF são tratadas na camada de visualização, que as recebe. Após o processamento, a API retorna as respostas correspondentes. No DRF existem dois tipos principais de *views*: as baseadas em classes, que oferecem recursos de *mixins*, e as *views* baseadas em funções, que são mais simples e flexíveis. Neste projeto será utilizado o modelo baseado em classes, dentro do arquivo "views.py" localizado na raiz da aplicação. O equivalente também pode ser feito para visualizações baseadas em funções.

A Listagem 6.16 implementa a classe "ListaUsuario", que é uma subclasse da classe APIView. As duas operações de CRUD, "listar" e "criar", são definidas por meio dos métodos "get" e "post", referenciando os métodos HTTP GET e POST. No método "get", a *view* busca todos os objetos do tipo "Usuario" do banco de dados utilizando a classe "Usuario.objects.all()" e, em seguida, serializa esses objetos utilizando a classe "UsuarioSerializer", criado na Listagem 6.15. O argumento "many=True" passado ao serializador indica que há vários objetos a serem serializados, em vez de um único objeto. Já no método "post", a *view* recebe uma requisição com os dados do novo usuário a ser adicionado, que são validados pelo serializador "UsuarioSerializer". Caso os dados sejam válidos, o novo usuário é salvo no banco de dados com o método "save()", e uma mensagem de sucesso é retornada como resposta com o código de status 201. Caso contrário, uma mensagem de erro com o código de status 400 é retornada.

```
class ListaUsuario(APIView):
    def get(self, request, format=None):
        usuario = Usuario.objects.all()
        serializer = UsuarioSerializer(usuario, many=True)
        return Response(serializer.data)

    def post(self, request, format=None):
        serializer = UsuarioSerializer(data=request.data)

        if serializer.is_valid():
            serializer.save()
            return Response({'mensagem': 'Usuário adicionado com
                sucesso!'}, status=status.HTTP_201_CREATED)

        return Response(serializer.errors, status=status.
            HTTP_400_BAD_REQUEST)
```

Listagem 6.16: Configuração da *views* dos métodos GET e POST

A implementação dos verbos HTTP: PUT e DELETE são implementados na classe "DetalhesUsuario", que também é uma subclasse da APIView, nela são feitas as definições dos métodos para atualizar e excluir um usuário, como apresentado na Listagem 6.17. O método "get_object(pk)" é uma função auxiliar para recuperar o usuário com o id correspondente a pk (*primary key*). Se não houver usuário com o id fornecido, ele retornará um erro 404. O método "put" recebe a requisição HTTP PUT com dados JSON atualizados para um usuário e, em seguida, valida os dados usando o serializador "UsuarioSerializer". Se os dados forem válidos, o serializador salva as atualizações no banco de dados e retorna uma mensagem de sucesso. Se os dados não forem válidos, o método retornará uma mensagem de erro 400. O método "delete" recebe uma requisição HTTP DELETE e remove o usuário correspondente ao id pk. Caso o usuário exista, é retornado um status 204.

```
class DetalhesUsuario(APIView):
    def get_object(self, pk):
        try:
            return Usuario.objects.get(pk=pk)
        except Usuario.DoesNotExist:
            raise Http404

    def put(self, request, pk, format=None):
        usuario = self.get_object(pk)
        serializer = UsuarioSerializer(usuario, data=request.data)

        if serializer.is_valid():
            serializer.save()
            return Response({'mensagem': 'Usuário atualizado com
                sucesso!'})

        return Response(serializer.errors, status=status.
            HTTP_400_BAD_REQUEST)

    def delete(self, request, pk, format=None):
        usuario = self.get_object(pk)
        usuario.delete()

        return Response(status=status.HTTP_204_NO_CONTENT)
```

Listagem 6.17: Configuração das *views* dos métodos GET e POST

As rotas da aplicação são implementadas no arquivo "`urls.py`" localizado na raiz da aplicação. A Listagem 6.18 apresenta um exemplo de duas rotas. Para a configuração das rotas globais do projeto, as mesmas são configuradas dentro do arquivo "`urls.py`" no diretório raiz do projeto, seguindo a Listagem 6.19. As rotas criadas da aplicação são:

1. "usuarios/": esta rota aponta para a classe de visualização "ListaUsuario" definida no arquivo "`views.py`" do aplicativo "usuarios". Quando um cliente da API acessa essa rota usando o método GET, todos os usuários cadastrados no banco de dados são retornados como resposta em formato JSON, como

apresenta a Figura 6.17. Quando um cliente usa o método POST, um novo usuário é adicionado ao banco de dados de acordo com as informações passadas no corpo da requisição, como mostra a Figura 6.18.

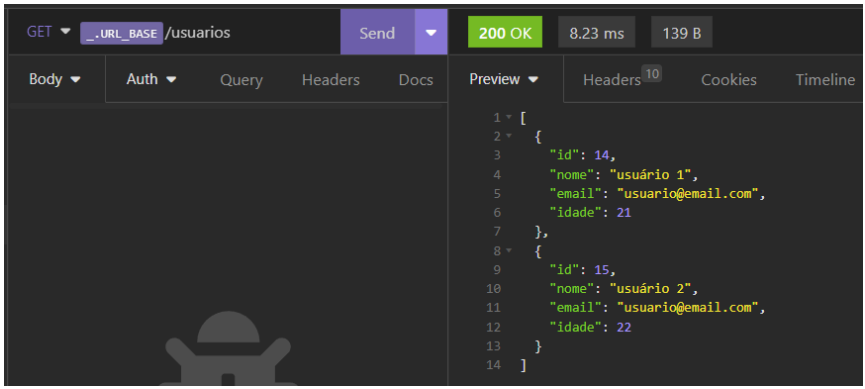


Figura 6.17: Utilização do método GET (API Django REST)

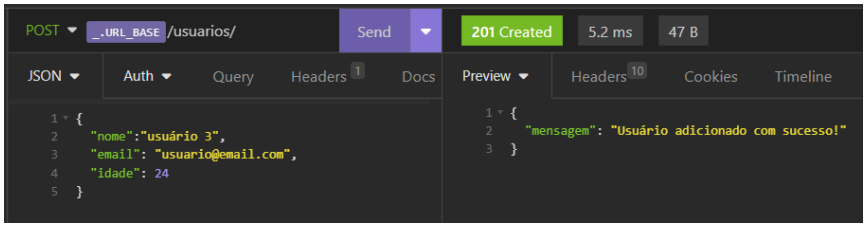


Figura 6.18: Utilização do método POST (API Django REST)

2. "`usuarios/<int:pk>/`": esta rota aponta para a classe de visualização "DetalhesUsuario" definida no arquivo "`views.py`" do aplicativo "usuarios". Quando um usuário acessa essa rota usando o método PUT, os dados do usuário com o ID fornecido são atualizados com os novos dados fornecidos no corpo da solicitação, como mostra a Figura 6.19. Quando um usuário usa o método DELETE, o usuário com o ID fornecido é excluído do banco de dados, apresentado na Figura 6.20.

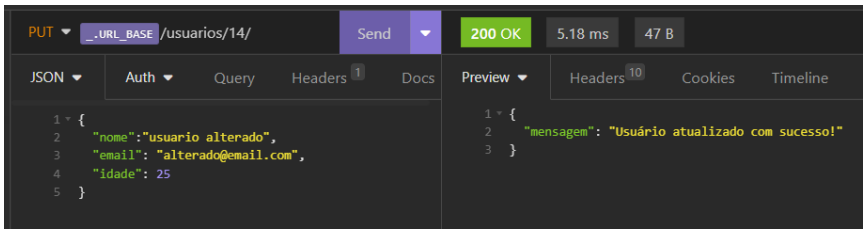


Figura 6.19: Utilização do método PUT (API Django REST)

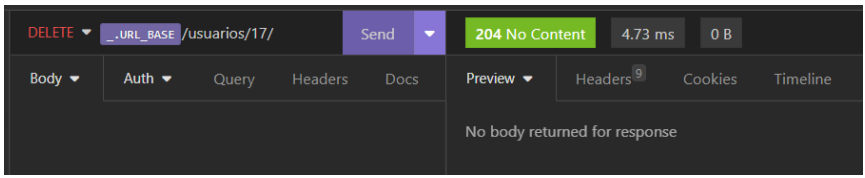


Figura 6.20: Utilização do método DELETE (API Django REST)

```python
from django.urls import path
from usuarios import views

urlpatterns = [
    path('usuarios/', views.ListaUsuario.as_view()),
    path('usuarios/<int:pk>/', views.DetalhesUsuario.as_view()),
]
```

Listagem 6.18: Configuração das URLs da aplicação

```python
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('usuarios.urls')),
]
```

Listagem 6.19: Configuração das URLs do projeto

Para fazer a sincronização do banco de dados com as alterações feitas nos modelos do aplicativo, é utilizado o comando "`python manage.py makemigrations`". Esse comando cria arquivos de migração de acordo com os modelos no diretório "*migrations*" do aplicativo. Em seguida, os arquivos de migração criados precisam ser aplicados ao banco de dados para atualizá-lo, esta migração é feita executando o comando "`python manage.py migrate`". Por padrão, também são criadas as

migrações de autenticação (*auth*) que incluem as tabelas do banco de dados para gerenciar usuários, grupos e permissões. Além disso, as migrações de *admin* são criadas, incluindo as tabelas de banco de dados para suportar o painel de administração do `Django`. Ao executar o comando "`python manage.py runserver`", é possível acessar a API tanto por meio do cliente web quanto pelo painel administrativo do `Django`. Isso permite realizar as operações necessárias, conforme mostrado na Figura 6.21.

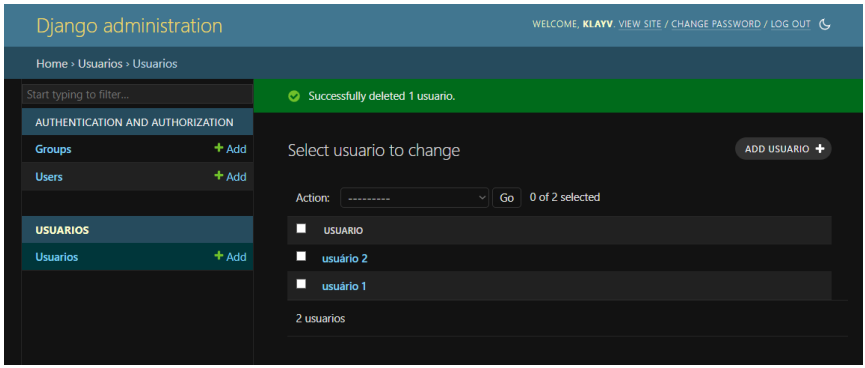


Figura 6.21: Painel administrativo do Django

6.5 Introdução ao .NET

O ecossistema do .NET[18] começou em 2002 com o lançamento da *Microsoft* de uma plataforma de desenvolvimento para criar aplicativos do *Windows*, chamada de `.NET Framework`. Em 2014, foi introduzido o `.NET Core`, um sucessor multiplataforma e de software livre do `.NET Framework`, utilizado para aplicativos em *Linux*, *macOS* e *Windows*. A plataforma `.NET` além dos sistemas operacionais citados, também dá suporte em ambientes emulados, além da possibilidade de ser utilizada em arquiteturas Arm64, x64 e x32. Os aplicativos e as bibliotecas do .NET podem ser criados utilizando tanto a *Common Language Infrastructure* (CLI) do `.NET` quanto um *Integrated Development Environment* (IDE) como o Visual Studio.

Os programas em *C#* são executados no `.NET` em um sistema de execução virtual chamado *Common Language Runtime* (CLR). O CLR é o *runtime* presente em todos os aplicativos `.NET`, com recursos fundamentais como coleta de lixo, segurança de memória, suporte de alto nível a linguagens de programação e design multiplataforma. Este *runtime* é compatível com várias linguagens de programação da *Microsoft*, como *C#*, *F#* e *Visual Basic*, e é multiplataforma. A capacidade de compilar para diferentes sistemas operacionais e arquiteturas permite que o

[18] ⟨https://learn.microsoft.com/pt-br/dotnet/fundamentals/⟩

aplicativo não precise ser recompilado para ser executado em novos ambientes, bastando instalar um *runtime* diferente.

Por ser uma linguagem orientada a objetos, o *C#* segue o paradigma de programação orientada a objetos POO. Sua construção e desenvolvimento são baseados em objetos, que são divididos em classes, e se comunicam entre si por meio de métodos. É possível utilizar vários conceitos deste paradigma, como herança, encapsulamento e polimorfismo, no desenvolvimento em *C#*. O POO permite a modularização, a reutilização e a facilidade de manutenção do código. Além disso, o *C#* também suporta outros paradigmas, como o paradigma funcional e a programação assíncrona (MICROSOFT, 2023).

A execução das aplicações em *C#* do `.NET` acontece por meio de uma compilação em uma linguagem intermediária (IL), que é um formato de código compactado com suporte em qualquer SO e arquitetura. O `.NET` dá suporte a modelos de compilação *Ahead-Of-Time* (AOT) e JIT, como, por exemplo, na compilação padrão JIT nas plataformas *Android*, *macOS* e *Linux*. O compilador JIT tem a capacidade de adaptar o código para funcionar melhor em um ambiente específico, considerando o sistema operacional e o *hardware* do dispositivo em que o aplicativo é executado. O AOT proporciona uma inicialização mais rápida de um aplicativo, pois realiza a compilação antes da execução. Este compilador possui como desvantagem a necessidade de compilar o aplicativo em diferentes sistemas operacionais e arquiteturas de *hardware* (MICROSOFT, 2023).

6.5.1 Gerenciador de pacotes NuGet e Entity Framework Core

NuGet[19] é o mecanismo com suporte da *Microsoft* para compartilhamento de código, que define como os pacotes do `.NET` são criados, hospedados e consumidos, além de fornecer as ferramentas para cada uma dessas funções. Por meio da ferramenta `NuGet` os desenvolvedores podem compartilhar pacotes em *hosts* públicos ou privados e consumi-los em seus projetos.

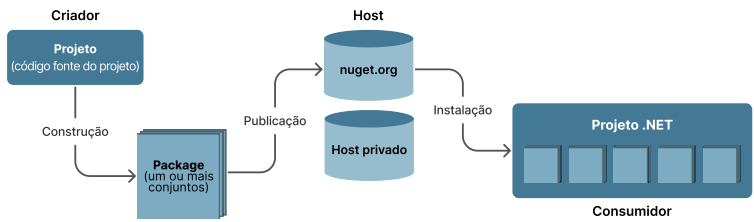


Figura 6.22: Disponibilização e consumo de pacotes por meio do NuGet

[19] ⟨https://learn.microsoft.com/pt-br/nuget⟩

O *Entity Framework* (EF)[20] é um *framework* de mapeamento objeto-relacional ORM. O *Entity Framework* Core[21] é uma versão mais leve e flexível do *Entity Framework*. Ambos foram criados pela *Microsoft* e são utilizados para o acesso e a manipulação de dados em um banco. O *Entity Framework Core* (EFC) é compatível com vários tipos de banco de dados, como `SqlServer`, `PostgreSQL` e `MySql`. No processo de desenvolvimento, o EFC também é utilizado como um *framework* ORM, no qual é possível realizar o mapeamento automático das classes e propriedades dos modelos de dados para as tabelas e colunas do banco de dados, sem a necessidade de escrever código SQL manualmente. O *framework* não apenas oferece suporte à linguagem de consulta SQL, mas também permite o uso da linguagem de consulta LINQ. Além disso, ele simplifica o processo de alteração no banco de dados por meio das migrações, proporcionando atualizações mais eficientes e organizadas.

6.5.2 Ambiente de desenvolvimento para o ASP.NET Core

Assim como nos outros *frameworks* mencionados, o ambiente para desenvolvimento em *C#* pode ser configurado em vários sistemas operacionais, incluindo *Windows*, *MacOS* e *Linux*. Há vários IDEs disponíveis para o desenvolvimento em *C#*, tais como Visual Studio Code, JetBrains Rider e Visual Studio, sendo este último o mais utilizado, pois permite a implantação e depuração de aplicações. Para desenvolver em *C#*, é necessário ter instalado o SDK do `.NET`, que inclui o compilador *C#* e as bibliotecas necessárias para criar aplicativos `.NET`.

O SDK é um conjunto de bibliotecas e ferramentas que permite aos desenvolvedores criar e desenvolver aplicações e bibliotecas. Ele pode ser baixado e instalado a partir do site oficial da *Microsoft*[22], conforme o sistema operacional. A Figura 6.23 mostra a página inicial do instalador, onde, ao clicar em "*Install*", serão instaladas todas as dependências necessárias. O SDK contém alguns componentes utilizados para a criação e execução de aplicações, como o `.NET` CLI, o *runtime*, as bibliotecas .NET e o *driver* 'dotnet'. Após o *download* e instalação, é possível verificar se ela foi realizada com sucesso executando o comando "`dotnet --version`" no console do terminal, no qual deverá aparecer a versão do SDK instalado.

[20] ⟨https://learn.microsoft.com/pt-br/ef/⟩
[21] ⟨https://learn.microsoft.com/pt-br/ef/core/⟩
[22] ⟨https://dotnet.microsoft.com/pt-br/download⟩

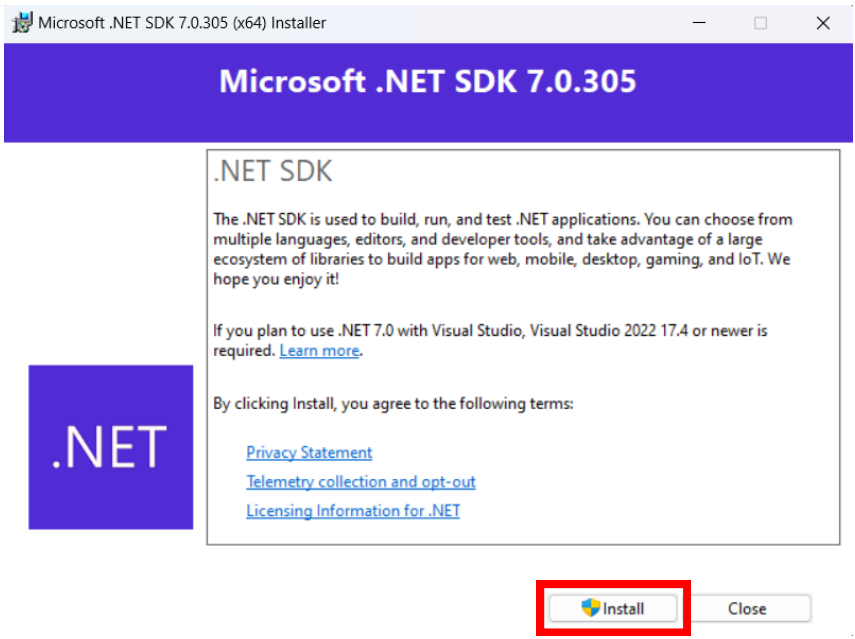


Figura 6.23: Página inicial do instalador do SDK

6.5.3 Exemplo CRUD ASP.NET Core

A criação de um CRUD em `ASP.NET Core` pode ser feita de duas maneiras, nas quais podemos ter um maior controle sobre a implementação, tendo maior personalização na criação manual de todas as partes do CRUD, como nos modelos, controladores e rotas, suprindo as necessidades específicas de cada projeto. Contudo, é maior a demanda de tempo para o desenvolvimento e esforço na criação de todas as partes de uma forma individual. Existem também geradores de código para auxiliar o processo de desenvolvimento, que realizam a geração automática de código com base nos modelos de dados e nas definições do banco de dados. A utilização do Visual Studio requer a instalação do ambiente para desenvolvimento `ASP.NET Core` que é fornecido na inicialização do editor, como mostra a Figura 6.24 com a opção destacada em vermelho.

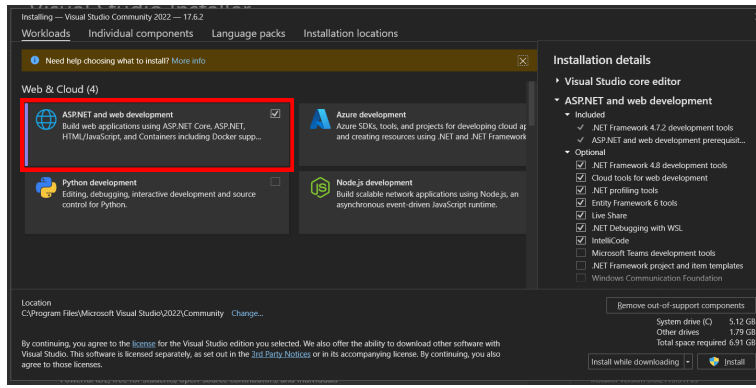


Figura 6.24: Instalação do ambiente ASP.NET

A inicialização do projeto é feita após a instalação do ambiente `ASP.NET`, no qual é possível criar um projeto baseado em vários *templates*. O CRUD será feito com base no *template* `ASP.NET` Core Web API, que já vem com um exemplo pronto de um serviço RESTful HTTP, além de poder ser utilizado para `ASP.NET` Core *Model View Controller* (MVC). Após a escolha do *template*, é necessário fazer as configurações iniciais do projeto, como a escolha do nome, do diretório e da versão do `.NET`, que foi escolhida como 7.0, além da configuração de autenticação, `Docker` e outros. Quando o projeto é criado, uma estrutura já é apresentada com um exemplo inicial, como mostrado na Figura 6.25.

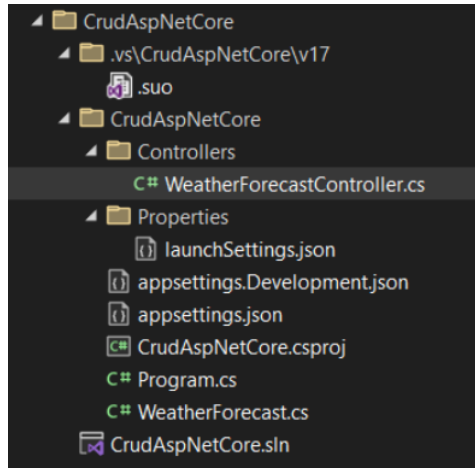


Figura 6.25: Árvore de arquivos inicial

Utilizando o gerenciador de pacotes `NuGet`, citado na Seção 6.5.1, serão instalados os seguintes pacotes obrigatórios:

- "Microsoft.EntityFrameworkCore": fornece as funcionalidades básicas do *Entity Framework Core*, como a definição de classes de entidade, contexto de banco de dados, configurações de relacionamento e recursos para consultas e atualizações no banco de dados. Ele é necessário para usar o EFC no projeto.

- "Microsoft.EntityFrameworkCore.Design": contém as ferramentas de *design* do *Entity Framework Core*, usadas para criar e modificar o modelo de entidade. Ele fornece funcionalidades, como *scaffolding* (geração de código a partir do banco de dados), *Migrations* (gerenciamento de esquema do banco de dados) e outras ferramentas úteis para o desenvolvimento com o EF Core.

- "Microsoft.EntityFrameworkCore.Tools": contém comandos da CLI do EF Core, permitindo a execução de tarefas relacionadas ao banco de dados, como a criação e aplicação de migrações, *scaffolding*, entre outras operações. Essas ferramentas serão úteis durante o desenvolvimento e manutenção do banco de dados, como no processo de *migrations*.

- "Pomelo.EntityFrameworkCore.MySql": é uma extensão específica para suportar o MySQL no *Entity Framework Core*, o mesmo banco de dados utilizado nas aplicações em `Node.js` e `Django` REST. Ele adiciona o provedor de banco de dados do `MySql` ao EF Core, permitindo a conexão a bancos de dados `MySQL`. Além de fornecer suporte completo para consultas, atualizações e migrações no `MySQL` com o EF Core.

Como foi feito nos projetos em `Node.js` e `Django`, será criado o modelo Usuário para a estruturação do banco de dados. Este processo é feito no arquivo "UsuarioModel.cs" dentro da pasta *Models*, criada na raiz do projeto. A Listagem 6.20 apresenta a estrutura do modelo de dados do usuário com as propriedades "Id", "Nome", "Email"e "Idade". O atributo [*Key*] na propriedade 'Id' indica que é a chave primária da entidade. Os atributos [*MaxLength*] em 'Nome' e 'Idade' definem os limites máximos de caracteres. Para a criação do banco de dados, utilizou-se o método *Code First*, no qual as classes de entidade e o contexto do banco de dados foram definidos diretamente no código.

```
namespace CrudUsuarios.Models
{
    public class UsuarioModel
    {
        [Key]
        public long Id { get; set; }

        [MaxLength(255)]
        public string Nome { get; set; }

        [MaxLength(45)]
        public string Email { get; set; }

        public int Idade { get; set; }
    }
}
```

Listagem 6.20: Modelo Usuário

Outra etapa necessária é a criação de um contexto de dados (*data context*). Ela é responsável por mapear as classes de entidade (criadas no passo anterior) para tabelas do banco de dados e gerenciar as operações de acesso aos dados. Este processo é feito no arquivo "UsuarioDbCotext.cs", localizado na pasta "Data", criada na raiz do projeto. A Listagem 6.21 mostra como é a estrutura, na qual possui a classe UsuarioDbContext, que herda de DbContext, fornecida pelo *Entity Framework Core*. Essa classe representará o contexto de dados e fornecerá acesso às entidades do modelo. O 'DbSet' permite o acesso e manipulação dos dados da tabel"Usuarios" por meio do contexto de banco de dados. Sendo assim, para realizar operações na tabela "Usuarios" no banco de dados, será utilizada a entidade "UsuarioModel" da aplicação.

```
namespace CrudUsuarios.Data
{
public class UsuarioDbContext : DbContext
{
    public UsuarioDbContext(DbContextOptions<UsuarioDbContext>
    options) : base(options)
    {

    }

 public DbSet<UsuarioModel> Usuarios { get; set; }
}
}
```

Listagem 6.21: Contexto de dados Usuário

Para estabelecer a conexão entre a aplicação e o banco de dados, é necessário definir a *string* de conexão no arquivo "`appsettings.json`", localizado na raiz da aplicação, seguindo o modelo da Listagem Listagem 6.22. Essa configuração contém informações essenciais para identificar o servidor, a porta, as credenciais de autenticação e outros detalhes do banco de dados. Os dados do banco de dados para a conexão seguem os passos apresentados na Seção 6.1.1. A configuração da *string* de conexão pode variar dependendo do ambiente de execução da aplicação, como os ambientes de desenvolvimento e de produção. Existem diferentes maneiras de configurá-la, utilizando extensões de arquivos específicas para cada ambiente.

```
"ConnectionStrings": {
  "DefaultConnection": "Server=localhost; Port=3306;
  initial catalog=nome_tabela; uid=root ;pwd=senha;"
}
```

Listagem 6.22: *String* de conexão do banco de dados

Após a configuração da *string* de conexão, é necessário configurar o contexto de dados no arquivo "`Program.cs`", situado na raiz do projeto, conforme demonstrado na Listagem Listagem 6.23. Neste arquivo, a *string* de conexão do arquivo de configuração será lida e, em seguida, o contexto de dados será configurado para utilizar essa *string* de conexão com o banco de dados MySQL.

```
using CrudUsuarios.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

var connectionStringMysql = builder.Configuration.GetConnectionString(
    "ConnectionMysql");
builder.Services.AddDbContext<UsuarioDbContext>(options => options.
    UseMySql(
        connectionStringMysql,
        ServerVersion.AutoDetect(connectionStringMysql)
        )
);
```

Listagem 6.23: Configuração do contexto de dados

Com todas as configurações da conexão feitas, é possível executar o comando "`add-migration init`" no console do gerenciador de pacotes. Esse comando criará uma migração inicial com base no modelo de dados atual. Após a execução, será gerado um arquivo com nome 'init' contendo os contextos de migração. O *Entity Framework* criará uma pasta 'Migrations' na raiz da aplicação com dois arquivos, um de *Snapshot* do modelo de dados e outro com uma classe de migração contendo o código necessário para criar o esquema do banco de dados inicial. Após a criação da migração, o comando "`update-dataset`" aplicará a migração ao banco de dados

`MySQL`, criando as tabelas, colunas e restrições de acordo com o modelo de dados previamente definido.

O processo de criação dos controladores RESTful foi realizado utilizando os próprios recursos da plataforma do Visual Studio e do EF. Nele é possível gerar código por meio do *scaffold*, permitindo realizar as operações CRUD (*Create*, *Read*, *Update*, *Delete*) nas entidades do modelo de dados. As Figuras 6.26 e 6.27 apresentam os passos para a criação e a configuração do controlador. Com isso, serão escolhidos, respectivamente, os tipos de geração de código e os modelos e as classes do contexto de dados a serem utilizados como base. Neste caso, foi escolhida a geração de controladores com ações utilizando o EF, além de se basear no modelo "UsuarioModel"e no contexto "UsuarioDbContext", que foram anteriormente criados. O arquivo a ser gerado para os controladores terá o nome "UsuarioController".

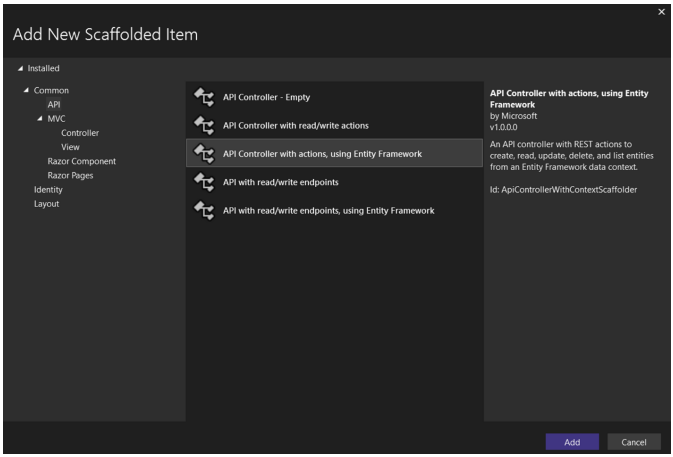


Figura 6.26: Geração dos controladores com *Scaffold*

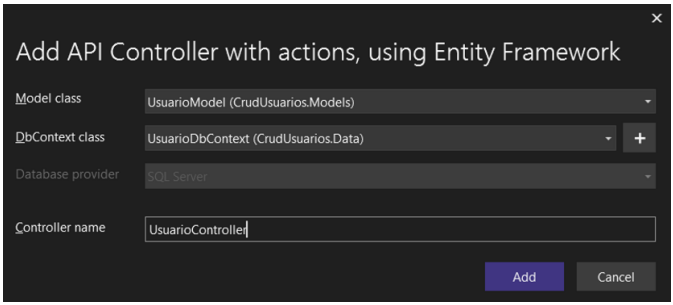


Figura 6.27: Configuração dos controladores

Após a configuração e criação de todos os arquivos, a aplicação pode ser executada pelo próprio Visual Studio, utilizando a interface gráfica, ou por meio do comando "`dotnet run`", que pode ser executado em um terminal no diretório raiz do projeto. O servidor ficará então disponível para acesso, com a geração automática de documentação padrão em *Swagger*, que permite a visualização e a interação gráfica com a API. Nele serão exibidas informações sobre os *endpoints* disponíveis, parâmetros aceitos, modelos de dados, entre outros detalhes que serão apresentados em seguida. O controlador do método GET gerado automaticamente é apresentado na Listagem Listagem 6.24, contendo dois *endpoints*, servindo tanto para leitura de todos os usuários quanto para um usuário com 'Id' específico. O controlador retornará uma lista de objetos 'UsuarioModel' ou um objeto específico, dependendo da rota. Um exemplo de uso da rota que retorna todos os usuários pode ser visto na Figura 6.28.

```
// GET: api/Usuario
[HttpGet]
public async Task<ActionResult<IEnumerable<UsuarioModel>>>
    GetUsuarios()
{
  if (_context.Usuarios == null)
  {
      return NotFound();
  }
    return await _context.Usuarios.ToListAsync();
}

// GET: api/Usuario/5
[HttpGet("{id}")]
public async Task<ActionResult<UsuarioModel>> GetUsuarioModel(
    long id)
{
  if (_context.Usuarios == null)
  {
      return NotFound();
  }
    var usuarioModel = await _context.Usuarios.FindAsync(id);

    if (usuarioModel == null)
    {
        return NotFound();
    }

    return usuarioModel;
}
```

Listagem 6.24: Controlador do método GET

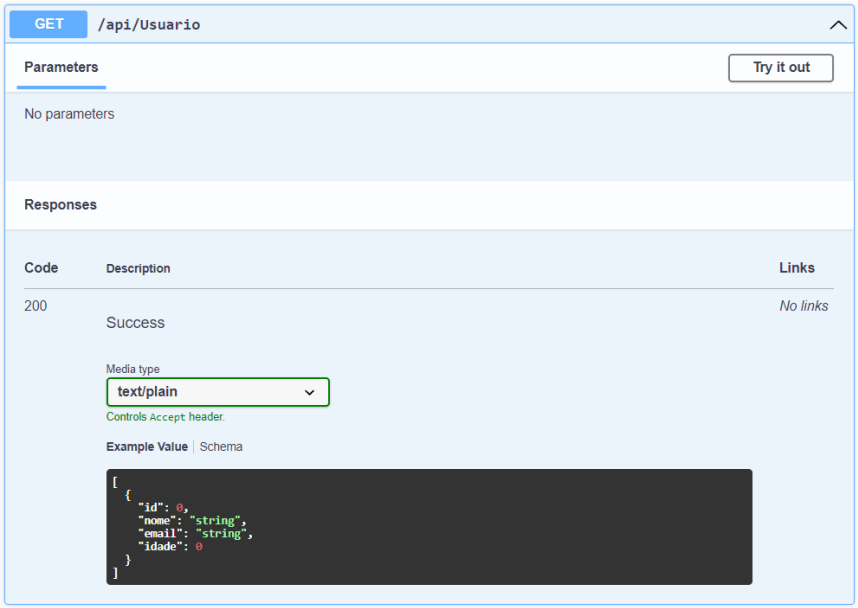


Figura 6.28: Exemplo método GET (API ASP.NET Core)

O método POST gerado é apresentado na Listagem 6.25. Possui o decorador "[HttpPost]"indicando que é acionado na rota "api/Usuario"por meio do método POST recebendo um objeto UsuarioModel no corpo da requisição, que contém os dados do usuário a serem criados. Caso a tabela exista no contexto de dados, o usuário será adicionado utilizando o método "Add()", salvando as alterações assíncronamente por meio do método "SavechangesAsync()". O "CreatedAtAction"será o retorno que indicará o sucesso da requisição. Um exemplo de uso do método POST pode ser visto na Figura 6.29.

```
// POST: api/Usuario
[HttpPost]
public async Task<ActionResult<UsuarioModel>> PostUsuarioModel(
    UsuarioModel usuarioModel)
{
  if (_context.Usuarios == null)
  {
      return Problem("Entity set 'UsuarioDbContext.Usuarios'
          is null.");
  }
    _context.Usuarios.Add(usuarioModel);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetUsuarioModel", new { id =
        usuarioModel.Id }, usuarioModel);
}
```

Listagem 6.25: Controlador do método POST

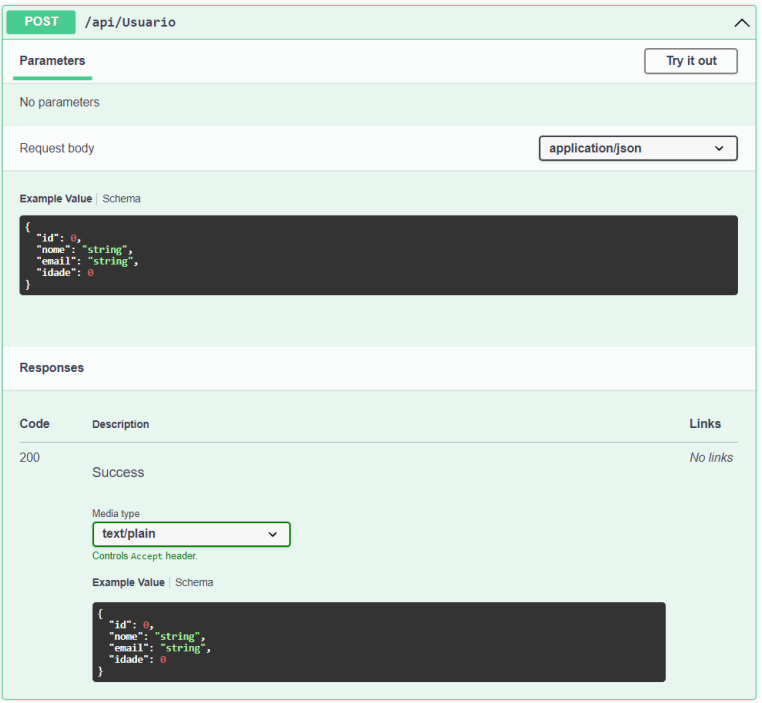


Figura 6.29: Exemplo método POST (API ASP.NET Core)

A Listagem 6.26 apresenta a geração do método PUT. O decorador '[HttpPut("id")]' faz a indicação de que o controlador lida com requisições do tipo PUT na rota 'api/Usuario/id', no qual 'id' é um parâmetro de rota que identifica o usuário que será alterado. O controlador inicialmente fará uma verificação para saber se o 'Id' do usuário repassado na rota é o mesmo do corpo da requisição, caso não seja, será retornado um 'BadRequest()'. Uma verificação também é realizada posteriormente para confirmar se o usuário está no banco de dados. Caso exista, a operação será concluída e retornará com código 204, indicando sucesso. Um exemplo da utilização da rota para alterar os dados de um usuário pode ser visto na Figura 6.30.

```
// PUT: api/Usuario/{id}
[HttpPut("{id}")]
public async Task<IActionResult> PutUsuarioModel(long id,
    UsuarioModel usuarioModel)
{
    if (id != usuarioModel.Id)
    {
        return BadRequest();
    }

    _context.Entry(usuarioModel).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!UsuarioModelExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

Listagem 6.26: Controlador do método PUT

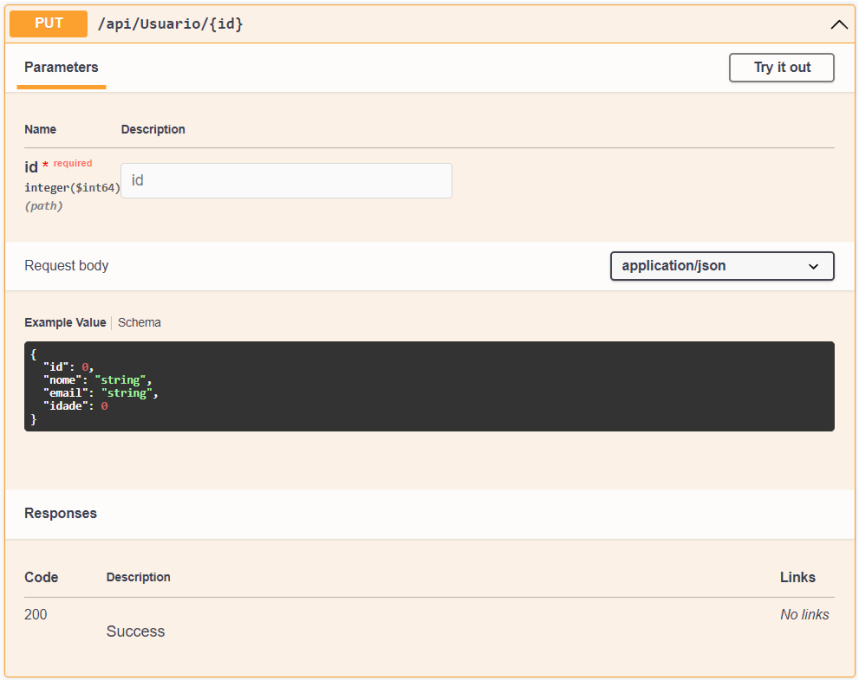


Figura 6.30: Exemplo método PUT (API ASP.NET Core)

Ao finalizar os controladores para um CRUD RESTful, é apresentado o controlador do método POST na Listagem 6.27. Semelhante ao decorador do método PUT, o '[HttpDelete("id")]' indicará que o controlador será utilizado na rota "api/Usuario/id"referente ao método DELETE. O mesmo fará a verificação para saber se o 'id' repassado no parâmetro de rota existe no banco de dados, caso exista, o usuário será deletado e retornará 'NoContent()' com o método 204, indicando sucesso na operação. Um exemplo de uso pode ser visto na Figura 6.31.

```
// DELETE: api/Usuario/{id}
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteUsuarioModel(long id)
{
    if (_context.Usuarios == null)
    {
        return NotFound();
    }

    var usuarioModel = await _context.Usuarios.FindAsync(id);

    if (usuarioModel == null)
    {
        return NotFound();
    }

    _context.Usuarios.Remove(usuarioModel);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

Listagem 6.27: Controlador do método DELETE



Figura 6.31: Exemplo método DELETE (API ASP.NET Core)

[H]

CARACTERÍSTICAS		Node.js	DRF	ASP.NET Core
Sistemas operacionais	**Windows**	Sim	Sim	Sim
	Linux	Sim	Sim	Sim*
	MacOS	Sim	Sim	Sim*

Tabela 6.2: Suporte das tecnologias aos principais SOs.

** É importante ressaltar que apenas a estrutura do ASP.NET Core é multiplataforma, diferentemente da estrutura .NET Framework, que tem suporte apenas para o sistema operacional Windows.*

6.6 Comparações entre as tecnologias de banc-end

Nesta seção, serão apresentados e discutidos os resultados da comparação entre as tecnologias back-end. Esta seção apresenta uma análise das características do `Node.js`, `Django REST Framework` (DRF) e `.NET Core`. O levantamento e análise das características envolveram a abordagem exploratória para realizar uma análise documental dessas tecnologias. Para isso, foram identificadas características essenciais para avaliar tecnologias de desenvolvimento, destacando suas funcionalidades, recursos, contextos de aplicação e outros fatores relevantes. No estudo empírico, foi desenvolvido um *software*-alvo de forma uniforme em todas as tecnologias selecionadas, garantindo que os aplicativos fossem equivalentes em termos de funcionalidade e recursos. Isso foi feito com o propósito de permitir uma comparação precisa das métricas de desempenho, como tempo de resposta e consumo de recursos, entre `Node.js`, `Django REST Framework` (DRF) e `.NET Core`.

As características das tecnologias foram obtidas por meio de estudos bibliográficos relacionados e de documentações oficiais de cada uma delas.

6.6.1 Sistemas Operacionais suportados

A Tabela 6.2 apresenta o suporte ao desenvolvimento, utilizando as tecnologias em estudo, em diferentes sistemas operacionais. Todas as tecnologias são compatíveis com os sistemas operacionais citados. Apesar da semelhança, o `.NET Core` apresenta melhor desempenho, por ser uma nova estrutura reescrita do `.NET Framework`. Para casos de migração de estrutura, é fornecido na própria documentação um suporte para tal atualização[23].

[23] ⟨https://learn.microsoft.com/pt-br/aspnet/core/migration/proper-to-2x/?view=aspnetcore-7.0⟩

Tabela 6.3: Suporte das tecnologias aos principais bancos de dados.

CARACTERÍSTICAS		Node.js	DRF	ASP.NET Core
Suporte à Banco de dados	**MySQL**	Sim	Sim	Sim
	PostgreSQL	Sim	Sim	Sim
	SQLite	Sim	Sim	Sim
	MongoDB	Sim	Não*	Sim
	SQL Server	Sim	Sim	Sim

Tabela 6.4: Suporte das tecnologias ao ORM nativo

CARACTERÍSTICAS	Node.js	DRF	ASP.NET Core
ORM nativo	Não	Sim	Não

6.6.2 Compatibilidade com bancos de dados

A Tabela 6.3 apresenta a compatibilidade das tecnologias com diferentes bancos de dados, relacionais e não relacionais. As três tecnologias em estudo têm suporte aos bancos de dados citados, com exceção do `MongoDB` no DRF. Os bancos de dados não relacionais não são oficialmente suportados pelo `Django`, porém há alguns projetos para implementação[24]. Para o suporte a bancos de dados não relacionais, em específico para o `MongoDB`, é citada na wiki do `Django` a utilização da biblioteca `djongo`[25], que faz a tradução de uma *query* SQL para uma *query* do `MongoDB`[26]. É importante mencionar que a plataforma `.NET` é altamente compatível com o `SQL Server`, um banco de dados desenvolvido pela Microsoft. Esses dois componentes funcionam bem juntos, facilitando o desenvolvimento de aplicativos corporativos.

6.6.3 Suporte ao ORM nativo

A Tabela 6.4 apresenta a relação das tecnologias que oferecem suporte ao ORM nativo. Tanto o `Node.js` quanto o `.NET Core` não possuem esta funcionalidade nativamente. No `Node.js`, é possível utilizar bibliotecas de terceiros para a implementação de ORM, como `Prisma`[27], `Sequelize`[28] e `TypeORM`[29]. O `.NET Core`, por sua vez, possui alguns ORMs, sendo o `Entity Framework Core`[30] o mais utilizado, tendo o Dapper[31] como uma alternativa de software livre popular. O DRF é o único que possui esta funcionalidade nativamente, não precisando fazer consultas SQL manualmente, podendo ser abstraída esta interação com o banco de dados.

[24] ⟨https://docs.djangoproject.com/pt-br/4.2/faq/models/#does-django-support-nosql-databases⟩
[25] ⟨https://github.com/doableware/djongo⟩
[26] ⟨https://code.djangoproject.com/wiki/NoSqlSupport⟩
[27] ⟨https://www.prisma.io/⟩
[28] ⟨https://sequelize.org/⟩
[29] ⟨https://typeorm.io/⟩
[30] ⟨https://learn.microsoft.com/pt-br/ef/⟩
[31] ⟨https://github.com/StackExchange/Dapper⟩

Tabela 6.5: Suporte das tecnologias à documentação nativa.

CARACTERÍSTICAS	Node.js	DRF	ASP.NET Core
Documentação nativa	Não	Não*	Sim

** A geração de documentação pelo DRF não é automática, porém há suporte integrado.*

6.6.4 Documentação nativa da API

A Tabela 6.5 apresenta o suporte aos sistemas operacionais mais utilizados para as tecnologias em estudo. Dentre as três, o `.Net` é o único *framework* que possui suporte à geração automática de documentação nativamente. Ele utiliza o `Swagger` como ferramenta para tal, gerando uma interface `SwaggerUI` que apresenta os serviços e os métodos de ação pública[32]. Esta documentação permite realizar o teste da API por meio da interface e verificar informações específicas de cada método.

No DRF, durante a construção do projeto, é gerado um painel administrativo em formato de interface web que permite a configuração e o gerenciamento de conteúdos. A geração de documentação da API não é feita de forma automática, no entanto, existe suporte integrado para a geração de esquemas OpenAPI, que é utilizado para geração de documentação[33]. Como biblioteca de terceiros, é recomendada a utilização do `drf-yasg`[34] e `drf-spectacular`[35]. No `Node.js` também é possível fazer esta configuração utilizando bibliotecas externas, como o `Swagger`. Com isto também pode ser gerada uma interface com `SwaggerUI` assim como no `.Net`.

[32] ⟨https://learn.microsoft.com/pt-br/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-7.0⟩
[33] ⟨https://www.django-rest-framework.org/topics/documenting-your-api/⟩
[34] ⟨https://github.com/axnsan12/drf-yasg/⟩
[35] ⟨https://github.com/tfranzel/drf-spectacular/⟩

Capítulo 7

Interface de programação de aplicações - API

Samyle de Sousa Araújo (UFC)

Fischer Ferreira (UNIFEI)

Este capítulo apresenta uma visão geral das principais abordagens de APIs web, com foco nas **APIs RESTful**, **APIs GraphQL** e **APIs gRPC**. Inicialmente, a Seção 7.1 introduz os conceitos fundamentais relacionados a APIs e serviços web, estabelecendo a base teórica necessária à compreensão das arquiteturas de comunicação empregadas. Em seguida, a Seção 7.2 realiza uma análise comparativa entre os diferentes padrões de APIs abordados, destacando suas características, vantagens e limitações. Por fim, a Seção 7.3 apresenta exemplos práticos de uso das APIs desenvolvidas neste trabalho, com o objetivo de aproximar os conceitos teóricos da implementação prática e subsidiar os experimentos realizados no estudo comparativo.

7.1 Uma visão geral sobre API

Ao longo do processo de evolução da internet, diversas tecnologias emergiram e contribuíram para o desenvolvimento das aplicações web, dentre as quais se destacam as APIs. A sigla API corresponde a *Application Programming Interface* e tem como objetivo principal possibilitar que duas ou mais aplicações se comuniquem de maneira estruturada, flexível e padronizada. A consolidação desse conceito ocorreu tanto no âmbito acadêmico, a partir de trabalhos como o de Fielding (FIELDING, 2000), que introduziu os fundamentos da arquitetura REST, quanto no contexto industrial, impulsionada por iniciativas de grandes empresas de tecnologia, como *Salesforce*, *Amazon* e *eBay*, que passaram a adotar e disponibilizar APIs como parte central de seus ecossistemas digitais (POSTMAN, 2019).

A partir dessa expansão teórica e comercial, as APIs passaram a desempenhar um papel essencial na arquitetura dos serviços digitais contemporâneos. Inicialmente, sua adoção esteve fortemente associada a aplicações comerciais, com foco na integração de sistemas voltados ao comércio eletrônico e à ampliação das vendas online. Contudo, com o amadurecimento das tecnologias web, as APIs expandiram-se para outros domínios, como redes sociais e plataformas colaborativas. Um exemplo representativo é o *Facebook*[1], que, em 2006, lançou uma versão inicial de sua API, permitindo que os usuários compartilhassem informações com aplicações externas (FACEBOOK, 2008). Esse tipo de iniciativa consolidou-se como prática recorrente, e atualmente diversas empresas de software disponibilizam APIs públicas ou restritas com o objetivo de fomentar a integração de seus serviços com aplicações de terceiros.

No processo de integração entre sistemas, a API atua como uma interface intermediária, por meio da qual uma aplicação disponibiliza funcionalidades e dados para consumo por outra. Esses dados, em geral, são armazenados em servidores e podem ser acessados ou modificados conforme regras previamente definidas. Dessa forma, os desenvolvedores utilizam APIs como mecanismos padronizados para leitura, criação, atualização e remoção de informações, abstraindo detalhes de implementação (JIN et al., 2018). Como exemplo, um sistema de biblioteca pode

[1] https://www.facebook.com/

empregar a API do Google Books[2] para consultar informações sobre um livro específico, como editora, edição e idioma, sem a necessidade de manter esse conjunto de dados localmente.

Cabe destacar que, embora frequentemente associadas a aplicações web, as APIs são amplamente utilizadas em diferentes contextos computacionais, incluindo aplicativos móveis, sistemas em nuvem e arquiteturas distribuídas. Apesar dessa diversidade de plataformas, o modelo básico de comunicação permanece fundamentado na arquitetura cliente-servidor, composta por dois elementos centrais: o cliente e o servidor. O cliente da API é responsável por iniciar a comunicação, enviando requisições que representam ações ou consultas desejadas. Em cenários práticos, esse cliente pode assumir a forma de um aplicativo web, um aplicativo móvel, um serviço em nuvem ou até mesmo uma ferramenta de testes automatizados (BIEHL, 2015).

Nesse contexto, existem ferramentas amplamente utilizadas que atuam como clientes de APIs, dentre as quais se destacam o *Postman*[3], o *RapidAPI*[4] e o *Insomnia*[5]. Após a implementação do servidor da API, tais ferramentas podem ser utilizadas para enviar requisições, analisar respostas e validar o comportamento da aplicação. O *Postman*, por exemplo, começou como uma ferramenta simples para o envio de requisições HTTP e evoluiu para uma plataforma que abrange diversas etapas do ciclo de vida do desenvolvimento de APIs, incluindo testes, documentação e monitoramento. O *Insomnia*, por sua vez, caracteriza-se como um cliente de código aberto multiplataforma, oferecendo funcionalidades como geração de código, criação de *mocks* e integração com sistemas de controle de versão (INSOMNIA, 2025). Já o *RapidAPI*, além de permitir o envio de requisições, atua principalmente como um *marketplace* de APIs, centralizando milhares de serviços públicos e privados em um único ambiente (BILAL, 2023).

Considerando a dinâmica de funcionamento de um cliente de API, quando um usuário realiza uma ação em uma aplicação, como uma pesquisa em um site, o cliente envia uma requisição à API para obter os dados correspondentes. O cliente também pode desempenhar outros papéis, como o de ferramenta de testes, validando se as operações estão sendo executadas corretamente, ou ainda o de um software externo que consome dados para fins próprios. No exemplo anteriormente citado da API Google Books, o sistema da biblioteca atua como cliente, uma vez que é ele quem solicita as informações sobre os livros à API.

Além do cliente, o servidor da API é o outro componente essencial da comunicação. O servidor é responsável por processar as requisições recebidas, validar os dados enviados, executar operações no banco de dados, aplicar mecanismos de autenticação e autorização, registrar eventos para fins de monitoramento e depuração e gerar as respostas apropriadas. Em termos de fluxo operacional, após o recebimento de uma requisição, o servidor verifica sua validade e as permissões associadas; em

[2]https://developers.google.com/books
[3]https://www.postman.com/
[4]https://rapidapi.com/
[5]https://insomnia.rest/

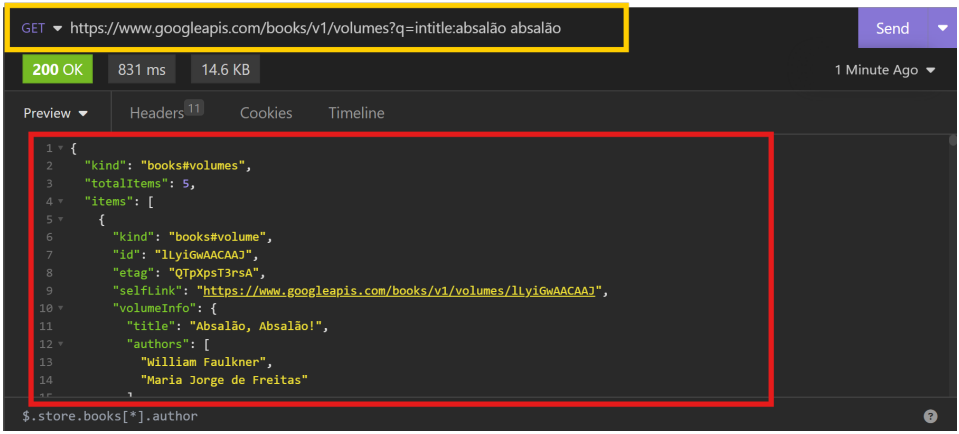


Figura 7.1: Requisição para a API Google Books

seguida, processa a solicitação, consulta as informações necessárias e retorna uma resposta ao cliente (OLIVEIRA, 2024). No caso da API Google Books, esse papel é desempenhado pela infraestrutura do Google, que hospeda e gerencia os dados das obras cadastradas.

A Figura 7.1 apresenta um exemplo de requisição realizada por um cliente Insomnia à API Google Books. Nessa requisição, solicitam-se livros que contenham no título o termo *Absalão Absalão*. Para a execução da consulta, o cliente informa elementos fundamentais da requisição, como a **URL** da API e o parâmetro **intitle** com seu respectivo valor. Esses componentes estão destacados em amarelo na figura. Após o processamento da solicitação, o servidor retorna uma resposta com as informações dos livros correspondentes, apresentada no formato JSON e destacada em vermelho na mesma figura.

É importante destacar que certas APIs podem exigir abordagens específicas, que dependem da classificação à qual a API se enquadra. Existem dois tipos de classificação, as baseadas no processo de comunicação adotado e as baseadas no seu modelo de implementação. Em relação ao processo de comunicação, existem APIs que operam em redes restritas ou globais; ou seja, os desenvolvedores podem construir APIs de acesso público, restrito ou privado. No que tange as aplicações públicas, algumas empresas de tecnologia, como *Google*, *Facebook*, *X* e *Amazon*, fornecem APIs para usuários acessarem e integrarem em seus projetos. Ademais, em relação à classificação do modelo que as APIs podem adotar, existem duas alternativas: baseadas em requisição-resposta ou em eventos. Dessa forma, apesar de a lógica de utilização permanecer inalterada para todas as APIs, pois o fluxo é sempre enviar uma requisição e receber uma resposta, algumas aplicações exigirão abordagens diferentes em determinadas funcionalidades, como, por exemplo, a adição

de um token de autenticação, elemento que será abordado em seções subsequentes do trabalho.

7.1.1 Arquitetura Cliente-Servidor

Os conceitos de cliente e servidor, apresentados anteriormente, constituem os dois elementos fundamentais da arquitetura cliente-servidor. Nessa arquitetura, o cliente é responsável por iniciar a comunicação, enviando mensagens, também denominadas requisições, ao servidor para solicitar dados ou serviços específicos. O servidor, por sua vez, recebe essas requisições, avalia se atendem aos critérios estabelecidos e decide se devem ser processadas. Quando a requisição é aceita, o servidor retorna ao cliente os recursos solicitados; caso contrário, uma mensagem de erro é enviada, indicando o motivo da falha. Esse modelo de comunicação viabiliza sistemas compostos por múltiplos clientes, distribuídos em diferentes máquinas, que interagem de forma centralizada com um único servidor (KUMAR, 2019).

Em cenários reais, os clientes podem estar presentes em diferentes tipos de dispositivos, como computadores pessoais, smartphones ou aplicações em nuvem, todos capazes de enviar requisições simultâneas ao servidor. Diante dessa multiplicidade de clientes, a arquitetura cliente-servidor precisa ser capaz de gerenciar um grande volume de solicitações concorrentes. Para isso, são empregados protocolos e mecanismos específicos que organizam o fluxo de comunicação, garantindo o envio correto das requisições e o retorno das respostas de forma eficiente e confiável (RAPID, 2021). Esses aspectos relacionados aos mecanismos de comunicação e aos protocolos envolvidos serão aprofundados ao longo do capítulo.

O servidor, enquanto elemento central dessa arquitetura, pode ser entendido como um computador ou um software responsável por disponibilizar serviços a outros sistemas. Entre suas principais atribuições estão a realização de consultas em bancos de dados, o processamento das requisições recebidas e o envio de diferentes tipos de recursos aos clientes, como documentos, imagens, vídeos e demais mídias (KUMAR, 2019). Conforme esse modelo, a arquitetura cliente-servidor caracteriza-se por um relacionamento do tipo **um para muitos**, no qual um único servidor pode atender, de forma simultânea e independente, a diversos clientes. Cada cliente interage exclusivamente com o servidor, sem ter conhecimento direto das requisições realizadas por outros clientes.

A Figura 7.2 ilustra esse modelo de funcionamento, apresentando um servidor central que atende a múltiplos clientes. Embora todos os clientes utilizem a mesma fonte de dados, suas interações ocorrem de forma isolada, tornando cada comunicação independente. Essa centralização permite que o servidor concentre aspectos críticos da aplicação, como autenticação, autorização e controle de acesso aos dados. Além disso, ao unificar o processamento das informações, o servidor assegura que todas as alterações realizadas no banco de dados sejam refletidas de forma consistente aos clientes. Ainda que a figura represente apenas três clientes conectados simultaneamente, esse número pode variar conforme a demanda do

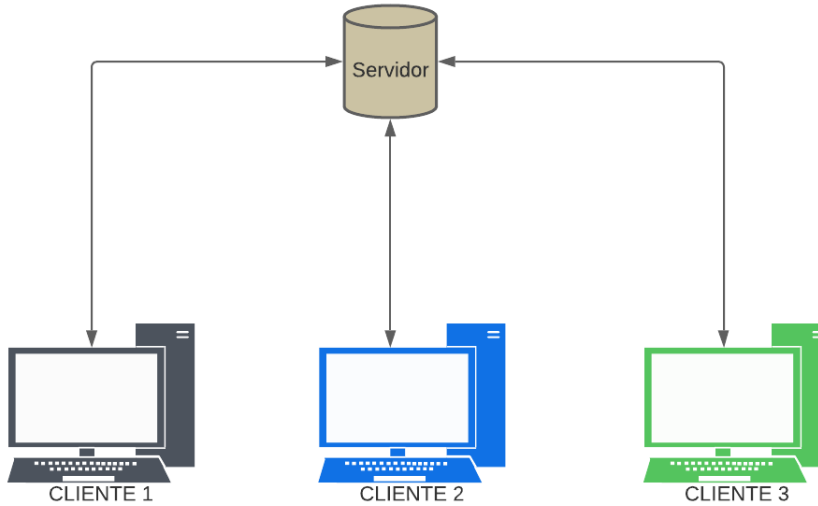


Figura 7.2: Arquitetura Cliente-Servidor

sistema, o que evidencia que a arquitetura cliente-servidor é inerentemente flexível e escalável.

7.1.2 Estrutura das Requisições de APIs

A comunicação entre clientes e servidores em APIs ocorre por meio de requisições HTTP, que são utilizadas para solicitar serviços ou acessar recursos disponibilizados pelo servidor. Esses recursos podem representar diferentes tipos de dados, como documentos, imagens, vídeos ou páginas HTML. As operações realizadas sobre tais recursos envolvem, por exemplo, procedimentos de leitura, criação, atualização ou exclusão de informações. A identificação dos recursos na web é realizada por meio de URL, as quais são utilizadas em conjunto com métodos HTTP para indicar ao servidor qual ação deve ser executada sobre determinado recurso (MOZDEVNET, 2023c).

A Figura 7.3 ilustra os principais componentes de uma requisição de leitura, composta por um endpoint, um método HTTP do tipo *GET* e um parâmetro de consulta. Esses elementos formam a base estrutural das requisições de APIs web e são detalhadamente discutidos a seguir.

Endpoints: Os endpoints são as URLs que funcionam como pontos de acesso entre o cliente e o servidor. Em conjunto com os métodos HTTP, definem a estrutura básica das requisições. Ao enviar uma requisição a um endpoint específico, o cliente indica ao servidor qual recurso deseja acessar e qual operação deve ser realizada.

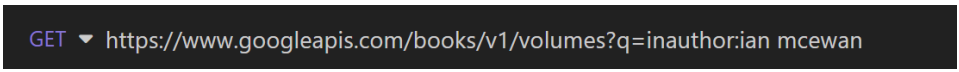


Figura 7.3: Componentes de uma requisição

Tabela 7.1: Verbos HTTP

Verbo	Função
GET	Retorna a representação de um recurso ou de uma lista de recursos
POST	Insere um novo recurso no servidor
PUT	Atualiza as informações de um recurso
PATCH	Realiza alterações parciais em um recurso
DELETE	Remove um recurso do servidor
HEAD	Retorna os metadados de um recurso
CONNECT	Estabelece um túnel para um servidor específico
OPTIONS	Informa as opções de requisição disponíveis para um recurso
TRACE	Realiza testes de loopback

Dessa forma, os endpoints viabilizam a comunicação entre as aplicações, permitindo a transferência controlada de dados entre cliente e servidor (POSTMAN, 2023b).

Métodos HTTP: Além da definição do endpoint, é necessário especificar a ação a ser executada sobre o recurso, o que é feito por meio dos métodos HTTP, também conhecidos como verbos HTTP. Esses métodos são componentes obrigatórios das requisições em APIs web (MOZDEVNET, 2023d). Entre os verbos mais utilizados destacam-se *GET*, *POST*, *PUT*, *PATCH* e *DELETE*, enquanto *HEAD*, *CONNECT*, *OPTIONS* e *TRACE* são empregados com menor frequência. A Tabela 7.1 apresenta os principais métodos HTTP e suas respectivas funções.

A Tabela 7.2 apresenta exemplos de endpoints da API Google Books associados ao recurso *Mylibrary.bookshelves*. Observa-se que diferentes operações utilizam métodos HTTP distintos, conforme a ação desejada. Por exemplo, a operação **addVolume** utiliza o verbo *POST*, por se tratar de uma ação de inserção de dados, enquanto a operação **list** emprega o verbo *GET*, voltado à recuperação de informações. As URLs indicam, por meio do caminho **/mylibrary/bookshelves**, qual coleção de dados será manipulada pelo servidor.

Corpo da requisição: O corpo da requisição, também denominado *body*, contém os dados enviados ao servidor em operações que envolvem a manipulação de informações, como a criação, atualização ou exclusão de recursos. Esse componente é utilizado principalmente em requisições dos tipos *POST*, *PUT* e *PATCH*. Os dados

Tabela 7.2: Tabela de endpoints extraída da documentação da API Google Books

Método	URI REST	Descrição
addVolume	POST /mylibrary/bookshelves/shelf/addVolume	Adiciona um volume à estante
clearVolumes	POST /mylibrary/bookshelves/shelf/clearVolumes	Remove todos os volumes da estante
get	GET /mylibrary/bookshelves/shelf	Recupera metadados de uma estante
list	GET /mylibrary/bookshelves	Recupera a lista de estantes do usuário
moveVolume	POST /mylibrary/bookshelves/shelf/moveVolume	Move um volume na estante
removeVolume	POST /mylibrary/bookshelves/shelf/removeVolume	Remove um volume da estante

podem ser estruturados em diferentes formatos, como **JSON**, **XML** ou **multipart forms** (ANALYTICS, 2024). Por exemplo, ao cadastrar um novo livro em um sistema de biblioteca, o cliente envia uma requisição *POST* contendo, no corpo da mensagem, informações como título, autor e ano de publicação. A Figura 7.4 ilustra esse cenário.

Parâmetros: Os parâmetros são informações adicionais que orientam o processamento da requisição pelo servidor. Eles podem ser incluídos na URL ou no corpo da requisição, dependendo da estratégia adotada pelo desenvolvedor. No exemplo apresentado na Figura 7.5, o parâmetro *id* é enviado na URL para indicar qual livro deve ser retornado pelo servidor.

Metadados das requisições: As informações relacionadas aos metadados das requisições são transmitidas por meio dos cabeçalhos HTTP. Esses cabeçalhos contêm dados complementares, como o tipo de conteúdo enviado, informações de autenticação, tokens de acesso e cookies associados à requisição (MOZDEVNET, 2023a). O cabeçalho de autorização, por exemplo, indica as permissões do cliente em relação aos recursos protegidos. Quando esse cabeçalho não é fornecido em uma requisição que exige autenticação, o servidor pode responder com o status *401 Unauthorized*. A Figura 7.6 apresenta exemplos de cabeçalhos comumente utilizados, como **User-Agent** e **Content-Type**.



Figura 7.4: Exemplo do corpo de uma requisição POST



Figura 7.5: Exemplo de requisição com envio de parâmetro

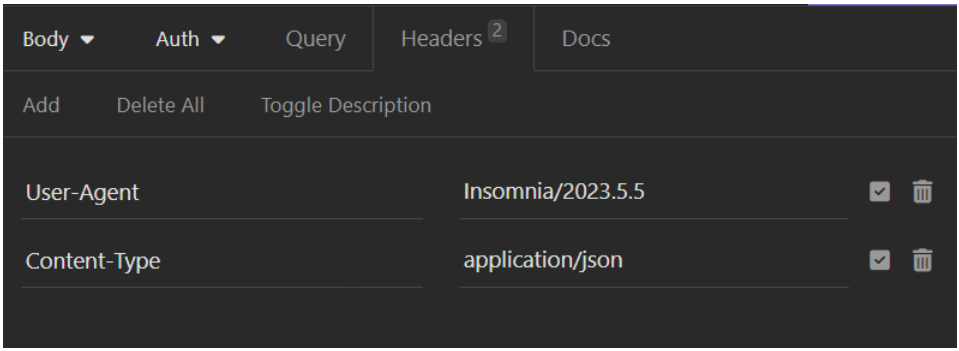


Figura 7.6: Cabeçalhos de uma requisição

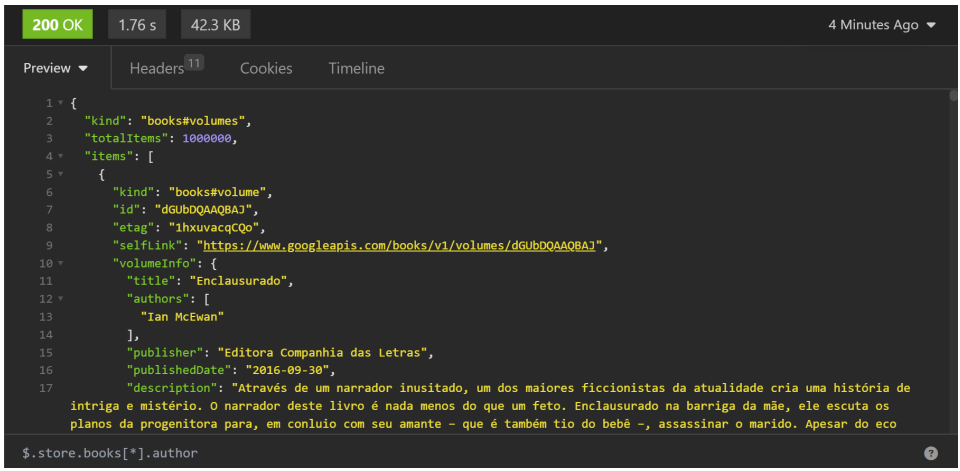


Figura 7.7: Resposta enviada após uma requisição de leitura

7.1.3 Estrutura das Respostas de APIs

Após o processamento de uma requisição, o servidor envia uma resposta ao cliente. Assim como as requisições, as respostas de APIs também apresentam uma estrutura bem definida, composta principalmente por três elementos: **código de status**, **cabeçalhos** e **corpo da resposta**. Esses componentes permitem ao cliente interpretar corretamente o resultado da operação solicitada, bem como compreender informações adicionais sobre o processamento da requisição.

Para fins ilustrativos, a Figura 7.7 apresenta a resposta retornada após a execução de uma requisição de leitura na API Google Books. Nessa resposta, observa-se o retorno das informações de um único livro, como título e autor, estruturadas no formato JSON. O conteúdo exibido no arquivo corresponde ao corpo da resposta. Além disso, são apresentados outros valores relevantes, como o código de status *200 OK* e metadados operacionais, incluindo o tempo de resposta (*1.76s*) e o tamanho da mensagem (*42.3KB*). Em conjunto, esses elementos constituem a estrutura básica de uma resposta de API.

Corpo da resposta: O corpo da resposta é o principal componente retornado pelo servidor, pois contém os dados efetivamente solicitados pelo cliente. De maneira análoga ao corpo das requisições, esse elemento pode ser representado em diferentes formatos, como **JSON**, **XML** ou **texto simples** (SENDBIRD, 2025). Retomando o exemplo da Figura 7.7, o corpo da resposta é composto por informações sobre o livro consultado, organizadas em um objeto JSON retornado pelo servidor.

Códigos de status HTTP: As respostas enviadas pelo servidor incluem códigos de status HTTP que indicam o resultado da requisição, informando se a operação foi realizada com sucesso ou se ocorreu algum tipo de erro. Por exemplo, caso um cliente realize uma requisição do tipo *PUT* para atualizar um recurso

Tabela 7.3: Códigos de status HTTP

Status	Significado
200 OK	A requisição foi processada com sucesso
201 CREATED	Um novo recurso foi criado com sucesso
202 ACCEPTED	A requisição foi aceita, mas ainda não processada
302 FOUND	O recurso foi temporariamente movido para outra URL
400 BAD REQUEST	A requisição contém erro por parte do cliente
401 UNAUTHORIZED	O acesso ao recurso requer autenticação
403 FORBIDDEN	O cliente não possui permissão para acessar o recurso
404 NOT FOUND	O recurso solicitado não foi encontrado

inexistente, o servidor poderá retornar o código **404 NOT FOUND**, indicando que o recurso solicitado não foi encontrado. A Tabela 7.3 apresenta os códigos de status mais comuns em APIs web. Observa-se que o código *200 OK*, apresentado na Figura 7.7, indica que a requisição de leitura foi processada com êxito.

Cabeçalhos da resposta: Assim como ocorre nas requisições, as respostas também incluem cabeçalhos HTTP que fornecem informações adicionais sobre a mensagem retornada pelo servidor. O cabeçalho *Content-Type*, por exemplo, indica o formato dos dados no corpo da resposta. No caso ilustrado na Figura 7.8, o cabeçalho *Content-Type: application/json* indica que os dados retornados estão estruturados em JSON.

Outro cabeçalho relevante é o *Cache-Control*, responsável por definir diretrizes para o armazenamento em cache da resposta, como o tempo de expiração e as regras de revalidação. No exemplo apresentado, o valor *private* indica que a resposta é destinada a um único usuário e não deve ser armazenada em caches compartilhados (MOZDEVNET, 2023b). Adicionalmente, o cabeçalho *Server* informa o nome ou a versão do software utilizado pelo servidor para gerar a resposta. Geralmente, essas informações são apresentadas de forma limitada, com o objetivo de reduzir riscos de segurança. Conforme observado na Figura 7.8, o valor informado é *EFS*, um software interno utilizado pela infraestrutura do Google.

7.1.4 APIs orientadas a requisição-resposta

As APIs podem ser classificadas, de forma geral, em dois grandes grupos: APIs orientadas a **requisição–resposta** e APIs orientadas a **eventos**. No primeiro caso, a comunicação ocorre de maneira síncrona, em que o cliente envia uma requisição ao servidor e aguarda explicitamente uma resposta. Esse modelo baseia-

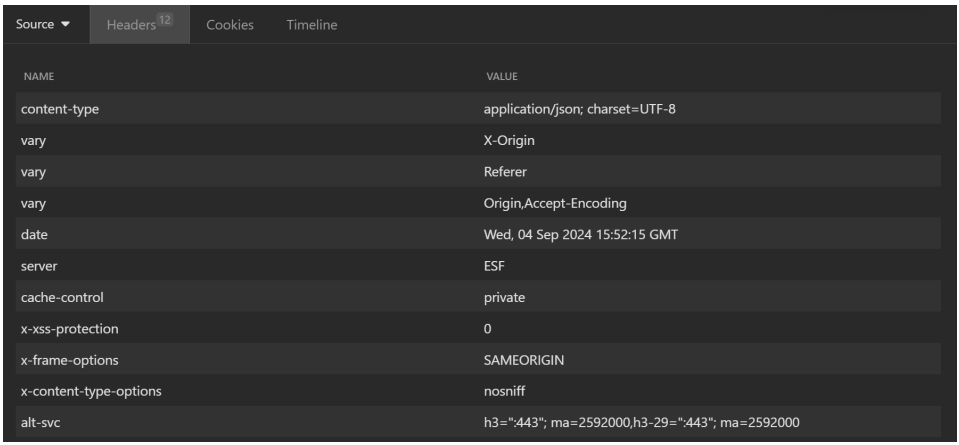


Figura 7.8: Cabeçalhos de uma resposta

se, predominantemente, no protocolo HTTP e utiliza servidores web para expor os serviços da aplicação. Assim, os clientes invocam requisições HTTP por meio de endpoints disponibilizados pela API e, em seguida, os servidores processam essas solicitações e retornam respostas, que podem ser representadas em diferentes formatos de dados (JIN et al., 2018).

As APIs orientadas a requisição-resposta podem adotar distintos paradigmas arquiteturais, dentre os quais se destacam *REST*, *RPC* e *GraphQL*. Ao longo da evolução das APIs web, o paradigma que obteve maior adoção no mercado foi o REST (SHEPARD, 2023), sendo amplamente utilizado por grandes empresas de tecnologia, como Google[6], X[7] e Amazon[8].

O paradigma REST foi proposto em 2000 por Roy Fielding, em sua tese de doutorado, na qual foram definidas seis restrições arquiteturais fundamentais para a organização da web (FIELDING, 2000). Posteriormente, esses princípios passaram a ser aplicados no desenvolvimento de APIs denominadas **RESTful**, que seguem rigorosamente o conjunto de regras estabelecido pelo modelo REST. Entre essas restrições destacam-se: a **arquitetura cliente–servidor**, a **interface uniforme**, o princípio de **stateless**, o uso de **cache**, a organização em **sistemas em camadas** e o conceito de **código por demanda** (MASSE, 2011).

A arquitetura cliente-servidor constitui uma das bases da comunicação em sistemas REST. Esse modelo favorece aplicações web ao promover a separação de responsabilidades entre cliente e servidor, permitindo que ambos evoluam de forma independente. Como resultado, os sistemas REST alcançam benefícios como maior

[6]https://developers.google.com/workspace/drive/api/reference/rest/v2
[7]https://docs.x.com/x-api/introduction
[8]https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html

portabilidade entre plataformas, melhor escalabilidade e facilidade de manutenção, aspectos essenciais para aplicações distribuídas em larga escala (FIELDING, 2000).

No que se refere à cacheabilidade, os servidores REST indicam, por meio de diretivas específicas, se as respostas podem ou não ser armazenadas em cache (MASSE, 2011). Por exemplo, ao responder a uma requisição, o servidor pode classificar os dados retornados como **armazenáveis em cache** ou **não armazenáveis**. No primeiro caso, o cliente pode reutilizar essas informações em requisições subsequentes equivalentes, reduzindo a necessidade de novas chamadas ao servidor. No segundo, a reutilização é explicitamente proibida. A correta aplicação das regras de cache contribui para aumentar a confiabilidade e a disponibilidade da API, além de reduzir os custos de desempenho.

De modo geral, as APIs RESTful são organizadas em **sistemas em camadas**, o que implica uma arquitetura hierárquica composta por diferentes níveis de processamento (FIELDING, 2000). Essa abordagem permite a inserção de componentes intermediários, como `proxies` e `gateways`, responsáveis por funções específicas, como o balanceamento de carga ou o controle de acesso. Embora essa organização traga benefícios estruturais, ela também pode introduzir sobrecarga adicional, que pode ser mitigada por estratégias adequadas, como o uso eficiente de cache.

Outro princípio central do REST é o conceito de **stateless**. Esse princípio estabelece que cada requisição enviada ao servidor deve conter todas as informações necessárias para que seja compreendida e processada, sem depender de estados armazenados de interações anteriores. A adoção desse princípio atende a quatro objetivos principais: eliminar a necessidade de manter o estado da aplicação entre requisições; permitir o processamento paralelo de requisições; possibilitar que intermediários compreendam requisições de forma isolada; e garantir que todas as informações relevantes para a cacheabilidade estejam presentes em cada mensagem (FIELDING, 2000).

O princípio de **código por demanda**, por sua vez, permite que o servidor envie código executável ao cliente, que pode ser baixado e executado dinamicamente. Essa abordagem reduz o número de funcionalidades que precisam ser implementadas previamente no cliente e pode melhorar a escalabilidade do sistema. Contudo, o código por demanda reduz a visibilidade da API e, por esse motivo, trata-se de uma restrição opcional no modelo REST, cabendo ao desenvolvedor decidir sobre sua adoção (FIELDING, 2000).

Além dessas restrições, Fielding (2000) destaca quatro princípios relacionados à manipulação de recursos. O primeiro refere-se à definição de **recursos**, que correspondem a quaisquer elementos acessíveis na web por meio de identificadores únicos. O segundo princípio estabelece que a manipulação desses recursos é responsabilidade do cliente, sendo possível que múltiplos clientes acessem e manipulem o mesmo recurso em diferentes representações. Por exemplo, um cliente pode requisitar um recurso e recebê-lo em HTML, enquanto outro obtém o mesmo recurso em JSON.

O terceiro princípio está relacionado às **mensagens autodescritivas**, segundo o qual as mensagens trocadas entre cliente e servidor devem conter informações

suficientes para descrever o estado atual do recurso. Assim, ao solicitar a atualização de um atributo, o cliente envia uma mensagem que representa o novo estado desejado, cabendo ao servidor aceitar ou rejeitar a alteração e retornar uma resposta que reflita o estado atualizado do recurso.

Por fim, o quarto princípio refere-se ao uso de **hypermedia**, que permite que os recursos contenham links para outros recursos relacionados, criando um mecanismo de navegação na própria API. Por exemplo, um recurso *Livro* pode conter um link para o recurso *Autor*, que, por sua vez, pode apontar para outros recursos associados. Essa abordagem torna a interação com a API mais dinâmica, permitindo que o cliente descubra e acesse funcionalidades por meio dos links fornecidos nas respostas.

7.1.5 APIs orientadas a eventos

Em cenários em que os dados sofrem alterações frequentes e tornam-se rapidamente obsoletos, o modelo tradicional de comunicação baseado em requisição-resposta pode não ser o mais adequado. Essa limitação decorre do elevado número de requisições necessárias para verificar se houve atualização de informações, o que pode resultar em desperdício de recursos computacionais, especialmente quando as chamadas não retornam novos dados. Diante desse contexto, o modelo orientado a eventos surge como uma alternativa mais eficiente, pois permite que o servidor notifique os clientes apenas quando um evento relevante ocorre (JIN et al., 2018).

As APIs orientadas a eventos adotam mecanismos que possibilitam a comunicação assíncrona e a entrega de dados em tempo real ou quase em tempo real. Entre os principais mecanismos utilizados nesse modelo, destacam-se os **WebHooks**, os **WebSockets** e o **HTTP Streaming**, cada um com características específicas e adequado a diferentes tipos de aplicação.

O primeiro mecanismo a ser considerado é o **WebHook**. Trata-se de uma URL configurada para receber chamadas HTTP, geralmente pelo método *POST*. Nesse modelo, sempre que um evento ocorre na aplicação, o servidor envia automaticamente uma requisição ao WebHook previamente registrado pelo cliente. Dessa forma, o cliente passa a receber notificações em tempo real, sem a necessidade de realizar requisições periódicas ao servidor. Contudo, para usufruir plenamente dessa abordagem, a API precisa lidar com desafios adicionais, como o tratamento de falhas de entrega, retentativas e validação das respostas retornadas pelo WebHook. Entre as plataformas que utilizam amplamente esse mecanismo estão *Stripe*, *GitHub* e *Zapier*.

Outro mecanismo amplamente utilizado em APIs orientadas a eventos é o **WebSockets**. Esse protocolo estabelece um canal de comunicação bidirecional persistente sobre uma conexão TCP, permitindo a troca contínua de mensagens entre cliente e servidor, ou mesmo entre servidores. Devido ao suporte nativo na maioria dos navegadores modernos, os WebSockets tornaram-se uma solução popular para aplicações em tempo real. Por exemplo, o *Slack*[9] utiliza WebSockets

[9]https://slack.com/

para transmitir eventos, como a criação de canais e o envio de mensagens, de forma imediata. De modo semelhante, o *Trello*[10] emprega esse protocolo para sincronizar, em tempo real, as alterações realizadas pelos usuários entre servidores e navegadores. Apesar das vantagens, o uso de WebSockets requer uma conexão contínua com a internet, o que pode representar uma limitação para determinados tipos de aplicação (JIN et al., 2018).

Por fim, destaca-se o **HTTP Streaming**, uma técnica que permite ao servidor manter uma conexão aberta com o cliente e enviar dados de forma incremental ao longo do tempo. Nesse modelo, o servidor pode prolongar a duração da resposta e transmitir novas informações à medida que elas se tornam disponíveis. Existem duas abordagens principais para a implementação do HTTP Streaming. Na primeira, o servidor utiliza o cabeçalho `Transfer-Encoding` com o valor *chunked*, indicando que os dados serão enviados em fragmentos sucessivos. Na segunda abordagem, a comunicação é realizada por meio de *Server-sent events* (SSE), que permite o envio unidirecional de eventos do servidor para o cliente. Como exemplo prático, a rede social X utiliza técnicas de streaming para manter conexões abertas e receber atualizações contínuas de dados (JIN et al., 2018).

7.1.6 Clientes e Servidores de APIs

Apesar da ampla variedade de clientes de API disponíveis no mercado, é comum que desenvolvedores implementem seus próprios clientes em linguagens de programação, especialmente no contexto de APIs web. Diversas linguagens oferecem bibliotecas específicas para facilitar a realização de requisições HTTP. A linguagem Python, por exemplo, disponibiliza a biblioteca **requests**, amplamente utilizada para enviar requisições a uma URL de forma simples e direta.

A Listagem 7.1 ilustra o uso da biblioteca *requests* para implementar um cliente de API em Python. Inicialmente, são definidas duas variáveis: *autor*, que armazena o valor de um parâmetro da requisição, e *url*, que contém o endpoint da API. Observa-se que o parâmetro é incorporado à URL por meio de interpolação de strings. Em seguida, a requisição é executada por meio da função *request*, à qual são passados o método HTTP *GET* e o endpoint definido na variável *url*. Como resultado, o código realiza uma solicitação de leitura a partir da API Google Books e imprime o conteúdo retornado na resposta.

[10] https://trello.com/

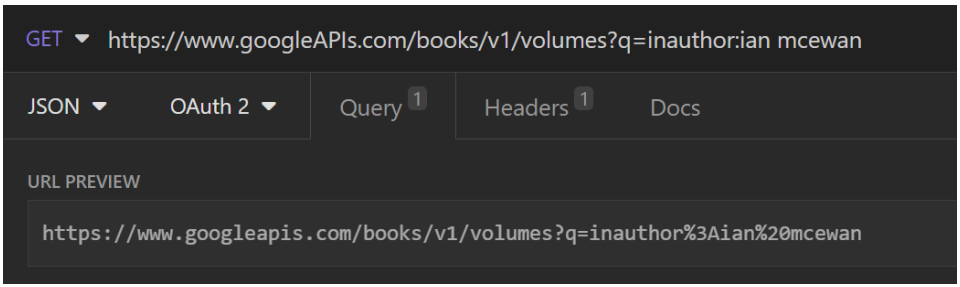


Figura 7.9: Requisição realizada na API Google Books utilizando o Insomnia

```python
import requests

autor = "Ian McEwan"
url = f"https://www.googleapis.com/books/v1/volumes?q=inauthor:{autor}"

response = requests.request("GET", url)

print(response.text)
```

Listagem 7.1: Cliente de uma API desenvolvido na linguagem Python

Além da implementação de clientes em código, o uso de softwares especializados que atuam como clientes de API pode facilitar significativamente as etapas de teste e depuração das aplicações. Essas ferramentas dispensam o conhecimento prévio de linguagens de programação ou de frameworks específicos, permitindo que o desenvolvedor realize requisições apenas informando o endpoint, o método HTTP e os parâmetros necessários (POSTMAN, 2024).

A Figura 7.9 apresenta a mesma requisição ilustrada na Listagem 7.1, porém executada com o software *Insomnia*. Nesse cenário, não é necessário escrever código para efetuar a requisição, sendo suficiente configurar visualmente os componentes da chamada, como a URL e os parâmetros.

O servidor de uma API web, por sua vez, é o componente responsável por receber as requisições dos clientes, verificar as permissões de acesso, validar os dados recebidos e processar as operações solicitadas. Por atuar de forma integrada ao banco de dados, o servidor pode ser compreendido como um agente intermediário entre a camada de persistência e a interface exposta pela API (POSTMAN, 2023a).

As respostas geradas pelo servidor são, em geral, respostas HTTP, frequentemente estruturadas nos formatos **JSON** ou **XML**. Além dos dados solicitados, essas respostas incluem mensagens de confirmação ou de erro, bem como códigos de status que indicam se a requisição foi processada com sucesso ou se ocorreu alguma falha durante a execução (LI et al., 2013). Em seções subsequentes, serão apresentados exemplos práticos de implementação de servidores de APIs, explorando diferentes padrões arquiteturais.

7.2 Comparação entre APIs RESTful, GraphQL e gRPC

Esta seção tem como objetivo apresentar e discutir, de forma estruturada e comparativa, três das principais abordagens para o desenvolvimento de APIs web amplamente utilizadas em sistemas contemporâneos: as APIs RESTful, as APIs GraphQL e as APIs gRPC. Cada uma dessas tecnologias adota princípios arquiteturais distintos, propõe diferentes modelos de comunicação cliente-servidor e atende a necessidades específicas no contexto de aplicações distribuídas, serviços web e sistemas orientados a dados.

Inicialmente, são abordadas as APIs RESTful, que se consolidaram como padrão dominante no desenvolvimento de serviços web em razão de sua simplicidade, ampla adoção e forte alinhamento com o protocolo HTTP. Em seguida, são apresentadas as APIs GraphQL, que surgem como uma alternativa ao modelo tradicional REST ao oferecer maior flexibilidade na recuperação de dados, permitindo que o cliente especifique exatamente quais informações deseja obter em cada requisição. Por fim, são discutidas as APIs gRPC, que se baseiam no paradigma de chamadas de procedimento remoto (RPC) e se destacam pelo alto desempenho, comunicação eficiente e forte tipagem, sendo particularmente adequadas para ambientes distribuídos e sistemas de alta escalabilidade.

Além da apresentação conceitual, esta seção apresenta exemplos práticos de implementação das três abordagens, com o intuito de evidenciar suas diferenças estruturais, operacionais e tecnológicas. Para isso, foram desenvolvidas APIs de referência utilizando frameworks representativos de cada modelo, possibilitando a análise de aspectos como a definição de recursos, a estrutura de requisições e respostas, as operações de criação, leitura, atualização e exclusão de dados, bem como a interação entre clientes e servidores.

Dessa forma, a seção estabelece a base teórica e prática necessária ao estudo comparativo a ser apresentado posteriormente, permitindo compreender as potencialidades, limitações e cenários de aplicação mais adequados para cada tipo de API. Essa abordagem contribui para uma análise mais fundamentada da escolha da tecnologia de API em projetos de software modernos, considerando critérios como desempenho, flexibilidade, complexidade de implementação e de manutenção.

7.2.1 APIs RESTful

As aplicações *RESTful* caracterizam-se principalmente pela modelagem das entidades como recursos, pela utilização do protocolo **HTTP/1.1** e pela adoção dos princípios do estilo arquitetural REST. Nesse contexto, o elemento central das APIs baseadas nesse paradigma são os recursos, que correspondem a entidades identificáveis por meio de um identificador único, um nome e um endereço. Esses recursos são empregados para representar e expor conjuntos de dados, que podem

Tabela 7.4: Recursos e endpoints da API REST

Recurso	Endpoint
Bolsas	⟨http://localhost:8000/bolsas⟩
Cursos	⟨http://localhost:8000/cursos⟩
Empresas	⟨http://localhost:8000/empresas⟩
Endereços	⟨http://localhost:8000/enderecos⟩
Estágios	⟨http://localhost:8000/estagios⟩
Plataformas	⟨http://localhost:8000/plataformas⟩
Professores	⟨http://localhost:8000/professores⟩

ser acessados por outras aplicações por meio de *endpoints* específicos (JIN et al., 2018).

Cada recurso deve ser nomeado com substantivos que possuam significado claro no domínio de aplicação, de modo a garantir que as interações entre cliente e servidor ocorram de forma objetiva e sem ambiguidades. Por exemplo, um desenvolvedor pode definir um recurso denominado *Cursos* para representar os cursos disponíveis em uma plataforma digital. Assim, sempre que o cliente necessitar de informações relacionadas aos cursos, deverá realizar uma requisição ao endpoint correspondente a esse recurso. A Tabela 7.4 apresenta os recursos e seus respectivos endpoints disponíveis na API REST desenvolvida neste capítulo.

Além dos endpoints, as requisições em APIs REST utilizam métodos HTTP. A estrutura básica de uma requisição resulta da combinação entre o endpoint, responsável por identificar o recurso, e o método HTTP, que indica o tipo de operação que o cliente deseja executar. No que se refere às operações possíveis em uma API, não restritas apenas ao modelo REST, o conjunto fundamental é conhecido como *CRUD*, acrônimo para *Create*, *Read*, *Update* e *Delete*. Dessa forma, as requisições básicas de uma API estão associadas às operações de criação, leitura, atualização e exclusão de recursos.

Por exemplo, quando o cliente deseja cadastrar uma nova oferta de estágio, deve-se realizar uma requisição com o método HTTP **POST**, direcionada ao endpoint correspondente ao recurso **Estágios**. Ressalta-se que, além dos elementos obrigatórios, as requisições em APIs REST podem incluir parâmetros, corpo da requisição e cabeçalhos, conforme discutido anteriormente nesta seção.

A Figura 7.10 ilustra a estrutura de uma requisição realizada pelo cliente *Insomnia* a uma API REST local. Observa-se o uso do método **POST**, que indica uma operação de criação, em conjunto com o endpoint do recurso **Estágios**, o que evidencia que o cliente está adicionando uma nova oferta de estágio à aplicação. Além do método e do endpoint, a requisição contém um corpo em formato JSON, elemento essencial nesse tipo de operação, pois é por meio dele que os dados são enviados ao servidor para persistência no banco de dados.

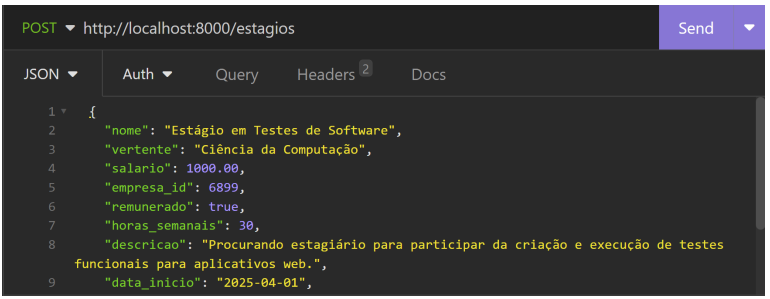


Figura 7.10: Requisição realizada em uma API REST usando o Insomnia

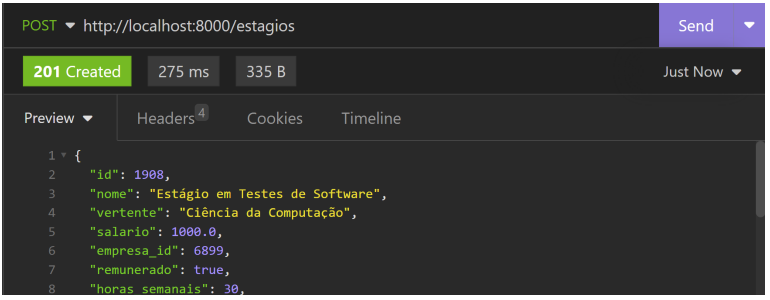


Figura 7.11: Resposta retornada por uma API REST

No que se refere à estrutura das respostas em APIs REST, a Figura 7.11 apresenta o retorno do servidor para a requisição descrita anteriormente. Destaca-se o código HTTP, responsável por indicar o sucesso ou a falha da operação. No exemplo apresentado, o código **201 CREATED** indica que a operação foi concluída com êxito e que o recurso foi criado com sucesso. Outros códigos de status comumente utilizados em respostas HTTP foram apresentados anteriormente na Tabela 7.3.

Além do código de status, o servidor retornou um corpo de resposta no formato JSON, contendo os dados cadastrados no banco de dados, incluindo o identificador do recurso. Cabe destacar que a decisão de retornar um corpo de resposta após uma operação de criação é uma escolha de projeto, e não uma exigência normativa do estilo REST. Adicionalmente, foram enviados metadados relacionados à requisição, como o tempo de processamento, neste caso **275 ms**, e o tamanho da resposta, correspondente a **335 B** (335 bytes).

7.2.2 Operações CRUD

Diversas linguagens de programação oferecem frameworks que facilitam o desenvolvimento de APIs REST. No ecossistema Python, destacam-se o **FastAPI**, o **Flask** e o **Django REST Framework**. Após uma análise fundamentada em

estudos recentes (WU, 2024) e em conteúdos técnicos especializados (BAAREN, 2024), considerando critérios como desempenho, validação automática de dados, tipagem explícita e suporte a operações assíncronas, optou-se pelo uso do **FastAPI** no desenvolvimento da API REST apresentada neste capítulo.

A Listagem 7.2 apresenta a inicialização de uma aplicação simples com o FastAPI. Nesse exemplo, é criada uma instância da classe *FastAPI*, armazenada na variável *app*, que passa a representar o núcleo da aplicação, sendo responsável por organizar rotas, modelos e regras de negócio.

É importante destacar que, na Listagem 7.2, as operações CRUD ainda não foram implementadas e o recurso *Estágios* é o único presente na aplicação. Dessa forma, na ausência das operações básicas, nenhuma requisição pode ser efetivamente realizada sobre esse recurso. No entanto, os modelos de dados associados ao recurso já foram definidos. Os modelos *EstagioBase*, *EstagioResponse* e *UpdateEstagio* correspondem, respectivamente, aos modelos de entrada, de saída e de atualização de dados. Esses modelos determinam quais atributos o cliente pode enviar e quais informações o servidor deve retornar nas respostas. Por fim, a aplicação disponibiliza um endpoint raiz que retorna uma mensagem simples de apresentação ao cliente.

```python
from fastapi import FastAPI
from pydantic import BaseModel
from datetime import date

app = FastAPI()

# Modelo de dados para o recurso Estagio
class EstagioBase(BaseModel):
    nome: str
    vertente: Optional[str] = None
    salario: Optional[float] = None
    empresa_id: Optional[int] = None
    remunerado: Optional[bool] = None
    horas_semanais: Optional[int] = None
    descricao: Optional[str] = None
    data_inicio: Optional[date] = None
    data_fim: Optional[date] = None

class EstagioResponse(BaseModel):
    id: int
    nome: str
    vertente: Optional[str] = None
    salario: Optional[float] = None
    empresa_id: Optional[int] = None
    remunerado: Optional[bool] = None
    horas_semanais: Optional[int] = None
    descricao: Optional[str] = None
    data_inicio: Optional[date] = None
    data_fim: Optional[date] = None

class UpdateEstagio(BaseModel):
    nome: Optional[str] = None
    vertente: Optional[str] = None
    salario: Optional[float] = None
    empresa_id: Optional[int] = None
    remunerado: Optional[bool] = None
    horas_semanais: Optional[int] = None
    descricao: Optional[str] = None
    data_inicio: Optional[date] = None
    data_fim: Optional[date] = None

@app.get("/")
def root():
    return {"mensagem": "Ola, Mundo"}
```

Listagem 7.2: API RESTful usando o framework FastAPI

A operação **CREATE** está associada à criação de novos recursos. Assim, quando o cliente deseja cadastrar uma nova oferta de estágio, deve realizar uma requisição com o método HTTP **POST**. No FastAPI, essa funcionalidade é implementada por meio do decorador *@app.post*, que associa a função ao método POST e ao endpoint correspondente. Além disso, é possível definir explicitamente o código de status a ser retornado em caso de sucesso, sendo o **201 CREATED** o mais adequado para operações de criação.

A Listagem 7.3 apresenta um exemplo dessa operação em uma API REST. A função *create_estagio* é executada sempre que o endpoint */estagios* é acessado pelo método POST. Para processar os dados enviados pelo cliente, utiliza-se uma instância de sessão fornecida pela biblioteca **SQLAlchemy**. Após o armazenamento

das informações no banco de dados, o recurso recém-criado é retornado ao cliente no formato JSON, utilizando o modelo *EstagioResponse*. Ressalta-se que o retorno do recurso criado na resposta constitui uma decisão de projeto adotada neste trabalho com finalidade didática.

```
@app.post("/estagios", status_code=status.HTTP_201_CREATED)
def create_estagio(input: EstagioBase):
    with get_session() as session:
        novo_estagio = Estagio(
            nome=input.nome,
            vertente=input.vertente,
            salario=input.salario,
            empresa_id=input.empresa_id,
            remunerado=input.remunerado,
            horas_semanais=input.horas_semanais,
            descricao=input.descricao,
            data_inicio=input.data_inicio,
            data_fim=input.data_fim)
        session.add(novo_estagio)
        session.commit()
        session.refresh(novo_estagio)

        return EstagioResponse(
            id=novo_estagio.id,
            nome=novo_estagio.nome,
            vertente=novo_estagio.vertente,
            salario=novo_estagio.salario,
            empresa_id=novo_estagio.empresa_id,
            remunerado=novo_estagio.remunerado,
            horas_semanais=novo_estagio.horas_semanais,
            descricao=novo_estagio.descricao,
            data_inicio=novo_estagio.data_inicio,
            data_fim=novo_estagio.data_fim)
```

Listagem 7.3: POST em uma API REST

A operação **READ** refere-se à leitura de recursos e utiliza o método HTTP **GET**. Para implementar essa funcionalidade, emprega-se o decorador *@app.get*, que associa a função a operações de consulta. A Listagem 7.4 apresenta um exemplo dessa operação para o recurso *Estágios*. Nesse caso, a função *get_estagios* é responsável por tratar as requisições direcionadas ao endpoint */estagios*, acessar o banco de dados, recuperar os registros disponíveis e retornar uma lista de estágios em formato JSON.

```python
@app.get("/estagios", status_code=status.HTTP_200_OK)
def get_estagios():
    with get_session() as session:
        resultado = session.execute(select(Estagio))
        item = resultado.scalars().all()

        return [EstagioResponse(
            id=row.id,
            nome=row.nome,
            vertente=row.vertente,
            salario=row.salario,
            empresa_id=row.empresa_id,
            horas_semanais=row.horas_semanais,
            remunerado=row.remunerado,
            descricao=row.descricao,
            data_inicio=row.data_inicio,
            data_fim=row.data_fim
        ) for row in item]
```

Listagem 7.4: GET em uma API RESTful

A operação **UPDATE** atua de forma semelhante à operação de criação, pois também requer o envio de um corpo de requisição. No entanto, diferentemente do CREATE, apenas os atributos que se deseja modificar precisam ser informados. Para essa finalidade, utiliza-se o método HTTP **PATCH**, associado ao decorador *@app.patch*. Quando a atualização é realizada com sucesso, o código de status retornado é o **200 OK**.

A Listagem 7.5 exemplifica a implementação dessa operação. A função *update_estagio* recebe dois parâmetros, o corpo da requisição, contendo os dados a serem atualizados, e o identificador do recurso, enviado como parâmetro na URL. Antes de efetuar a atualização, o sistema verifica a existência do recurso no banco de dados. Caso não seja encontrado, retorna-se o código **404 Not Found**. Caso contrário, apenas os campos informados na requisição são modificados, e o recurso atualizado é retornado ao cliente.

```python
@app.patch("/estagios/{estagio_id}", status_code=status.HTTP_200_OK)
def update_estagio(input: UpdateEstagio, estagio_id: int):
    with get_session() as session:
        resultado = session.get(Estagio, estagio_id)
        if not resultado:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="Estagio nao encontrado"
            )

        input_elementos = input.model_dump()
        for key, value in input_elementos.items():
            if value is not None:
                setattr(resultado, key, value)

        session.commit()
        session.refresh(resultado)
        return EstagioResponse(
            id=resultado.id,
            nome=resultado.nome,
            vertente=resultado.vertente,
            salario=resultado.salario,
            empresa_id=resultado.empresa_id,
            remunerado=resultado.remunerado,
            horas_semanais=resultado.horas_semanais,
            descricao=resultado.descricao,
            data_inicio=resultado.data_inicio,
            data_fim=resultado.data_fim
        )
```

Listagem 7.5: PATCH em uma API REST

Por fim, a operação **DELETE** corresponde à exclusão de recursos e utiliza o método HTTP **DELETE**. A Listagem 7.6 apresenta a função *delete_estagio*, responsável por remover um recurso com base em seu identificador. Caso o recurso exista, ele é excluído do banco de dados e uma mensagem de sucesso é retornada. Caso contrário, o servidor responde com o código **404 Not Found**, acompanhado de uma mensagem de erro.

```python
@app.delete("/estagios/{estagio_id}", status_code=status.HTTP_200_OK)
def delete_estagio(estagio_id: int):
    with get_session() as session:
        resultado = session.get(Estagio, estagio_id)
        if not resultado:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="Item nao encontrado"
            )
        session.delete(resultado)
        session.commit()
        return {"mensagem": "Elemento deletado com sucesso"}
```

Listagem 7.6: DELETE em uma API REST

7.2.3 APIs GraphQL

O **GraphQL** é uma linguagem de consulta desenvolvida pelo Facebook em 2012 e disponibilizada ao público como projeto de código aberto em 2015. Sua criação esteve associada à necessidade de tornar mais eficiente a obtenção de dados em aplicações móveis, uma vez que abordagens tradicionais baseadas em múltiplas requisições resultavam em degradação significativa de desempenho. Atualmente, o GraphQL é mantido e supervisionado pela **GraphQL Foundation** e consolidou-se como uma alternativa às APIs REST em diversos cenários de desenvolvimento de software. Em razão de sua ampla adoção, foram desenvolvidos diversos frameworks e ferramentas para apoiar sua implementação, como o *Apollo*[11], o *Offix*[12], o *Graphback*[13] e o *Hasura*[14] (BRITO; VALENTE, 2020).

A principal característica do GraphQL reside em sua orientação explícita ao fornecimento de dados, tendo sido concebido para permitir consultas flexíveis e precisas. Essa abordagem torna a linguagem especialmente adequada para aplicações que lidam com requisições complexas, como serviços web e aplicações móveis, nas quais a eficiência no acesso aos dados é um requisito central. Além disso, por se tratar de uma linguagem fortemente tipada, o GraphQL favorece a validação rigorosa dos dados, tanto no lado do cliente quanto no servidor, contribuindo para a redução de inconsistências e erros em tempo de execução.

Diferentemente das APIs REST tradicionais, que normalmente expõem múltiplos endpoints, as APIs GraphQL operam por meio de um único endpoint, no qual todas as operações de consulta e modificação de dados são concentradas. Essa característica permite que o cliente realize uma única requisição para obter exatamente os dados necessários, reduzindo o número de chamadas ao servidor. Ademais, por se tratar de um modelo declarativo, cabe ao cliente especificar explicitamente quais campos deseja receber em cada resposta, conferindo maior controle sobre o conteúdo retornado.

As aplicações GraphQL são definidas a partir de um **schema**, responsável por descrever a estrutura das entidades, os tipos de dados, os campos disponíveis e as operações permitidas pela aplicação. O schema define tanto a estrutura das consultas quanto a das mutações e é escrito em uma linguagem própria do GraphQL denominada *Schema Definition Language* (SDL) (*Schema Definition Language*). Essa linguagem apresenta uma sintaxe simples e expressiva para definir tipos e campos da API. A Listagem 7.7 apresenta a definição de um *Object Type* denominado *Estagio* na linguagem SDL, na qual são especificados os campos da entidade e os respectivos tipos de dados. O símbolo *!* indica que o campo não admite valores nulos, o que o torna obrigatório.

[11] https://www.apollographql.com/
[12] https://github.com/aerogear/offix
[13] https://github.com/aerogear/graphback
[14] https://hasura.io/

```
type Estagio {
    id: ID!
    nome: String!
    vertente: String
    salario: String
    empresa_id: Int
    remunerado: Boolean
    horas_semanais: Int
    descricao: String
}
```

Listagem 7.7: Definição de um tipo no schema do GraphQL

Além da definição em SDL, os tipos GraphQL podem ser mapeados diretamente a partir de linguagens de programação por meio de frameworks específicos. A Listagem 7.8 apresenta a definição do mesmo tipo *Estagio* em uma API GraphQL implementada em Python com o auxílio do framework *Strawberry*. Nesse caso, o decorador *@strawberry.type* é utilizado para converter classes Python em tipos GraphQL, preservando a estrutura lógica da entidade e simplificando a implementação.

```
@strawberry.type
class Estagio:
    id: int
    nome: str
    vertente: Optional[str] = None
    salario: Optional[float] = None
    empresa_id: Optional[int] = None
    remunerado: Optional[bool] = None
    horas_semanais: Optional[int] = None
    descricao: Optional[str] = None
```

Listagem 7.8: Definição de um tipo em uma API GraphQL

As APIs GraphQL admitem dois tipos principais de operações: as **consultas** (*queries*) e as **mutações** (*mutations*). As consultas correspondem às operações de leitura de dados e são compostas por um conjunto de campos que representam as propriedades solicitadas pelo cliente. Nessa abordagem, o cliente também define explicitamente a estrutura da resposta desejada, permitindo que o servidor retorne apenas os dados solicitados. A Listagem 7.9 exemplifica uma consulta realizada em uma API GraphQL, na qual são solicitados apenas alguns atributos do recurso *Estagio* associado ao identificador 90. Essa flexibilidade contrasta com APIs REST tradicionais, nas quais, frequentemente, a lógica do servidor precisa ser ajustada para atender a diferentes formatos de resposta.

```
query GetIdEstagios {
    getIdEstagios(input: { id: 90 }) {
        nome
        horasSemanais
        descricao
        dataInicio
        dataFim
        vertente
    }
}
```

Listagem 7.9: Consulta realizada em uma API GraphQL

As mutações, por sua vez, correspondem às operações que modificam os dados persistidos, abrangendo ações de criação, atualização e exclusão de recursos. Assim como nas consultas, o cliente define tanto os dados enviados ao servidor quanto os campos que deseja receber na resposta. A Listagem 7.10 apresenta um exemplo de mutação responsável pela criação de um novo recurso *Estagio*, evidenciando a definição explícita dos campos retornados após a execução da operação.

```
mutation CriarEstagio {
    criarEstagio(
        input: {
            nome: "Desenvolvedor junior Back-end"
            vertente: "Ciencia da Computacao"
            salario: 1000
            horasSemanais: 30
            dataInicio: "2025-05-28"
            dataFim: "2026-05-28"
        }
    ) {
        nome
        salario
        vertente
    }
}
```

Listagem 7.10: Operação de mutação para criar um novo recurso

Entre as principais vantagens das APIs GraphQL destacam-se a redução do problema de *over-fetching*, a diminuição do número de requisições necessárias para obtenção de dados complexos e a validação rigorosa proporcionada pela tipagem forte da linguagem. Por outro lado, essas APIs também apresentam desafios, como maior complexidade de implementação, especialmente em aplicações de grande porte, e maior dificuldade na implementação de mecanismos de cache quando comparadas às APIs REST tradicionais. Ademais, a curva de aprendizado pode ser mais acentuada, dependendo da familiaridade prévia do desenvolvedor com o paradigma declarativo adotado pelo GraphQL.

7.2.4 APIs gRPC

Além das APIs REST e GraphQL, destacam-se as APIs gRPC, que se baseiam no paradigma de chamadas de procedimento remoto, conhecido como *Remote*

Procedure Call (RPC) (*Remote Procedure Call*). Esse paradigma simula um modelo de comunicação de alto nível, semelhante ao utilizado em sistemas operacionais, e tem como principal objetivo viabilizar a comunicação cliente-servidor em ambientes distribuídos. Por meio do RPC, torna-se possível realizar chamadas a procedimentos localizados em máquinas ou redes distintas, abstraindo os detalhes de baixo nível relacionados à troca de mensagens (IBM, 2024).

Em aplicações que adotam o paradigma RPC, os clientes podem ser computadores ou processos que solicitam serviços ou informações a outros processos, enquanto os servidores implementam esses serviços e disponibilizam os dados solicitados. Quando uma chamada RPC é realizada, o ambiente que invoca a função é temporariamente suspenso; os parâmetros da chamada são transmitidos para a máquina remota e a execução do procedimento é iniciada. Ao término da execução, os resultados são enviados de volta ao ambiente que realizou a chamada, permitindo que a aplicação cliente retome seu fluxo de execução de forma transparente.

Existem diferentes variações de chamadas RPC, entre as quais se destacam as chamadas **síncronas**, nas quais o cliente aguarda explicitamente a resposta; as chamadas **não bloqueantes**, em que o cliente pode prosseguir com sua execução sem aguardar o término da chamada remota; e as chamadas do tipo **broadcast**, nas quais a mesma solicitação é enviada para múltiplos servidores, retornando ao cliente todos os resultados produzidos (GILLIS et al., 2024).

No contexto de aplicações que utilizam RPC via HTTP, o método HTTP mais empregado é o **POST**, pois permite encapsular chamadas de função e seus parâmetros no corpo da requisição. Dessa forma, para realizar uma chamada RPC, o cliente precisa conhecer previamente o nome do método remoto e os parâmetros esperados. Quanto ao formato dos dados retornados nas respostas, sua definição fica a cargo do servidor, o que tende a restringir a flexibilidade do cliente em comparação a abordagens orientadas a recursos, como as APIs REST (AMAZON, 2024).

Com a aplicação do paradigma RPC no desenvolvimento de APIs, surgiram frameworks específicos para esse propósito, entre os quais se destacam o gRPC[15] e o tRPC[16]. O **gRPC**, desenvolvido pelo Google, permite que aplicações clientes invoquem diretamente métodos implementados em servidores remotos. Sua proposta central baseia-se na definição explícita de serviços, na qual são especificados os métodos disponíveis, seus parâmetros de entrada e os tipos de retorno aceitos. Para descrever a interface desses serviços e a estrutura das mensagens trocadas, o gRPC utiliza buffers de protocolo, definidos por meio de uma *Interface Definition Language* (IDL) (*Interface Definition Language*) (GOOGLE, 2024).

A dinâmica de comunicação cliente-servidor nas APIs gRPC é conceitualmente semelhante à observada em sistemas RPC tradicionais. Uma característica relevante desse framework é o suporte nativo a múltiplas linguagens de programação, permitindo que clientes e servidores sejam implementados em linguagens distintas, como Java, Go, Python e Ruby, desde que sejam compatíveis com o padrão gRPC.

[15] **gRPC**: https://grpc.io/
[16] **tRPC**: https://trpc.io/

O desenvolvimento de uma API gRPC inicia-se com a definição da estrutura de dados em um arquivo **.proto**. Esse arquivo descreve os elementos utilizados na comunicação e é processado pelo compilador do gRPC, que gera automaticamente o código necessário para serialização, desserialização e comunicação entre cliente e servidor. O arquivo *proto* contém dois tipos principais de componentes: as **mensagens**, que representam estruturas de dados compostas por campos nomeados e tipados, e os **serviços**, que especificam os métodos que podem ser invocados remotamente, bem como seus parâmetros e tipos de retorno (GRPC, 2024).

A Listagem 7.11 apresenta a definição das mensagens associadas ao recurso *Estágio* na API gRPC desenvolvida neste trabalho. Observa-se a definição de diferentes mensagens, cada uma relacionada a uma operação específica: *CreateEstagioRequest*, destinada às requisições de criação; *UpdateEstagioRequest*, utilizada nas operações de atualização; *DeleteEstagio*, associada às requisições de exclusão; e, por fim, a mensagem *Estagio*, que representa a estrutura geral do recurso.

```
syntax = "proto3";

message CreateEstagioRequest {
  string nome = 1;
  optional string vertente = 2;
  optional float salario = 3;
  optional int32 empresa_id = 4;
  optional bool remunerado = 5;
  optional int32 horas_semanais = 6;
  optional string descricao = 7;
  optional Data data_inicio = 8;
  optional Data data_fim = 9;
}

message Estagio {
  int32 id = 1;
  string nome = 2;
  string vertente = 3;
  float salario = 4;
  int32 empresa_id = 5;
  bool remunerado = 6;
  int32 horas_semanais = 7;
  string descricao = 8;
  Data data_inicio = 9;
  Data data_fim = 10;
}

message UpdateEstagioRequest {
  int32 id = 1;
  optional string nome = 2;
  optional string vertente = 3;
  optional float salario = 4;
  optional int32 empresa_id = 5;
  optional bool remunerado = 6;
  optional int32 horas_semanais = 7;
  optional string descricao = 8;
  optional Data data_inicio = 9;
  optional Data data_fim = 10;
}

message DeleteEstagio {
  int32 id = 1;
}
```

Listagem 7.11: Arquivo de definição da estrutura dos dados do recurso Estágio na API gRPC

No que se refere aos serviços, estes também são definidos nos arquivos *proto* e especificam os métodos remotos que os clientes podem chamar. A Listagem 7.12 apresenta a definição do serviço *EstagioService*, que inclui cinco métodos principais: *GetEstagio*, *CreateEstagio*, *ListEstagios*, *UpdateEstagio* e *DeleteEstagio*. Com base nessa definição, é possível identificar claramente as estruturas de entrada e de saída associadas a cada método. Após a especificação completa das mensagens e dos serviços, o compilador do gRPC gera automaticamente o código-base necessário para a implementação do cliente e do servidor, bem como as configurações para a troca eficiente de mensagens.

```
service EstagioService {
  rpc GetEstagio (EntidadeRequest) returns (Estagio) {}

  rpc CreateEstagio (CreateEstagioRequest) returns (Estagio) {}

  rpc ListEstagios (Empty) returns (EstagioListResponse) {}

  rpc UpdateEstagio (UpdateEstagioRequest) returns (Estagio) {}

  rpc DeleteEstagio (EntidadeRequest) returns (Empty) {}
}
```

Listagem 7.12: Arquivo de definição dos serviços da API gRPC

A implementação do servidor da API gRPC é apresentada na Listagem 7.13. Nela, observa-se a importação das bibliotecas responsáveis pela comunicação via gRPC, bem como dos arquivos gerados a partir do processamento do arquivo *proto*. O servidor registra os serviços definidos e estabelece a escuta de requisições na porta *50051*, utilizando um *ThreadPoolExecutor* para possibilitar o atendimento simultâneo de múltiplas conexões.

```python
import grpc
from concurrent import futures
import dados_pb2
import dados_pb2_grpc

class EnderecoService(dados_pb2_grpc.EnderecoServiceServicer):
class PlataformaService(dados_pb2_grpc.PlataformaServiceServicer):
class ProfessorService(dados_pb2_grpc.ProfessorServiceServicer):
class BolsaService(dados_pb2_grpc.BolsaServiceServicer):
class EstagioService(dados_pb2_grpc.EstagioServiceServicer):
class CursoService(dados_pb2_grpc.CursoServiceServicer):
class EmpresaService(dados_pb2_grpc.EmpresaServiceServicer):

#iniciando o servidor
async def serve():
    #configurando o servidor o grpc
    servidor = grpc.aio.server(futures.ThreadPoolExecutor(max_workers=10))
    dados_pb2_grpc.add_EnderecoServiceServicer_to_server(EnderecoService(),
    servidor)
    dados_pb2_grpc.add_PlataformaServiceServicer_to_server(PlataformaService(),
    servidor)
    dados_pb2_grpc.add_ProfessorServiceServicer_to_server(ProfessorService(),
    servidor)
    dados_pb2_grpc.add_BolsaServiceServicer_to_server(BolsaService(),
    servidor)
    dados_pb2_grpc.add_EstagioServiceServicer_to_server(EstagioService(),
    servidor)
    dados_pb2_grpc.add_CursoServiceServicer_to_server(CursoService(),
    servidor)
    dados_pb2_grpc.add_EmpresaServiceServicer_to_server(EmpresaService(),
    servidor)
    servidor.add_insecure_port('[::]:50051')
    await servidor.start()
    print("Servidor inicializou na porta 50051")
    await servidor.wait_for_termination()

if __name__ == '__main__':
    asyncio.run(serve())
```

Listagem 7.13: Servidor API gRPC

Por fim, a Listagem 7.14 apresenta a implementação do cliente da API gRPC. O cliente estabelece um canal de comunicação com o servidor, define um *stub* correspondente ao serviço remoto e invoca as operações básicas associadas ao recurso *Estágio*. Os *stubs* atuam como intermediários entre cliente e servidor, sendo responsáveis por gerenciar a serialização das mensagens, o envio das requisições e o recebimento das respostas (ALRAZAK, 2023). Dessa forma, são realizadas as operações de criação, leitura, atualização e exclusão do recurso, o que conclui o fluxo básico de comunicação em uma API gRPC.

```python
import asyncio
import grpc
import dados_pb2
import dados_pb2_grpc
from clientes_entidades.cliente_estagio import create_estagios, get_all_estagios
    , get_estagio_id, update_estagio, delete_estagio


options = [
    ('grpc.max_receive_message_length', 50 * 1024 * 1024),
]
channel = grpc.insecure_channel('localhost:50051', options=options)

async def run():
    async with grpc.aio.insecure_channel('localhost:50051'):
        stub = dados_pb2_grpc.EstagioServiceStub(channel)

        request = {"nome": 'Desenvolvedor de jogos junior', "vertente": "Ciencia
            da computacao"}
        create_estagio(stub, dados_pb2, request)
        get_all_estagios(stub)
        get_estagio_id(stub, dados_pb2, {'id': 89})
        update_estagio(stub, dados_pb2, {"id": 35527, "nome": "Tecnico de redes"
            })
        delete_estagio(stub, dados_pb2, {"id": 36533})


if __name__ == '__main__':
    asyncio.run(run())
```

Listagem 7.14: Cliente da API gRPC

7.3 Exemplo de uso das APIs

Ao longo da fundamentação teórica deste capítulo, foram apresentadas três APIs de exemplo com o objetivo de discutir, de forma progressiva, os conceitos associados aos diferentes padrões de arquitetura de APIs web. Essas aplicações desempenharam um papel central na aproximação entre os fundamentos teóricos abordados e suas respectivas implementações práticas. Considerando que o objetivo principal deste trabalho é a realização de um estudo comparativo, as APIs inicialmente concebidas como exemplos foram posteriormente ampliadas, refinadas e consolidadas como objetos formais de estudo.

Nesse contexto, foram desenvolvidas três APIs web baseadas nos padrões **REST**, **GraphQL** e **gRPC**, que serviram de base para a execução dos testes comparativos apresentados nos capítulos subsequentes. Conforme discutido anteriormente, as três aplicações foram implementadas na mesma linguagem de programação, Python, e compartilham as mesmas entidades, operações e regras de negócio. Essa decisão metodológica teve como finalidade reduzir vieses experimentais e assegurar maior confiabilidade, precisão e comparabilidade dos resultados obtidos.

Para a conceitualização das APIs, definiu-se um cenário fictício comum, no qual as aplicações seriam consumidas por um sistema web responsável por divulgar

Tabela 7.5: Entidades representadas no banco de dados desenvolvido para as APIs

Entidade	**Objetivo**
Bolsa	Bolsas acadêmicas ofertadas, remuneradas e não-remuneradas
Cursos	Curso disponíveis, gratuitos e pagos
Estágios	Estágios ofertados, remunerados e não-remunerados
Professores	Professores responsáveis por bolsas
Empresas	Empresas ofertando os estágios
Endereços	Endereços das empresas
Plataformas	Plataformas que disponibilizam os cursos

Tabela 7.6: Operações que as APIs desenvolvidas podem executar

Operação	**Método utilizado**	**Objetivo**
Cadastro	POST	Cadastrar um novo elemento.
Consulta individual	GET	Recuperar informações sobre um elemento específico.
Consulta múltipla	GET	Recuperar uma lista com todos os elementos.
Atualização parcial	PATCH	Atualizar um ou mais atributos de determinado elemento.
Exclusão	DELETE	Deletar um elemento específico.

oportunidades acadêmicas para estudantes do curso de Engenharia de Computação, como estágios, bolsas acadêmicas e cursos de capacitação. A partir desse contexto, foram definidas as entidades centrais do domínio da aplicação e as operações básicas que as três APIs deveriam oferecer suporte. A Tabela 7.5 apresenta as entidades modeladas, enquanto a Tabela 7.6 descreve as operações que podem ser realizadas sobre esses dados.

Após a definição das entidades e das operações, foram definidos os formatos de entrada e de saída adotados pelas APIs. Para as aplicações REST e GraphQL, definiu-se o uso do formato JSON tanto nas requisições quanto nas respostas, em conformidade com as práticas mais comuns desses padrões. Em contrapartida, a API gRPC utiliza buffers de protocolo como formato de comunicação, em razão de sua fundamentação no protocolo HTTP/2 e no uso de mensagens binárias serializadas.

Considerando que o propósito do estudo comparativo é analisar os padrões de arquitetura sem introduzir variações que descaracterizem suas abordagens originais, optou-se por manter o uso de *protocol buffers* na API gRPC, sem adaptações para formatos textuais como JSON. De forma semelhante, funcionalidades relacionadas à autenticação e autorização não foram incorporadas às aplicações, uma vez que diferentes estratégias poderiam impactar os resultados dos testes e comprometer a imparcialidade da comparação.

Por fim, ressalta-se que o código-fonte das três APIs desenvolvidas neste estudo está disponível publicamente no GitHub, conforme indicado a seguir: [17] [18] [19].

7.3.1 Construção do banco de dados

Antes da implementação das três APIs, foi desenvolvido o banco de dados que serve como fonte comum de dados para todas as aplicações. Essa decisão metodológica teve como objetivo garantir imparcialidade nos testes comparativos, assegurando que todas as requisições realizadas pelas APIs consumissem exatamente o mesmo conjunto de dados, eliminando variações decorrentes de diferenças na origem das informações.

Inicialmente, realizou-se um levantamento das entidades e dos tipos de dados necessários para representar o domínio da aplicação, conforme apresentado na Tabela 7.7. Esse levantamento orientou a criação das tabelas do banco de dados e a definição de seus respectivos atributos, considerando as características funcionais de cada entidade e suas relações no contexto do sistema proposto.

Após a definição das entidades e de seus atributos, foi elaborado o esquema do banco de dados, apresentado na Figura 7.12. O modelo foi construído utilizando a ferramenta **dbdesigner**, na qual foram definidas as chaves primárias, as chaves estrangeiras, os tipos de dados e os relacionamentos entre as tabelas. Todas as entidades adotam um identificador único como chave primária, garantindo a unicidade dos registros.

Algumas entidades possuem chaves estrangeiras que estabelecem relacionamentos com outras tabelas, como ocorre, por exemplo, na entidade Curso, que possui o atributo *plataforma_id*, responsável por associar cada curso à plataforma que o disponibiliza. A Tabela 7.8 sintetiza os principais relacionamentos definidos no banco de dados.

Após a modelagem, procedeu-se à escolha do sistema de gerenciamento de banco de dados. Considerando a necessidade de lidar com grandes volumes de dados e múltiplas conexões simultâneas durante os testes de desempenho e carga, optou-se pelo **MySQL**, versão 8.0.36. Esse SGBD foi escolhido por sua ampla adoção em aplicações web de larga escala, pelo suporte ao modelo cliente-servidor e pela compatibilidade com as bibliotecas utilizadas neste estudo. Para a modelagem e a administração do banco de dados, adotou-se o **MySQL Workbench**, versão 8.0.36.

Após a especificação das entidades e ferramentas a serem utilizadas, o próximo estágio foi a construção do banco de dados e a sua população com registros fictícios para fins de teste de comparação entre as três APIs. Foi escolhida a biblioteca **Faker**, disponibilizada na linguagem Python, para gerar os dados falsos, tal decisão foi adotada porque os dados de exemplo gerados pela biblioteca são razoavelmente realistas, pela variedade de métodos para diferentes tipos de dados, como nomes,

[17] **API REST:** https://github.com/samylesousa/apiREST
[18] **API GraphQL:** https://github.com/samylesousa/apiGraphQL
[19] **API gRPC:** https://github.com/samylesousa/apiGRPC

Tabela 7.7: Entidades e atributos do banco de dados das APIs

Entidade	**Atributo**	**Tipo de dado**
Curso	nome	VARCHAR
	categoria	VARCHAR
	preco	DECIMAL
	nivel	VARCHAR
	vertente	VARCHAR
	data_inicial	DATE
	data_final	DATE
	plataforma_id	INT
Estágio	nome	VARCHAR
	vertente	VARCHAR
	remunerado	TINYINT
	salario	DECIMAL
	horas_semanais	INT
	descricao	TEXT
	data_inicial	DATE
	data_final	DATE
	empresa_id	INT
Bolsa	nome	VARCHAR
	remunerada	TINYINT
	vertente	VARCHAR
	salario	DECIMAL
	quantidade_vagas	INT
	horas_semanais	INT
	descricao	VARCHAR
	data_inicial	DATE
	data_final	DATE
	professor_id	INT
Professor	nome	VARCHAR
	vertente	VARCHAR
	telefone	VARCHAR
	email	VARCHAR
	website	VARCHAR
	formacao	VARCHAR
Empresa	nome	VARCHAR
	vertente	VARCHAR
	cnpj	VARCHAR
	telefone	VARCHAR
	email	VARCHAR
	website	VARCHAR
	status	TINYINT
	endereco_id	INT
Endereço	rua	VARCHAR
	numero	INT
	bairro	VARCHAR
	cidade	VARCHAR
	estado	CHAR
	cep	CHAR
Plataforma	nome	VARCHAR
	vertente	VARCHAR
	email	VARCHAR
	website	VARCHAR
	tipo	TINYINT

Tabela 7.8: Relacionamentos entre as tabelas do banco de dados

Tabela 1	Tabela 2	Tipo de Relação	Chave Estrangeira
curso	plataforma	1:1	curso.plataforma_id → plataforma.id
professor	bolsa	1:N	professor.bolsa_id → bolsa.id
empresa	estagio	1:N	empresa.estagio_id → estagio.id
empresa	endereco	1:1	empresa.endereco_id → endereco.id

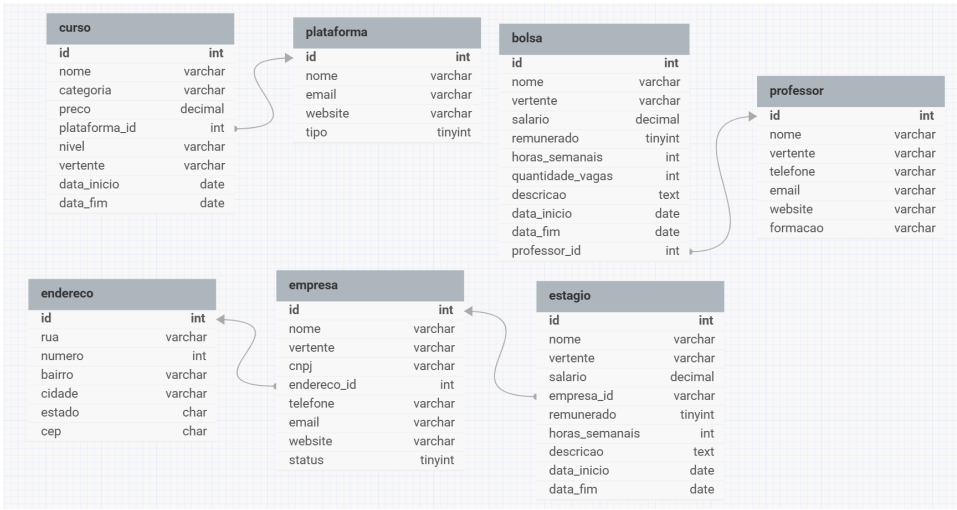


Figura 7.12: Representação visual do banco de dados

id	rua	numero	bairro	cidade	estado	cep
36522	Condomínio de ...	145	Jardim Leblon	Lima de Costa	PI	02630509
36521	Conjunto de Ro...	93	Pedreira Padro Lopes	Marques de Bor...	MA	88726-956
36520	Trecho Nina Ca...	546	Santa Rosa	Moreira	RJ	64976-833
36519	Praia de da Rosa	942	Conjunto Santa Maria	da Cunha de da...	AP	80314-385
36518	Vale Ester Marq...	857	Vila Nova Gameleira ...	Martins	CE	91954153
36516	Favela de Mend...	248	Jardim Alvorada	Mendonça de A...	PB	80752308
36515	Conjunto de An...	586	Vila Tirol	Cunha	AP	66886741
36514	Estrada Rodrig...	946	Mantiqueira	Rocha	PB	35887-273
36513	Viela Alves	751	Maravilha	Siqueira de Goiás	ES	41422-895
36512	Travessa da Mata	715	Carmo	Câmara do Ca...	AC	25850-713
36511	Quadra de And...	718	Vila Novo São Lucas	Oliveira	GO	00573330
36510	Campo de Freitas	689	Campo Alegre	Cardoso de Vieira	SP	04608814

Figura 7.13: Exemplos dos dados fictícios gerados pela biblioteca Faker para a tabela Endereço

endereços e telefones. No entanto, mesmo utilizando todas as ferramentas oferecidas pela biblioteca, alguns dados de teste não refletem adequadamente a realidade, todavia, tal detalhe não afetou os testes realizados nas APIs. A Figura 7.13 exibe uma parcela dos dados falsos gerados para a tabela Endereço.

7.3.2 Integração com o banco de dados

Com o objetivo de manter uniformidade na camada de persistência, as três APIs utilizaram a biblioteca **SQLAlchemy** para o acesso e a manipulação do banco de dados. Essa escolha possibilita o trabalho com os dados por meio de objetos do Python, o que simplifica a implementação, melhora a legibilidade do código e facilita a replicação dos experimentos em estudos futuros. Além disso, o SQLAlchemy oferece suporte nativo ao SGBD MySQL e integração com programação assíncrona, característica relevante no contexto das aplicações desenvolvidas neste estudo (DATTA, 2024).

De maneira geral, em cada API foram definidos dois arquivos relacionados à configuração e ao mapeamento do banco de dados. O primeiro arquivo é responsável pela inicialização da *engine* de conexão e pela implementação de uma função que, quando invocada, retorna uma sessão ativa para interação com o banco. A Listagem 7.15 apresenta o script correspondente a esse arquivo, no qual são importados os métodos necessários das bibliotecas utilizadas, seguido da especificação da URL de conexão com o banco de dados e da criação da *engine* assíncrona.

Ao final do código, destaca-se a função *get_session*, que fornece uma sessão de banco de dados a ser utilizada de forma transitória em um contexto controlado. Essa abordagem assegura que a sessão seja corretamente encerrada após o uso, evitando vazamentos de conexão e contribuindo para a estabilidade da aplicação. Ressalta-se que esse código, assim como as três APIs desenvolvidas, utiliza a

biblioteca **asyncio**[20] para a execução de funções assíncronas, de modo que a criação e o gerenciamento das conexões com o banco de dados ocorrem de forma não bloqueante.

```python
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import sessionmaker
from contextlib import asynccontextmanager
from dotenv import load_dotenv
import os

load_dotenv()

DATABASE_URL = os.getenv("DATABASE")
engine = create_async_engine(DATABASE_URL, echo=False)
SessionLocal = sessionmaker(bind=engine, class_=AsyncSession, expire_on_commit=
    False)

@asynccontextmanager
async def get_session():
    async with SessionLocal() as session:
        yield session
```

Listagem 7.15: Criação e inicialização da conexão com o banco

```python
import sqlalchemy as sa
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
class Curso(Base):
    __tablename__ = 'curso'
    def __str__(self):
        return f"""Nome: {self.nome},
            Vertente: {self.vertente},
            Categoria: {self.categoria},
            Preco: {self.preco},
            Plataforma Id: {self.plataforma_id},
            Nivel: {self.nivel},
            Data de Inicio: {self.data_inicio},
            Data de Fim: {self.data_fim}
        """

    id = sa.Column(sa.Integer, primary_key=True)
    nome = sa.Column(sa.String)
    categoria = sa.Column(sa.String)   #se e pago ou nao
    preco = sa.Column(sa.Float)
    plataforma_id =  sa.Column(sa.Integer, sa.ForeignKey('plataforma.id'))
    nivel = sa.Column(sa.String)    #as tres opcoes sao, iniciante,
        intermediario e avancado
    vertente = sa.Column(sa.String)
    data_inicio = sa.Column(sa.DATE)
    data_fim = sa.Column(sa.DATE)

    #relacionamento com a tabela plataforma (many to one)
    plataforma = sa.orm.relationship("Plataforma", back_populates="cursos")
```

Listagem 7.16: Definição das classes que representam as tabelas

[20]**asyncio**: https://docs.python.org/3/library/asyncio.html

7.3.3 Documentação das APIs

Embora as três APIs desenvolvidas compartilhem a mesma base conceitual, as mesmas entidades e regras de negócio, cada padrão arquitetural impõe o uso de bibliotecas e frameworks específicos. As APIs REST e GraphQL utilizam o protocolo HTTP/1.1 como base de comunicação, enquanto a API gRPC é fundamentada no protocolo HTTP/2, o que implica diferenças significativas tanto no modelo de comunicação quanto nas ferramentas adotadas.

Nesse contexto, o framework FastAPI foi empregado no desenvolvimento de APIs REST e GraphQL, uma vez que oferece suporte nativo à construção de serviços HTTP, à validação automática de dados e à documentação integrada. Por outro lado, o FastAPI não foi utilizado na API gRPC, pois esse padrão adota uma lógica distinta, baseada na definição explícita de serviços e mensagens por meio de arquivos *.proto*, os quais são processados por ferramentas específicas do ecossistema gRPC.

A Tabela 7.9 apresenta as principais bibliotecas e frameworks utilizados em cada uma das APIs desenvolvidas. Observa-se que algumas bibliotecas são comuns às três implementações, como o **SQLAlchemy**, responsável por abstrair o acesso ao banco de dados relacional, e o **Asyncmy**[21], utilizado como driver assíncrono para comunicação com o MySQL.

O framework **FastAPI**[22] foi aplicado exclusivamente nas APIs REST e GraphQL, por ser voltado ao desenvolvimento de aplicações HTTP. No caso específico do GraphQL, também foi utilizada a biblioteca **Strawberry**[23], responsável por auxiliar na definição do *schema* e na construção dos resolvers.

Já no contexto da API gRPC, destacam-se as bibliotecas **grpcio**[24], que fornece os recursos necessários para a implementação de clientes e servidores gRPC, e **grpcio-tools**[25], utilizada para a geração automática de código Python a partir dos arquivos *.proto*. Essa diversidade de ferramentas reflete as particularidades de cada padrão arquitetural e evidencia as diferenças estruturais entre as abordagens analisadas.

7.3.4 API REST

Após a configuração do banco de dados na API REST, a etapa seguinte do desenvolvimento consistiu na definição dos modelos de dados utilizados nos corpos das requisições e das respostas. Esses modelos foram implementados por meio dos *Pydantic Models*, disponibilizados pelo framework FastAPI, que possibilitam a validação automática dos dados de entrada e a padronização das estruturas de dados trafegadas entre o cliente e o servidor.

Considerando que as operações de leitura apresentam características distintas das de criação e atualização, tornou-se necessário definir modelos específicos para

[21] https://pypi.org/project/asyncmy/
[22] https://fastapi.tiangolo.com/
[23] https://strawberry.rocks/
[24] https://pypi.org/project/grpcio/
[25] https://pypi.org/project/grpcio-tools/

Tabela 7.9: Frameworks e Bibliotecas integradas nas APIs

GRAPHQL	**GRPC**	**REST**
SQLAlchemy	SQLAlchemy	SQLAlchemy
Asyncmy	Asyncmy	Asyncmy
Strawberry	grpcio[26]	FastAPI
FastAPI	grpcio-tools	

cada tipo de interação. A Listagem 7.17 apresenta três modelos de dados associados ao recurso *Bolsa*. O primeiro, denominado *BolsaBase*, representa o modelo base utilizado nas operações de criação, no qual apenas o atributo *nome* é obrigatório, uma vez que o identificador do recurso é gerado automaticamente pelo banco de dados. O segundo modelo, *BolsaResponse*, é utilizado nas respostas da API e, por esse motivo, define os atributos *id* e *nome* como obrigatórios. Por fim, o modelo *UpdateBolsa* é utilizado nas operações de atualização e caracteriza-se por definir todos os atributos como opcionais, permitindo que o cliente envie apenas os campos que deseja modificar.

```
class BolsaBase(BaseModel):
    nome: str
    vertente: Optional[str] = None
    salario: Optional[float] = None
    remunerado: Optional[bool] = None
    horas_semanais: Optional[int] = None
    quantidade_vagas: Optional[int] = None
    descricao: Optional[str] = None
    data_inicio: Optional[date] = None
    data_fim: Optional[date] = None
    professor_id: Optional[int] = None
class BolsaResponse(BaseModel):
    id: int
    nome: str
    vertente: Optional[str] = None
    salario: Optional[float] = None
    remunerado: Optional[bool] = None
    horas_semanais: Optional[int] = None
    quantidade_vagas: Optional[int] = None
    descricao: Optional[str] = None
    data_inicio: Optional[date] = None
    data_fim: Optional[date] = None
    professor_id: Optional[int] = None
class UpdateBolsa(BaseModel):
    nome: Optional[str] = None
    vertente: Optional[str] = None
    salario: Optional[float] = None
    remunerado: Optional[bool] = None
    horas_semanais: Optional[int] = None
    quantidade_vagas: Optional[int] = None
    descricao: Optional[str] = None
    data_inicio: Optional[date] = None
    data_fim: Optional[date] = None
    professor_id: Optional[int] = None
```

Listagem 7.17: Definição das classes que representam as tabelas

Posteriormente, foram implementadas as funções e rotas responsáveis pelas operações CRUD da API REST. Conforme ilustrado na Listagem 7.18, que apresenta a operação de leitura associada ao recurso *Bolsa*, todas as funções da API REST foram implementadas de forma assíncrona. Quando o cliente realiza uma requisição ao endpoint `/bolsas`, a função `get_bolsas` é acionada, estabelece uma sessão assíncrona com o banco de dados, recupera os registros da tabela correspondente e os retorna no formato definido pelo modelo *BolsaResponse*.

Além disso, foram incorporados mecanismos de tratamento de exceções por meio da estrutura **try/except**, com o objetivo de capturar eventuais falhas durante a execução das consultas e retornar respostas apropriadas ao cliente. Essa abordagem contribui para a robustez da API e para a padronização das mensagens de erro. Ressalta-se que todas as demais funções associadas aos recursos definidos na conceitualização das APIs seguem a mesma estrutura e lógica apresentadas neste exemplo, diferenciando-se apenas nos detalhes específicos de cada operação e entidade.

```python
#funcao de leitura das bolsas academicas
@app.get("/bolsas", status_code=status.HTTP_200_OK)
async def get_bolsas():
    async with get_session() as session:
        try:
            resultado = await session.execute(select(Bolsa))
            item = resultado.scalars().all()

            return [BolsaResponse(
                id=row.id,
                nome=row.nome,
                vertente=row.vertente,
                salario=row.salario,
                professor_id=row.professor_id,
                horas_semanais=row.horas_semanais,
                remunerado=row.remunerado,
                quantidade_vagas=row.quantidade_vagas,
                data_inicio=row.data_inicio,
                data_fim=row.data_fim,
                descricao=row.descricao
                ) for row in item]

        except Exception as e:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail=f"As bolsas nao foram encontradas: {str(e)}"
            )
```

Listagem 7.18: Função de leitura para as bolsas acadêmicas da API REST

7.3.5 API GraphQL

No que se refere à API GraphQL desenvolvida para os testes comparativos, esta seguiu o mesmo esquema de configuração do banco de dados adotado pela API REST e também utilizou o framework FastAPI como base para a definição do endpoint da aplicação. Entretanto, diferentemente da API REST, que opera com múltiplos endpoints, a API GraphQL foi projetada para utilizar um único endpoint, definido como */graphql*. Ressalta-se que essa denominação constitui uma decisão de projeto e pode variar conforme a implementação adotada.

Para facilitar o desenvolvimento da API, foi empregada a biblioteca **Strawberry**, que permite definir o *schema* GraphQL diretamente a partir de classes Python, reduzindo a complexidade da implementação e promovendo maior integração com o ecossistema da linguagem. Após a configuração dos módulos necessários, foram definidos os tipos de objetos que compõem a API. A Listagem 7.19 apresenta a declaração de cinco tipos associados às operações relacionadas à entidade *Bolsa*.

O tipo *GetIDType* é utilizado nas operações que recebem informações enviadas pelo cliente, especificamente o identificador da entidade. O tipo *MensagemInput* está associado às respostas que retornam mensagens de status da operação. O tipo *BolsaType* representa a estrutura principal da entidade *Bolsa* e é utilizado nas respostas às requisições, com os atributos *id* e *nome* definidos como obrigatórios. Já o tipo *BolsaInputCreate* é utilizado nas operações de cadastro, não possui o atributo

id, uma vez que este é gerado automaticamente pelo banco de dados, e exige apenas o campo *nome* como obrigatório. Por fim, o tipo *BolsaUpdateInput* é empregado nas operações de atualização, nas quais apenas o identificador é obrigatório, permitindo que os demais atributos sejam modificados conforme a necessidade do cliente.

```
@strawberry.input
class GetIDType:
    id: int
@strawberry.type
class MensagemInput:
    ok: Optional[bool]
    message: str
@strawberry.type
class BolsaType:
    id: int
    nome: str
    vertente: Optional[str] = None
    salario: Optional[float] = None
    remunerado: Optional[bool] = None
    horas_semanais: Optional[int] = None
    quantidade_vagas: Optional[int] = None
    descricao: Optional[str] = None
    data_inicio: Optional[datetime.date] = None
    data_fim: Optional[datetime.date] = None
    professor_id: Optional[int] = None
@strawberry.input
class BolsaInputCreate:
    nome: str
    vertente: Optional[str] = None
    salario: Optional[float] = None
    remunerado: Optional[bool] = None
    horas_semanais: Optional[int] = None
    quantidade_vagas: Optional[int] = None
    descricao: Optional[str] = None
    data_inicio: Optional[datetime.date] = None
    data_fim: Optional[datetime.date] = None
    professor_id: Optional[int] = None
@strawberry.input
class BolsaUpdateInput:
    id: int
    nome: Optional[str] = None
    vertente: Optional[str] = None
    salario: Optional[float] = None
    remunerado: Optional[bool] = None
    horas_semanais: Optional[int] = None
    quantidade_vagas: Optional[int] = None
    descricao: Optional[str] = None
    data_inicio: Optional[datetime.date] = None
    data_fim: Optional[datetime.date] = None
```

Listagem 7.19: Tipos de objetos da API GraphQL

Na sequência, foram implementadas as funções responsáveis por atender às requisições dos clientes da API GraphQL. Para fins ilustrativos, a Listagem 7.20 apresenta duas funções associadas à entidade *Bolsa*: uma de leitura, responsável por retornar todas as bolsas acadêmicas cadastradas, e uma de atualização, que recebe como parâmetro o corpo da requisição com os atributos a serem atualizados.

De modo geral, as funções da API GraphQL seguem a mesma lógica estrutural das funções implementadas na API REST, o que constitui uma decisão metodológica

adotada para assegurar a imparcialidade dos testes comparativos. Ambas utilizam programação assíncrona e compartilham o mesmo fluxo de acesso ao banco de dados, diferenciando-se apenas pelas convenções e abstrações impostas pelo padrão GraphQL.

```
async def get_bolsas():
    async with get_session() as session:
        try:
            resultado = await session.execute(select(Bolsa))
            item = resultado.scalars().all()
            return [BolsaType(id=row.id,
                nome=row.nome,
                vertente=row.vertente,
                salario=row.salario,
                professor_id=row.professor_id,
                horas_semanais=row.horas_semanais,
                remunerado=row.remunerado,
                quantidade_vagas=row.quantidade_vagas,
                data_inicio=row.data_inicio,
                data_fim=row.data_fim,
                descricao=row.descricao) for row in item]
        except Exception as e:
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND)
async def update_bolsa(self, input: BolsaUpdateInput) -> BolsaType:
    async with get_session() as session:
        try:
            resultado = await session.get(Bolsa, input.id)
            if not resultado: raise Exception("Bolsa nao foi encontrada")
            input_elementos = asdict(input)
            for key, value in input_elementos.items():
                if key != "id" and value is not None:
                    setattr(resultado, key, value)
            await session.commit()
            await session.refresh(resultado)
            return BolsaType(id=resultado.id,
                nome=resultado.nome,
                vertente=resultado.vertente,
                salario=resultado.salario,
                remunerado=resultado.remunerado,
                horas_semanais=resultado.horas_semanais,
                quantidade_vagas=resultado.quantidade_vagas,
                descricao=resultado.descricao,
                data_inicio=resultado.data_inicio,
                data_fim=resultado.data_fim,
                professor_id=resultado.professor_id)
        except Exception as e:
            await session.rollback()
            raise Exception(f"Erro: {str(e)}")
```

Listagem 7.20: Função de leitura para as bolsas acadêmicas da API GraphQL

Após a definição das funções, foram estruturadas as classes responsáveis pelas consultas e mutações da API. A Listagem 7.21 apresenta a classe *Query*, que agrupa as operações de leitura, e a classe *Mutation*, responsável pelas operações de modificação dos dados. As demais entidades da API foram implementadas seguindo o mesmo padrão estrutural. Por fim, o *schema* da aplicação foi definido utilizando o recurso de *federation* da biblioteca Strawberry, o que permite maior flexibilidade e extensibilidade na definição da API.

```python
@strawberry.type
class Query:
    getBolsas: List[BolsaType] = strawberry.field(resolver=get_bolsas)
    getIdBolsas: BolsaType = strawberry.field(resolver=getbyid_bolsa)


@strawberry.type
class Mutation:
    criarBolsa: BolsaType = strawberry.field(resolver=criar_bolsa)
    updateBolsa: BolsaType = strawberry.field(resolver=update_bolsa)
    deleteBolsa: MensagemInput = strawberry.field(resolver=delete_elementos(
        Bolsa))

schema = strawberry.federation.Schema(query=Query, mutation=Mutation)
```

Listagem 7.21: Consultas e Mutações definidas para a API GraphQL

7.3.6 API gRPC

No desenvolvimento da API gRPC, manteve-se a mesma lógica de negócio e o mesmo modelo conceitual adotados nas APIs REST e GraphQL, assegurando consistência metodológica entre as três abordagens analisadas. Essa decisão foi fundamental para garantir que as diferenças observadas nos testes comparativos fossem atribuídas exclusivamente aos padrões arquiteturais, e não a variações no domínio da aplicação ou nas regras de negócio.

A primeira etapa da implementação consistiu na definição do arquivo *.proto*, responsável por especificar as mensagens e os serviços associados às entidades da aplicação. Nesse arquivo são descritos, de forma explícita, as estruturas de dados utilizadas na comunicação e os métodos remotos que podem ser invocados pelos clientes, constituindo o contrato formal entre cliente e servidor no padrão gRPC.

A Listagem 7.22 apresenta a definição das mensagens e dos serviços relacionados à entidade *Bolsa*. Observa-se que o serviço *BolsaService* expõe cinco métodos correspondentes às operações básicas de manipulação de dados: leitura individual (*GetBolsa*), criação (*CreateBolsa*), leitura múltipla (*ListBolsas*), atualização (*UpdateBolsa*) e exclusão (*DeleteBolsa*). Cada método especifica explicitamente os tipos de mensagens de entrada e saída, reforçando o caráter fortemente tipado da comunicação no gRPC.

As mensagens definidas descrevem as estruturas de dados trocadas entre o cliente e o servidor. Por exemplo, a mensagem *CreateBolsaRequest* representa os dados necessários para a criação de uma nova bolsa acadêmica, enquanto a mensagem *Bolsa* define a estrutura completa do recurso retornado nas respostas. Já a mensagem *BolsaListResponse* encapsula uma lista de registros, permitindo o retorno de múltiplas instâncias em uma única chamada.

```
service BolsaService {
  rpc GetBolsa (EntidadeRequest) returns (Bolsa) {}
  rpc CreateBolsa (CreateBolsaRequest) returns (Bolsa) {}
  rpc ListBolsas (Empty) returns (BolsaListResponse) {}
  rpc UpdateBolsa (UpdateBolsaRequest) returns (Bolsa) {}
  rpc DeleteBolsa (EntidadeRequest) returns (Empty) {} }
message CreateBolsaRequest {
  string nome = 1;
  optional string vertente = 2;
  optional float salario = 3;
  optional bool remunerado = 4;
  optional int32 horas_semanais = 5;
  optional int32 quantidade_vagas = 6;
  optional string descricao = 7;
  optional int32 professor_id = 8;
  optional string data_inicio = 9;
  optional string data_fim = 10; }
message Bolsa {
  int32 id = 1;
  string nome = 2;
  optional string vertente = 3;
  optional float salario = 4;
  optional bool remunerado = 5;
  optional int32 horas_semanais = 6;
  optional int32 quantidade_vagas = 7;
  optional string descricao = 8;
  optional string data_inicio = 9;
  optional string data_fim = 10;
  optional int32 professor_id = 11;
  bool success = 12; }
message BolsaListResponse {
  repeated Bolsa bolsas = 1; }
message UpdateBolsaRequest {
  int32 id = 1;
  optional string nome = 2;
  optional string vertente = 3;
  optional float salario = 4;
  optional bool remunerado = 5;
  optional int32 horas_semanais = 6;
  optional int32 quantidade_vagas = 7;
  optional string descricao = 8;
  optional int32 professor_id = 9;
  optional string data_inicio = 10;
  optional string data_fim = 11; }
```

Listagem 7.22: Definição das mensagens e serviços da entidade Bolsa

Após a definição das mensagens e serviços no arquivo *.proto*, procedeu-se à implementação das funções responsáveis por atender às requisições realizadas pelos clientes da API gRPC. De maneira semelhante à adotada nas APIs REST e GraphQL, as funções foram implementadas de forma assíncrona, utilizando o mesmo fluxo de acesso ao banco de dados, de modo a preservar a uniformidade metodológica entre as abordagens.

A Listagem 7.23 apresenta a implementação das funções associadas à entidade *Bolsa*. A função *GetBolsa* é apresentada detalhadamente, enquanto as funções *CreateBolsa*, *ListBolsas*, *UpdateBolsa* e *DeleteBolsa* são mostradas de forma resumida, uma vez que seguem a mesma estrutura lógica. Diferentemente das APIs REST e GraphQL, no padrão gRPC as operações são implementadas como métodos de uma

classe de serviço, neste caso *BolsaService*, refletindo o modelo orientado a serviços característico desse padrão.

```
class BolsaService(dados_pb2_grpc.BolsaServiceServicer):
    async def GetBolsa(self, request: dados_pb2.EntidadeRequest, context):
        async with get_session() as session:
            item = await session.get(BolsaModel, request.id)
            if item is None:
                context.set_code(grpc.StatusCode.NOT_FOUND)
                context.set_details("Item Indisponivel")
                return dados_pb2.Bolsa()
            return dados_pb2.Bolsa(id=item.id,
                nome=item.nome,
                vertente=item.vertente,
                salario=item.salario,
                remunerado=item.remunerado,
                horas_semanais=item.horas_semanais,
                quantidade_vagas=item.quantidade_vagas,
                descricao=item.descricao,
                data_inicio=str(item.data_inicio),
                data_fim=str(item.data_inicio),
                professor_id=item.professor_id)
    async def CreateBolsa(self, request: dados_pb2.CreateBolsaRequest, context):
    async def ListBolsas(self, request, context):
    async def UpdateBolsa(self, request: dados_pb2.UpdateBolsaRequest, context):
    async def DeleteBolsa(self, request: dados_pb2.EntidadeRequest, context):
async def serve():
    servidor = grpc.aio.server(futures.ThreadPoolExecutor(max_workers=10))
    dados_pb2_grpc.add_EnderecoServiceServicer_to_server(EnderecoService(),
        servidor)
    dados_pb2_grpc.add_PlataformaServiceServicer_to_server(PlataformaService(),
        servidor)
    dados_pb2_grpc.add_ProfessorServiceServicer_to_server(ProfessorService(),
        servidor)
    dados_pb2_grpc.add_BolsaServiceServicer_to_server(BolsaService(), servidor)
    dados_pb2_grpc.add_CursoServiceServicer_to_server(CursoService(), servidor)
    dados_pb2_grpc.add_EmpresaServiceServicer_to_server(EmpresaService(),
        servidor)
    dados_pb2_grpc.add_EstagioServiceServicer_to_server(EstagioService(),
        servidor)
    servidor.add_insecure_port('[::]:50051')
    await servidor.start()
    await servidor.wait_for_termination()
```

Listagem 7.23: Definição das funções do CRUD referente a entidade Bolsa

Cada método do serviço recebe como parâmetros a mensagem de requisição e o contexto da chamada, e é responsável por processar a solicitação, interagir com o banco de dados e retornar a mensagem de resposta apropriada. No exemplo apresentado, a função *GetBolsa* verifica a existência do recurso solicitado e, caso não seja encontrado, define explicitamente o código de status *NOT_FOUND* no contexto da chamada, retornando uma resposta vazia. Quando o recurso é localizado, seus dados são mapeados para a mensagem *Bolsa* e enviados ao cliente.

Por fim, todos os serviços definidos foram registrados no servidor gRPC, que foi inicializado para escutar requisições na porta *50051*. Esse processo consolida a API gRPC como uma aplicação funcional e pronta para uso em testes comparativos,

mantendo a equivalência estrutural e semântica com as APIs REST e GraphQL desenvolvidas neste estudo.

Em conclusão, este capítulo apresentou uma visão geral dos principais padrões de arquitetura de APIs web, explorando, de forma integrada, seus fundamentos conceituais, estruturas de comunicação e formas de implementação. As APIs REST, GraphQL e gRPC foram discutidas sob uma perspectiva teórica e prática, evidenciando suas características, vantagens e limitações, bem como as diferenças inerentes aos seus modelos de interação entre cliente e servidor. Ao empregar o mesmo domínio de aplicação, uma base de dados comum e regras de negócio equivalentes, foi possível estabelecer um ambiente controlado e metodologicamente consistente para a análise comparativa.

Capítulo 8

Front-end

Marcos Vinícius Andrade de Sousa (UFC)

Evilasio Costa Junior (UFC)

Fischer Ferreira (Unifei)

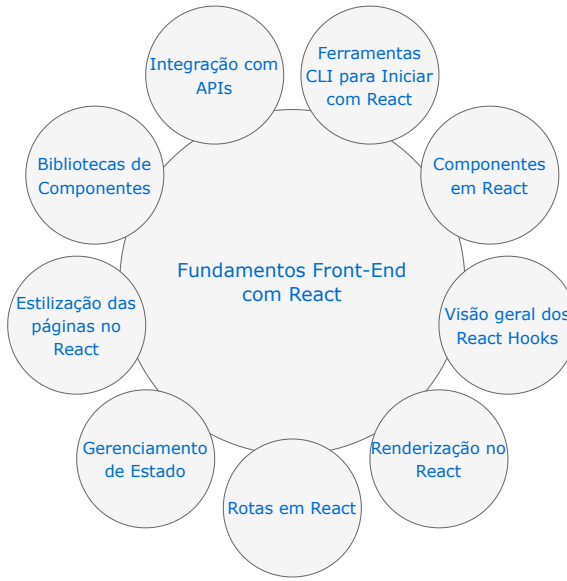


Figura 8.1: Tópicos discutidos neste capítulo

Este capítulo está organizado em duas partes principais. A primeira apresenta o React (Seção 8.2), um dos *frameworks* mais populares para desenvolvimento *front-end*, abordando seus conceitos fundamentais e recursos essenciais. A segunda parte oferece uma visão geral do Angular (Seção 8.17), destacando sua arquitetura e principais funcionalidades.

8.1 Fundamentos do Front-End

Para melhor organização e compreensão do leitor, a Figura 8.1 apresenta os tópicos que serão abordados neste capítulo sobre *React* para a criação do módulo de *front-end* de aplicações web. A Figura 8.1 sintetiza tópicos dos principais conceitos abordados; ela não necessariamente representa uma ordem de aprendizado, mas garante uma visão ampla do tema com base na documentação do React (2025).

Aplicações web e móveis são geralmente divididas em duas áreas principais: *front-end* e *back-end* (LAZUARDY; ANGGRAINI, 2022). No contexto das aplicações web, acessadas por navegadores, o desenvolvimento front-end se concentra na criação e na gestão da interface gráfica e da interação do usuário. Os sites são estruturados principalmente por meio de três tecnologias: HTML, CSS e JavaScript (MDN, 2024a).

Os navegadores modernos utilizam motores de renderização *(rendering engines)* para interpretar e exibir conteúdos estruturados em HTML e estilizados com

CSS (MDN, 2024b). Além disso, esses navegadores possuem motores de execução de *JavaScript*, como o *V8 JavaScript Engine* do Google Chrome [1]. Estes motores interpretam JavaScript segundo os padrões *ECMAScript* e outras especificações tecnológicas, permitindo a execução dinâmica de *scripts* (NODE, 2024). Com essas tecnologias, é possível construir sites estáticos em que o conteúdo permanece constante (W3SCHOOLS, 2024).

O desenvolvimento *front-end* envolve a implementação de elementos visuais e interativos, como botões, e a adaptação da página para diferentes tamanhos de tela (MADURAPPERUMA et al., 2022). Para fornecer conteúdo dinâmico, como atualizações em tempo real, o *front-end* deve se comunicar com o *back-end*. Essa interação é essencial para solicitar e atualizar informações automaticamente (MADURAPPERUMA et al., 2022). O *back-end*, por sua vez, interage com bancos de dados para gerenciar e armazenar dados, como ilustrado na Figura 8.2.

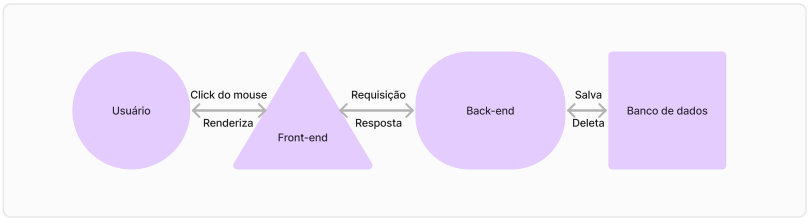


Figura 8.2: Visualização simplificada da comunicação web

O HTML é a base estrutural de qualquer página na web. Ele organiza o conteúdo de forma lógica e semântica por meio de tags, que definem elementos como títulos, parágrafos, listas e links. Essa estrutura hierárquica facilita a interpretação dos dados tanto por navegadores quanto por mecanismos de busca, promovendo clareza e acessibilidade. Dominar HTML é indispensável para criar aplicações web robustas e bem estruturadas (W3SCHOOLS, 2025b).

Esse conjunto de elementos que compõe o HTML promove a clareza e a acessibilidade do conteúdo, permitindo que tanto o usuário quanto as ferramentas de indexação interpretem o HTML de uma página web (MOZILLA, 2025c). Dessa forma, dominar o HTML é indispensável para qualquer desenvolvedor que deseje criar aplicações web robustas e bem estruturadas (MOZILLA, 2025c).

O CSS complementa o HTML ao definir a aparência visual da página, controlando aspectos como cores, fontes, espaçamentos e posicionamento dos elementos (MOZILLA, 2025a). Essa linguagem de estilos permite a criação de layouts responsivos e adaptáveis a diferentes tamanhos de tela, proporcionando uma experiência agradável em dispositivos variados (MOZILLA, 2025a). Por meio de regras e seletores, o CSS facilita a manutenção do código, permitindo atualizações e ajus-

[1] Google Chrome disponível em: ⟨https://www.google.com/intl/pt-BR/chrome/⟩

tes ao *design* da página (W3SCHOOLS, 2025a). A modularidade oferecida por essa ferramenta também possibilita a reutilização de estilos em diversos projetos (W3SCHOOLS, 2025a). Assim, o domínio do CSS é essencial para transformar uma estrutura básica em uma interface atrativa e funcional (MOZILLA, 2025a).

O *JavaScript* agrega dinamismo e interatividade às páginas, sendo a ferramenta que transforma conteúdos estáticos em experiências interativas (W3SCHOOLS, 2025d). Com ele, desenvolvedores podem manipular os elementos definidos pelo HTML e estilizados pelo CSS, respondendo às ações dos usuários, como cliques e inserção de dados (MOZILLA, 2025f). Essa linguagem possibilita a criação de funcionalidades, como a validação de formulários, animações e o carregamento assíncrono de informações, contribuindo para uma experiência de uso mais dinâmica (W3SCHOOLS, 2025d). Dessa forma, o JavaScript demonstra ser um componente indispensável para o desenvolvimento de interfaces web (W3SCHOOLS, 2025d).

Com o domínio dessas três tecnologias, sendo o HTML responsável pela estrutura, o CSS pelo estilo e o JavaScript pela interatividade, o desenvolvedor está apto a criar páginas completas e funcionais. Porém, à medida que as aplicações web se tornam mais complexas, surgem desafios relacionados à organização, à manutenção e à reutilização de código. Para lidar com esses desafios, surgiram *frameworks* e bibliotecas que oferecem abordagens mais eficientes para a construção de interfaces. Entre essas ferramentas, o *React* se destaca por seu modelo baseado em componentes, pela facilidade no gerenciamento de estado e pela ampla adoção no mercado, tornando-se uma das escolhas mais populares no desenvolvimento *front-end* moderno.

8.2 Visão geral sobre o *React*

O React é uma biblioteca JavaScript desenvolvida pelo Facebook que se posiciona como a camada de *View* do padrão de arquitetura *Model-View-Controller* (ROBBESTAD, 2016). Nesse padrão, a View representa a interface gráfica, ou seja, a forma como as informações são apresentadas ao usuário (MOZILLA, 2025g). O *React* é amplamente utilizado para a renderização de elementos da interface de usuário, tais como botões, barras de pesquisa, cabeçalhos, campos de texto e rodapés (REACT, 2025). Cada um desses elementos é implementado como um componente modular, o que permite sua reutilização em diferentes páginas ou até mesmo em múltiplas partes de uma mesma página (REACT, 2025).

Inicialmente, o *React* teve suas raízes no XHP[2], uma extensão do PHP[3] que suporta a linguagem de marcação XML. Em 2010, essa extensão permitia que o PHP interpretasse XML, possibilitando a reutilização de componentes de forma semelhante à adotada pelo *React*. Embora o XHP tenha sido concebido para reduzir ataques de *Cross Site Scripting* (XSS), o Facebook enfrentou dificuldades para gerenciar aplicações web dinâmicas que exigiam muitas requisições do usuário

[2]XHP disponível em: ⟨https://github.com/phplang/xhp⟩
[3]PHP disponível em: ⟨https://www.php.net/⟩

(KOPPALA, 2021). Essa limitação evidenciou a necessidade de uma novas soluções como o *BoltJS* e o *FaxJS*, *frameworks* que evoluíram para o *React* (KOPPALA, 2021).

Um dos principais diferenciais do *React* é a sua abordagem baseada em componentes, que são blocos independentes e reutilizáveis responsáveis por renderizar partes específicas da interface. Essa organização modular favorece a manutenção, a escalabilidade e a clareza do código. Para definir a estrutura desses componentes de forma mais intuitiva, o *React* utiliza o JSX, uma extensão de sintaxe que combina JavaScript com elementos semelhantes ao HTML. Essa combinação permite descrever a interface de forma declarativa, tornando o desenvolvimento mais ágil e legível.

8.3 Ferramentas CLI para Iniciar com *React*

As ferramentas de linha de comando (CLI) permitem executar instruções diretamente no terminal, automatizando processos e configurando rapidamente ambientes de desenvolvimento (AMAZON, 2025). Elas permitem a automação de processos e a configuração rápida de ambientes para o desenvolvimento de aplicações. Essa abordagem agiliza o fluxo de trabalho, dispensando o uso de interfaces gráficas complexas. Com as CLI, é possível criar, configurar e gerenciar projetos de forma padronizada e eficiente.

Antes de iniciar um projeto, é fundamental ter instalado um gerenciador de pacotes, como o `npm`[4], que é responsável por baixar e organizar as dependências necessárias (MOZILLA, 2025h). Esse gerenciador garante que as bibliotecas sejam instaladas nas versões corretas, evitando conflitos no ambiente de desenvolvimento (MOZILLA, 2025h). Garantir que o `npm` esteja instalado e funcionando corretamente é essencial para iniciar o projeto com estabilidade (NPMJS, 2025).

Para iniciar um projeto com *Create React App*[5], execute o comando da Linha 1 da Listagem 8.1, que cria a estrutura inicial do aplicativo (META, 2025d). Em seguida, acesse a pasta do projeto com o comando *cd nome-do-projeto* (Linha 2). Na Linha 3, instale as dependências necessárias com o comando *npm run install*. Por fim, na Linha 4, inicie o servidor de desenvolvimento com o comando *npm run dev*.

```
1 npx create-react-app nome-do-projeto
2 cd nome-do-projeto
3 npm run install
4 npm run dev
```

Listagem 8.1: Passos para iniciar um projeto com *Create React App*

[4]NPM disponível em: ⟨https://docs.npmjs.com/downloading-and-installing-node-js-and-npm⟩
[5]*Create React App* disponível em: ⟨https://create-react-app.dev/docs/getting-started/⟩

No caso do *Vite*[6], o procedimento é semelhante e oferece uma configuração moderna e eficiente para projetos em *React* (VITE, 2023). Para iniciar o projeto, execute o comando da Linha 1 da Listagem 8.2, que utiliza o template *React* para configurar o ambiente. Em seguida, acesse a pasta do projeto com o comando da Linha 2 (*cd nome-do-projeto*) e, na Linha 3, instale as dependências necessárias usando o comando *npm run install*. Por fim, execute o comando da Linha 4 (*npm run dev*) para iniciar o servidor de desenvolvimento e começar a trabalhar no projeto.

```
1 npm create vite@latest nome-do-projeto -- --template react
2 cd nome-do-projeto
3 npm run install
4 npm run dev
```

Listagem 8.2: Passos para iniciar um projeto com Vite

8.4 Componentes em *React*

Os componentes em *React* constituem a base para a construção de interfaces modulares e reutilizáveis (W3SCHOOLS, 2025e). Essa abordagem permite dividir a interface em partes menores, facilitando o desenvolvimento e a manutenção (META, 2025j). Cada componente é responsável por uma funcionalidade específica, promovendo a organização do código e a escalabilidade da aplicação (META, 2025j). A modularidade também favorece o reuso de código, contribuindo para a padronização do projeto. Dessa forma, a estrutura de componentes é fundamental para o desenvolvimento de interfaces modernas (W3SCHOOLS, 2025e).

Os componentes funcionais em *React* são funções JavaScript que retornam elementos de interface (META, 2025j). Essa abordagem apresenta uma sintaxe mais simples e concisa em comparação com os componentes baseados em classes (ROBBESTAD, 2016). Por serem funções, elas facilitam o entendimento e a manutenção do código, promovendo uma escrita mais direta (META, 2025j). Além disso, a utilização de *hooks* permite gerenciar o estado e os efeitos colaterais de forma eficaz (W3SCHOOLS, 2023).

Para fins de ilustração, observe a Listagem 8.3, que apresenta a implementação de um componente funcional denominado *Saudacao*. Inicialmente, na Linha 1, é declarada a classe Component, cujo propósito principal é exibir a mensagem "Olá, mundo!"na interface do usuário. A seguir, a partir da Linha 2, o componente retorna um fragmento de HTML. Essa parte do código é responsável por estruturar e organizar os elementos visuais a serem renderizados na tela. Finalmente, na Linha 9, observa-se a exportação do componente *Saudacao*, o que permite que ele seja utilizado em outras partes do projeto.

[6]Vite disponível em: ⟨https://vite.dev/guide/⟩

```
function Saudacao() {
  return (
    <div>
      <h1>Olá, mundo!</h1>
    </div>
  );
}

export default Saudacao;
```

Listagem 8.3: Exemplo de um componente funcional em React

8.5 Extensão de sintaxe para *JavaScript*, Importações e Exportações de Componentes

A Extensão de sintaxe para JavaScript (JSX) permite a mistura de HTML e JavaScript para definir a interface de forma declarativa (META, 2025i). Essa abordagem torna o código mais intuitivo, facilitando a visualização da estrutura da aplicação (META, 2025i). Ao utilizar JSX, o desenvolvedor pode escrever componentes de forma semelhante à criação de elementos HTML (W3SCHOOLS, 2025g). Essa sintaxe é então transpilada para *JavaScript* puro, garantindo compatibilidade com os navegadores (META, 2025i). Embora a sintaxe lembre HTML, o JSX não é HTML puro: durante o processo de transpilação, ele é convertido em chamadas à função *React.createElement*, gerando código JavaScript compatível com qualquer navegador. Assim, JSX contribui para a clareza e a organização do código (W3SCHOOLS, 2025g).

Em *React*, a modularidade e a reutilização de código são promovidas por meio dos mecanismos de *import* e *export* (W3SCHOOLS, 2025c). Ao exportar um componente, utiliza-se a palavra-chave *export* para torná-lo acessível a outros arquivos do projeto (MOZILLA, 2025d). Existem duas formas principais: a exportação padrão, com *export default*, e a exportação nomeada (MOZILLA, 2025d). Por sua vez, a importação é realizada com o comando *import*, que permite incorporar componentes exportados em diferentes partes do código (MOZILLA, 2025e).

A Listagem 8.4 ilustra o processo de importação e exportação de componentes em um projeto *React*. Nas Linhas 1 e 2, o componente *Cabecalho* é importado para ser utilizado no componente *PaginaPrincipal*, definido na função. Após a utilização do componente, na Linha 13, o componente *PaginaPrincipal* é exportado, permitindo que ele seja utilizado em outras seções da aplicação, como demonstrado na Listagem 8.5. Nesse exemplo, o componente *PaginaPrincipal* renderiza o texto "Bem-vindo à página principal!"com o cabeçalho importado, como mostrado na Linha 5. O uso prático desse processo é ilustrado ao importar *PaginaPrincipal* para o *LayoutDaPagina*, compondo a estrutura da aplicação.

Na Listagem 8.5, o componente *PaginaPrincipal*, previamente criado, é importado na Linha 2 e utilizado na Linha 7. Nesse caso, o componente *PaginaPrincipal* é retornado juntamente com o componente *Rodape*, formando a estrutura principal da página. Essa abordagem modular facilita a construção de aplicações mais robustas e escaláveis, permitindo uma melhor organização do código e a divisão clara de responsabilidades entre os componentes (W3SCHOOLS, 2025c).

```
1  import React from 'react';
2  import Cabecalho from './Cabecalho';
3
4  function PaginaPrincipal() {
5    return (
6      <div>
7        <Cabecalho />
8        <p>Bem-vindo à página principal!</p>
9      </div>
10   );
11 }
12
13 export default PaginaPrincipal;
```

Listagem 8.4: Exemplo de importação e exportação de componentes em React

```
1  import React from 'react';
2  import PaginaPrincipal from './PaginaPrincipal';
3  import Rodape from './Rodape';
4
5  function LayoutDaPagina() {
6    return (
7      <main>
8        <PaginaPrincipal />
9        <Rodape />
10     </main>
11   );
12 }
13
14 export default LayoutDaPagina;
```

Listagem 8.5: Exemplo de uso do componente da página principal

8.6 Props e State

Em *React*, o gerenciamento de dados ocorre por meio de *props* e *state*. As *props* (propriedades) permitem transmitir informações de um componente pai para um

componente filho, sendo imutáveis no componente que as recebe. Já o *state* gerencia dados internos e mutáveis do componente, permitindo que sejam alterados ao longo do tempo e provocando a re-renderização da interface. O exemplo da Listagem 8.6 apresenta um componente Contador, que exibe um valor inicial recebido via props e o armazena no state usando o hook useState. Esse hook retorna um array com o valor atual e uma função para atualizá-lo. Cada clique no botão incrementa o contador em 1.

Na Listagem 8.6, *props* são definidas na Linha 3 e *state* na Linha 4. O exemplo apresenta um componente chamado *Contador*, que exibe um número e um botão. O propósito deste componente é incrementar o número exibido na Linha 8 toda vez que o botão, localizado na Linha 9, for clicado. O valor inicial do contador é passado como *props*, definido pelo componente pai ao instanciar o componente. O valor atual do contador é gerenciado pelo *state*, em que *contador* armazena o valor exibido no HTML e *setContador* é a função responsável por atualizá-lo, incrementando-o em 1 a cada clique no botão.

```
1  import React, { useState } from 'react';
2
3  function Contador({ valorInicial }) {
4    const [contador, setContador] = useState(valorInicial);
5
6    return (
7      <div>
8        <p>Contador: {contador}</p>
9        <button onClick={() => setContador(contador + 1)}>
10         Incrementar
11       </button>
12     </div>
13   );
14 }
15
16 export default Contador;
```

Listagem 8.6: Exemplo de uso de *props* e *state* em um componente

O gerenciamento de dados em *React* é realizado por meio de *props* e *state* (META, 2025a). As *props* são utilizadas para transmitir informações entre componentes, enquanto o *state* gerencia os dados internos de um componente, permitindo que esses dados sejam modificados ao longo do tempo (META, 2025a). Essa distinção é fundamental para a criação de interfaces dinâmicas que se adaptam às interações do usuário (META, 2025f). A combinação de *props* e *state* não só facilita a comunicação entre os componentes, mas também permite a atualização em tempo real das informações exibidas (META, 2025a).

8.7 Renderização Condicional

O exemplo na Listagem 8.7 ilustra o conceito de renderização condicional em *React*. Nele, um componente chamado *SaudacaoCondicional* é definido na Linha 3 e recebe uma *props* chamada *estaLogado*, que determina se o usuário está autenticado. As chaves {} são usadas para permitir inserir expressões *JavaScript*. Dentro dessa lógica condicional, a *props estaLogado* decide o conteúdo renderizado: se o usuário estiver logado, a Linha 7 será exibida com a mensagem "Bem-vindo, usuário!"; caso contrário, será exibida a mensagem "Por favor, faça login.". Dessa forma, o componente oferece uma resposta dinâmica com base no estado de autenticação do usuário.

A renderização condicional é uma técnica que permite exibir elementos de forma dinâmica, com base em condições específicas, como o estado de um componente ou as *props* recebidas (META, 2025c). Por exemplo, utilizando operadores condicionais como `if`, `ternary` ou `&&`, é possível definir quais elementos serão renderizados em diferentes situações. Por exemplo, se a *props* `estaLogado` for verdadeira, pode-se exibir uma mensagem de boas-vindas ao usuário; caso contrário, exibe-se um convite para fazer login. Essa abordagem permite alterar a interface da aplicação de acordo com a interação do usuário, tornando a experiência mais personalizada e responsiva (W3SCHOOLS, 2025f). A flexibilidade da renderização condicional é essencial para criar interfaces interativas e dinâmicas (META, 2025c).

```
1  import React from 'react';
2
3  function SaudacaoCondicional({ estaLogado }) {
4    return (
5      <div>
6        {estaLogado ? (
7          <h1>Bem-vindo, usuário!</h1>
8        ) : (
9          <h1>Por favor, faça login.</h1>
10       )}
11     </div>
12   );
13 }
14
15 export default SaudacaoCondicional;
```

Listagem 8.7: Exemplo de renderização condicional em React

8.8 *Composition vs Inheritance*

No React, *composition* (composição) e *inheritance* (herança) são abordagens distintas para criar e reutilizar componentes. A composição é a técnica recomendada,

pois promove modularidade, flexibilidade e fácil manutenção (META, 2025b), enquanto a herança é menos comum e pode gerar estruturas mais rígidas e difíceis de escalar. A abordagem de *composition* permite reutilizar componentes, combinando-os para criar novas *interfaces* (META, 2025b). A *composition* permite encapsular funcionalidades em componentes menores e independentes, evitando estruturas rígidas e complexas (META, 2025b).

A Listagem 8.8 ilustra a prática de *composition* em React com dois componentes. O primeiro, denominado *BordaEstilizada*, é definido na Linha 3 e recebe uma *props* chamada *children*. O segundo componente, *DialogoBoasVindas*, é definido na Linha 11 e utiliza o componente *BordaEstilizada* na Linha 13. O conteúdo dentro de *DialogoBoasVindas*, especificamente nas Linhas 14 e 15, é passado como valor para a *props children*. O componente *BordaEstilizada* então recebe este conteúdo como *props* e o renderiza dentro de um `div` com a classe *borda-estilizada* na Linha 6. O resultado final exibido na interface do usuário será a renderização dos textos "Bem-vindo"e "Obrigado por visitar nossa aplicação", com a estilização fictícia da borda aplicada ao conteúdo.

```
1  import React from 'react';
2
3  function BordaEstilizada({ children }) {
4    return (
5      <div className="borda-estilizada">
6        {children}
7      </div>
8    );
9  }
10
11 function DialogoBoasVindas() {
12   return (
13     <BordaEstilizada>
14       <h1>Bem-vindo</h1>
15       <p>Obrigado por visitar nossa aplicação.</p>
16     </BordaEstilizada>
17   );
18 }
19
20 export default DialogoBoasVindas;
```

Listagem 8.8: Exemplo de *composition* em React

Por outro lado, a abordagem de *inheritance* pode ser utilizada para estender o comportamento de um componente base, permitindo a herança de propriedades e métodos (META, 2025b). No entanto, essa técnica pode resultar em código mais rígido, dificultando a manutenção e a reutilização em aplicações maiores (META, 2025b).

A Listagem 8.9 apresenta um exemplo de *inheritance* (herança) em *React*. Neste exemplo, dois componentes são definidos como classes nas Linhas 3 e 12. O primeiro, *Botao*, estende a classe *Component* do *React* (importada na Linha 1) e define o método *handleClick*, que exibe a mensagem "Clicado"no console quando o botão é clicado. Em seguida, o componente *BotaoPrimario* herda a classe *Botao*, preservando o método *handleClick*, mas sobrescreve o método *render* para aplicar uma estilização diferenciada, adicionando a classe *primario* ao botão (linha 14) e alterando seu rótulo para "Clique aqui". O resultado na interface do usuário é um botão que, ao ser clicado, imprime "Clicado"no console, e esse botão final é estilizado com a classe *primario*.

```
1  import React, { Component } from 'react';
2
3  class Botao extends Component {
4    handleClick() {
5      console.log("Clicado");
6    }
7    render() {
8      return <button onClick={this.handleClick.bind(this)}>
9      Clique</button>;
10   }
11 }
12
13 class BotaoPrimario extends Botao {
14   render() {
15     return <button className="primario"
16     onClick={this.handleClick.bind(this)}>Clique aqui</button>;
17   }
18 }
19
20 export default BotaoPrimario;
```

Listagem 8.9: Exemplo de *inheritance* em React

8.9 Visão geral dos *React Hooks*

Esta seção apresenta uma visão geral dos *Hooks* integrados no *React*. Introduzidos na versão 16.8, em 2018, os *Hooks* possibilitaram que funcionalidades antes exclusivas dos componentes baseados em classes fossem utilizadas em componentes funcionais (BUGL, 2019). Dessa forma, os *Hooks* facilitam o acesso a conceitos fundamentais do *React*, como *props*, *contexts*, *refs*, estados e o ciclo de vida dos componentes (META, 2023). Essa inovação promove uma abordagem mais direta e intuitiva no desenvolvimento de interfaces (BUGL, 2019).

A Listagem 8.10 apresenta um exemplo prático que renderiza, na interface do usuário, um e-mail e, após 10 segundos, o substitui. Na Linha 4, o *Hook* de estado, *useState*, é empregado para armazenar dados que precisam ser monitorados pela aplicação (W3SCHOOLS, 2023). Em seguida, na Linha 6, o *Hook* de efeito, *useEffect*, é utilizado para gerenciar efeitos colaterais, como chamadas a uma API, atualizações do *Virtual Document Object Model* (VirtualDOM) e controle de temporizadores (W3SCHOOLS, 2023). Esse exemplo demonstra como os *Hooks* tornam o código mais organizado e legível (META, 2023). Dessa forma, o uso combinado desses recursos reflete uma abordagem prática e modular.

```
1  import { useState, useEffect } from "react";
2
3  export const Hooks = () => {
4    const [email, setEmail] = useState("sample@invalid.com");
5
6    useEffect(() => {
7      setTimeout(() => {
8        setEmail("another_sample@invalid.com");
9      }, 10000);
10   });
11
12   return (
13     <div>
14       <p>{email}</p>
15     </div>
16   );
17 };
```

Listagem 8.10: Exemplificação dos *Hooks* *useState* e *useEffect*

O *useState* é um dos *Hooks* mais fundamentais, permitindo que componentes funcionais possuam um estado interno, o que antes era exclusivo de componentes de classe. Esse *Hooks* retorna um par de valores: o valor atual do estado e uma função para atualizá-lo. Por exemplo, em um componente de contador, *useState* pode ser utilizado para armazenar o valor atual do contador e atualizá-lo sempre que o usuário interagir com a interface, mantendo o componente dinâmico e responsivo.

Já o *useEffect* é essencial para lidar com efeitos colaterais, como a atualização do DOM, chamadas assíncronas ou a manipulação de dados que não envolvem diretamente a renderização do componente. Ele executa funções após o componente ser renderizado, permitindo que o desenvolvedor realize operações como buscar dados de uma API ou configurar *timers*, sem bloquear o ciclo de vida do componente. Usado adequadamente, *useEffect* melhora a eficiência e organiza o fluxo de execução, sendo crucial para criar interfaces mais interativas e funcionais.

Além disso, os *Hooks* foram criados para solucionar desafios específicos enfrentados durante o desenvolvimento com *React* (META, 2023). A primeira motivação

reside na necessidade de compartilhar estados entre componentes sem alterar a hierarquia de componentes (W3SCHOOLS, 2023). Em seguida, buscou-se resolver a dispersão e o acoplamento inadequado da lógica de estado, que muitas vezes separa funções inter-relacionadas e une lógicas independentes (W3SCHOOLS, 2023). Por fim, os *Hooks* visam facilitar a compreensão do código, especialmente em comparação com os métodos baseados em classes (META, 2023). Essas motivações evidenciam a importância dos *Hooks* na evolução das práticas de desenvolvimento.

Além disso, para garantir o correto funcionamento dos *Hooks*, três regras fundamentais devem ser seguidas: eles devem ser chamados apenas dentro de componentes funcionais, sempre no início desses componentes (ou seja, antes do retorno), e não podem ser invocados de forma condicional (W3SCHOOLS, 2023). O descumprimento dessas diretrizes pode ocasionar comportamentos inesperados e dificultar a manutenção do código.

O *React* disponibiliza diversos *Hooks* nativos e ainda permite a criação de *Hooks* customizados, bem como a integração com *Hooks* oferecidos por bibliotecas externas, como *Apollo Client* [7], *React Redux* [8] e *React Query* [9]. Essa flexibilidade permite encapsular funcionalidades específicas e reutilizá-las em diferentes partes da aplicação. Entre os *Hooks* nativos, podem ser destacados: os *State Hooks* (*useState* e *useReducer*), os *Context Hooks* (*useContext*), os *Ref Hooks* (*useRef* e *useImperativeHandle*), os *Effect Hooks* (*useEffect*, *useLayoutEffect* e *useInsertionEffect*) e os *Performance Hooks* (*useMemo*, *useCallback*, *useTransition* e *useDeferredValue*) (W3SCHOOLS, 2023).

8.10 Renderização no *React*

Para compreender o processo de renderização no *React*, é fundamental entender o que é o DOM. O DOM é uma interface de programação que representa a estrutura de documentos HTML ou XML como uma árvore de objetos, onde cada elemento da página corresponde a um nó (MOZILLA, 2025b). Essa representação permite que linguagens de script, como *JavaScript*, acessem e manipulem dinamicamente os elementos e a organização do conteúdo (MDN, 2024a). Contudo, a manipulação direta do DOM pode ser custosa, pois cada alteração pode desencadear reprocessamentos que impactam o desempenho (META, 2025h).

Para mitigar tais desafios, o *React* adota o conceito de VirtualDOM (META, 2025h). Essa abordagem consiste em manter uma versão simplificada e em memória do DOM real, ou seja, uma representação ideal sincronizada com o DOM real por meio da biblioteca *ReactDOM* (META, 2025h). Ao ocorrer uma mudança no estado de um componente, o *React* atualiza primeiro o VirtualDOM, permitindo um processo de reconciliação que identifica e aplica apenas as alterações necessárias

[7] *Apollo Client* disponível em: ⟨https://www.apollographql.com/docs/⟩
[8] *React Redux* disponível em:⟨https://react-redux.js.org/⟩
[9] *React Query* disponível em: ⟨https://tanstack.com/query/v3/docs/react/overview⟩

(ROBBESTAD, 2016). Esse mecanismo, conhecido como *diffing*, minimiza as operações onerosas e otimiza a renderização (META, 2025h).

O *React* não interage diretamente com o DOM real; todas as alterações ocorrem no VirtualDOM. Quando um componente sofre uma alteração de estado, o *framework* compara a nova versão com a anterior, aplicando apenas as mudanças imprescindíveis (ROBBESTAD, 2016). Essa estratégia permite atualizações isoladas, em que alterações em determinados elementos não interferem nos demais (ROBBESTAD, 2016). Dessa forma, a renderização torna-se mais eficiente, o que contribui para o desempenho geral da aplicação.

Adicionalmente, o *React* orienta que o desenvolvedor não acesse nem modifique diretamente os elementos da interface por meio do DOM (META, 2025e). Em vez disso, recomenda-se o uso de *refs* para associar um nó a uma referência, permitindo o acesso indireto (META, 2025e). Essa abordagem, realizada por meio do *hook useRef*, garante que as manipulações ocorram por meio da atualização do VirtualDOM, mantendo a consistência da interface (META, 2025e).

A Listagem 8.11 renderiza na interface do usuário um campo para digitar, e um botão que quando clicado coloca o foco do navegador no campo de texto. Esse exemplo mostra como usar o *hook useRef* para acessar um elemento *input*. Primeiro, o *useRef* é instanciado na Linha 4. Em seguida, ele é atribuído ao elemento na Linha 12. Por fim, o acesso ao elemento ocorre na Linha 6 por meio do método correspondente.

```
1  import { useRef } from 'react';
2
3  function MeuComponente() {
4    const inputRef = useRef(null);
5
6    function handleClick() {
7      inputRef.current.focus();
8    }
9
10   return (
11     <>
12       <input ref={inputRef} type="text" />
13       <button onClick={handleClick}>focus</button>
14     </>
15   );
16 }
17
18 export default MeuComponente;
```

Listagem 8.11: Acessando um elemento com *useRef*

8.11 Rotas em *React*

A navegação em aplicações *React* é essencial para construir interfaces interativas e dinâmicas (META, 2023). O conceito de *routing* permite que o usuário seja direcionado para diferentes páginas sem o recarregamento completo da aplicação (ROUTER, 2023). Em sites tradicionais, cada mudança de rota implica a recarga de todos os recursos, o que pode comprometer a experiência do usuário (ROUTER, 2023). Em contrapartida, aplicações SPA (*Single Page Application*) atualizam apenas os componentes necessários, proporcionando uma navegação mais fluida e eficiente. Dessa forma, o uso de bibliotecas especializadas torna o processo de transição entre páginas mais ágil e responsivo (MOZILLA, 2025i).

Para implementar o *routing* no *React*, utiliza-se a biblioteca *React Router Dom*, que possibilita a atualização da URL e o carregamento de novos componentes sem a necessidade de recarregar toda a página (ROUTER, 2023). A instalação dessa ferramenta é simples e pode ser realizada via `npm`, conforme demonstrado na Listagem 8.12 (ROUTER, 2023). Essa etapa é fundamental para que a aplicação possa gerenciar suas rotas de forma eficaz e escalável, integrando componentes responsáveis pela definição e exibição de conteúdo.

```
1 npm i react-router-dom
```

Listagem 8.12: Comando para instalar o React Router Dom

A Listagem 8.13 apresenta a estrutura básica de um componente *Router*, utilizando os componentes *BrowserRouter*, *Routes*, *Route* e *Outlet*. O *BrowserRouter* funciona como o contêiner principal, englobando todas as rotas da aplicação, enquanto o *Outlet* é responsável por injetar o conteúdo específico de cada rota em um layout comum, que pode incluir elementos como *header* e *footer*. Essa organização modular facilita a manutenção e a escalabilidade da aplicação, garantindo que o layout seja consistente em todas as páginas. Além disso, essa estrutura torna a navegação mais eficiente e a criação de novas rotas mais simples (ROUTER, 2023).

```
import { BrowserRouter, Routes, Route, Outlet }
from "react-router-dom";
import React from "react";

export const Router = () => {
  return (
    <BrowserRouter>
      <Routes>
        <Route
          path="/"
          element={
            <main>
              <header>header</header>
              <Outlet />
              <footer>footer</footer>
            </main>
          }
        >
          <Route path="/" element={<h1>menu principal</h1>} />
          <Route path="/perfil"
          element={<h1>perfil do usuário</h1>} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
};
```

Listagem 8.13: Componente Router Inicial

A Listagem 8.14 demonstra a integração dos componentes *Link* e *Route* para a navegação entre páginas. Nesse caso, a rota principal exibe um menu com um link que direciona para a rota de perfil, evitando o recarregamento completo da página. A utilização do componente *Link* assegura transições suaves entre as páginas, reforçando a eficiência do *routing* do lado do cliente. Esse exemplo fornece ao desenvolvedor todos os elementos necessários para implementar uma navegação robusta em aplicações *React*.

```
import { BrowserRouter, Routes, Route, Outlet, Link }
from "react-router-dom";

export const Router = () => (
  <BrowserRouter>
    <Routes>
      <Route
        path="/"
        element={
          <main>
            <header>header</header>
            <Outlet />
            <footer>footer</footer>
          </main>
        }
      >
        <Route
          path="/"
          element={
            <>
              <h1>menu principal</h1>
              <Link to="/perfil">Perfil</Link>
            </>
          }
        />
        <Route
          path="/perfil"
          element={
            <>
              <h1>perfil do usuário</h1>
              <Link to="/">Voltar</Link>
            </>
          }
        />
      </Route>
    </Routes>
  </BrowserRouter>
);
```

Listagem 8.14: Componente Router Completo

8.12 Gerenciamento de Estado

O gerenciamento de estado em aplicações *React* é fundamental para o desenvolvimento de interfaces dinâmicas e escaláveis (META, 2025g). A utilização do *createContext* possibilita o compartilhamento de dados sem a necessidade de repassar propriedades manualmente (REACT, 2025). Essa técnica simplifica a comunicação interna entre componentes e promove uma estrutura de código mais limpa (DODDS, 2025). Adicionalmente, integra-se facilmente a bibliotecas como

Redux [10] e MobX [11] para cenários mais complexos (REDUX, 2025). Este tutorial apresenta exemplos práticos sobre o tema.

Antes de explorar os exemplos, observe que a Listagem 8.15 implementa o contexto para gerenciar o estado que define o tema de cores de uma página web. Já a Listagem 8.16 demonstra a utilização desse contexto na aplicação, exibindo ao usuário o tema atual e um botão para alternar entre os temas. Essas referências facilitam a compreensão da estrutura modular adotada, destacando como o uso de contexto centraliza o gerenciamento de estado e elimina o repasse desnecessário de propriedades entre os componentes. Como resultado, a aplicação torna-se mais eficiente, escalável e de fácil manutenção.

A Listagem 8.15 demonstra como implementar um contexto para gerenciar a alternância entre os modos de tema *light* e *dark* em uma aplicação *React*. O código começa por criar um contexto, o *ThemeContext*, utilizando o *createContext* na Linha 3. Esse contexto armazena e compartilha o valor do tema com outros componentes. Na Linha 5, é configurado um *Provider* para envolver os componentes consumidores do contexto, permitindo que qualquer componente dentro desse *Provider* acesse e altere o tema. O estado do tema é gerenciado por meio do *hook useState*, que define inicialmente o tema como *light*. Uma função de alternância, definida na Linha 8, é responsável por modificar o valor do estado do tema, alternando entre light e *dark* conforme a interação do usuário. No componente *ThemedComponent*, o *hook useContext* é utilizado para acessar o valor atual do tema e a função de alternância, permitindo que o usuário visualize o tema atual e altere-o ao clicar no botão. Esse exemplo modulariza a troca de temas, facilitando a manutenção do estado global e promovendo uma estrutura de código organizada.

[10] Redux disponível em: ⟨https://redux.js.org/⟩
[11] MobX disponível em: ⟨https://mobx.js.org/⟩

```
1  import React, { createContext, useState, useContext } from 'react';
2
3  const ThemeContext = createContext();
4
5  function ThemeProvider({ children }) {
6    const [theme, setTheme] = useState('light');
7
8    const toggleTheme = () => {
9      setTheme(prevTheme => prevTheme === 'light' ? 'dark' : 'light');
10   };
11
12   return (
13     <ThemeContext.Provider value={{ theme, toggleTheme }}>
14       {children}
15     </ThemeContext.Provider>
16   );
17 }
18
19 function ThemedComponent() {
20   const { theme, toggleTheme } = useContext(ThemeContext);
21
22   return (
23     <div>
24       <p>The current theme is {theme}</p>
25       <button onClick={toggleTheme}>Toggle Theme</button>
26     </div>
27   );
28 }
29
30 export { ThemeProvider, ThemedComponent };
```

Listagem 8.15: Exemplo de gerenciamento de estado com *createContext*

A Listagem 8.16 demonstra o uso do contexto em um componente principal denominado *App*. Neste exemplo, o *ThemeProvider* (linha 6) envolve o componente *ThemedComponent* (linha 7), permitindo que este acesse e consuma o estado global definido no contexto. Essa estrutura ajuda a centralizar o gerenciamento do estado e elimina a necessidade de passar as propriedades manualmente para cada componente. O exemplo reforça a importância de uma arquitetura modular e escalável, destacando como o padrão de *composition* pode ser aplicado em conjunto com a *Context API* do *React*, promovendo uma abordagem mais limpa, organizada e fácil de manter.

Em resumo, o uso de *createContext* em conjunto com os *hooks* do *React* oferece uma solução leve e eficaz para o gerenciamento de estado (DODDS, 2025). Ao centralizar os dados, elimina-se a necessidade de repassar propriedades entre múltiplos componentes, simplificando o fluxo de dados (REACT, 2025). Essa abordagem resulta em um código mais organizado e de fácil manutenção (REACT, 2025). Embora bibliotecas externas possam fornecer soluções mais robustas, o padrão nativo do *React* é adequado para muitas situações, oferecendo uma alternativa prática e eficiente para a maioria dos cenários (TOOLKIT, 2025).

```
import React from 'react';
import { ThemeProvider, ThemedComponent } from './ThemeContext';

function App() {
  return (
    <ThemeProvider>
      <ThemedComponent />
    </ThemeProvider>
  );
}

export default App;
```

Listagem 8.16: Exemplo de utilização do contexto no componente *App*

8.13 Estilização das páginas no *React*

O *React* não impõe uma única abordagem para a estilização de páginas HTML (META, 2023). Embora seja possível aplicar técnicas tradicionais de CSS em páginas estáticas, o *React* apresenta particularidades que devem ser observadas. Por exemplo, o atributo *class* do HTML é substituído por *className* no *React* (REACT, 2025). Embora essa mudança possa parecer simples, ela é fundamental para evitar problemas durante o desenvolvimento. Compreender essas diferenças é crucial para aplicar os estilos de maneira adequada e sem conflitos.

A estilização de páginas *React* pode ser realizada de três formas principais: *inline styling*, *CSS stylesheet* e *CSS Modules* (W3SCHOOLS, 2025h). No primeiro método, os estilos são aplicados diretamente no atributo *style* de um componente (W3SCHOOLS, 2025h). No exemplo da Listagem 8.17, o componente cria um objeto *JavaScript* para configurar as propriedades de estilo, e nomes compostos, como *font-family*, devem ser escritos com a convenção *camelCase* (W3SCHOOLS, 2025h). Essa abordagem é ideal para ajustes rápidos e testes de estilos, pois permite aplicar regras diretamente no componente, de forma prática e eficiente (W3SCHOOLS, 2025h).

```
const HelloWorld = () => {
  return (
    <div style={{ border: "solid" }}>
      <h1 style={{ color: "red", fontFamily: "fantasy",
      textAlign: "center" }}>Hello World!</h1>
    </div>
  );
};

export default HelloWorld;
```

Listagem 8.17: Componente HelloWorld com estilização *inline*

A segunda abordagem envolve o uso de uma *CSS stylesheet*, na qual os estilos são definidos em um arquivo *.css* separado e importados diretamente no componente. Na Listagem 8.18, observa-se a importação da folha de estilo; na Listagem 8.19, é possível ver o conteúdo do arquivo de estilo (W3SCHOOLS, 2025h). É importante destacar que, por se tratar de uma estilização global, os estilos definidos afetam todos os elementos que compartilham o mesmo seletor, como a tag *h1*. Essa característica exige um planejamento cuidadoso da arquitetura de estilos, especialmente em aplicações maiores, para evitar efeitos indesejados em elementos que não deveriam ser estilizados de forma semelhante.

```
import "./HelloWorld.style.css";

const HelloWorld = () => {
  return (
    <div>
      <h1>Hello World!</h1>
    </div>
  );
};

export default HelloWorld;
```

Listagem 8.18: Componente HelloWorld importando uma folha de estilo

```
div {
  border: solid;
}

h1 {
  color: red;
  font-family: fantasy;
  text-align: center;
}
```

Listagem 8.19: Arquivo *HelloWorld.style.css*

Por fim, a terceira alternativa é o uso de *CSS Modules*, que permite aplicar estilos de forma encapsulada (W3SCHOOLS, 2025h). Inicialmente, cria-se um arquivo com extensão *.module.css*, onde são definidos módulos de estilo, como exemplificado na Listagem 8.20. Em seguida, esses módulos são importados para o componente, usualmente por meio de um objeto denominado *styles*, e aplicados às tags por meio do atributo *className*, conforme visto na Listagem 8.21. Essa abordagem minimiza conflitos e facilita a manutenção dos estilos.

```
.bordaSolida {
  border: solid;
}

.tituloVermelho {
  color: red;
  font-family: fantasy;
  text-align: center;
}
```

Listagem 8.20: Arquivo *HelloWorld.module.css*

```
import styles from "./HelloWorld.module.css";

const HelloWorld = () => {
  return (
    <div className={styles.bordaSolida}>
      <h1 className={styles.tituloVermelho}>Hello World!</h1>
    </div>
  );
};

export default HelloWorld;
```

Listagem 8.21: Componente HelloWorld com estilização por módulos

A Tabela 8.1 compara as três principais abordagens de estilização no React (inline styling, CSS stylesheet e CSS Modules) destacando suas vantagens, desvantagens e os contextos mais indicados para aplicação. Essa organização permite ao desenvolvedor avaliar rapidamente qual técnica melhor atende às necessidades do projeto, considerando fatores como facilidade de manutenção, risco de conflitos de estilo, escalabilidade e suporte a recursos avançados do CSS. A análise comparativa também evidencia que não há uma solução única para todos os casos, sendo comum a combinação dessas abordagens em um mesmo projeto para aproveitar o melhor de cada uma.

Tabela 8.1: Comparação das abordagens de estilização no *React*

Abordagem	Vantagens	Desvantagens	Casos de uso recomendados
Inline styling	Fácil de aplicar; permite uso de variáveis e lógica condicional diretamente no estilo; dispensa arquivos externos.	Sintaxe mais verbosa; dificulta manutenção em estilos grandes; não suporta pseudoclasses ou pseudoelementos.	Prototipagem rápida; estilos específicos e dinâmicos; ajustes pontuais em elementos.
CSS stylesheet	Separação clara entre HTML/JSX e CSS; suporte total ao CSS, incluindo pseudoclasses e media queries; fácil integração com ferramentas e bibliotecas.	Estilos aplicados globalmente; risco de conflitos de classes; requer planejamento para evitar sobrescritas indesejadas.	Projetos pequenos ou médios; aplicações sem grande complexidade de estilos.
CSS Modules	Escopo restrito ao componente; evita conflitos de classe; facilita a manutenção e a escalabilidade.	Sintaxe ligeiramente mais complexa; requer configuração em alguns projetos.	Projetos médios e grandes; aplicações que exigem organização e reaproveitamento de componentes.

Adaptado de React Documentation (REACT, 2025), W3Schools (W3SCHOOLS, 2025h).

8.14 Bibliotecas de Componentes

Bibliotecas e *frameworks* de componentes prontos, como Material-UI [12] e Shadcn [13], que aceleram o desenvolvimento de interfaces. Estas ferramentas facilitam a criação de layouts modernos e a padronização visual dos projetos, promovendo a reutilização de código e a agilidade na implementação (COYIER, 2008). O uso de soluções consolidadas contribui para a redução de erros e a melhoria da manutenção. Ademais, a documentação e a comunidade ativa de cada biblioteca oferecem suporte robusto aos desenvolvedores. Dessa forma, evidencia-se a importância de integrar tais recursos à engenharia de *software*.

A Listagem 8.22 apresenta o comando utilizado para instalar o *Material-UI*, uma biblioteca de componentes *React* de código aberto que implementa o *Material Design* do Google. De forma análoga, a Listagem 8.23 ilustra o comando de instalação do Shadcn. Ressalta-se que, conforme o *framework* ou a CLI adotados na inicialização do projeto, a configuração pode variar; recomenda-se consultar a documentação oficial para garantir a correta implementação.

[12] Material-UI disponível em: ⟨https://mui.com/material-ui/⟩
[13] Shadcn disponível em: ⟨https://ui.shadcn.com/⟩

```
npm install @mui/material @emotion/react @emotion/styled
```

Listagem 8.22: Instalação do *Material-UI*

```
npx shadcn init
```

Listagem 8.23: Instalação do *Shadcn*

O próximo exemplo apresenta a importação e o uso de um componente do *Material-UI* em um projeto *React*. A Listagem 8.24 ilustra como o componente *Button* pode ser importado e incorporado em uma função que define um componente funcional. Esse exemplo vai renderizar na interface do usuário um botão com a estilização definida na biblioteca *Material-UI*, com a variação de botão *contained*, cor *primary* e com o texto "Clique aqui" definido na Linha 8 do componente.

A prática de usar bibliotecas de componentes permite criar interfaces interativas com *design* padronizado, facilitando a manutenção e a escalabilidade do código. Ao utilizar componentes prontos, o desenvolvedor concentra-se na lógica de negócio, sem se preocupar com detalhes de estilização.

```
1  import React from 'react';
2  import Button from '@mui/material/Button';
3
4  function App() {
5    return (
6      <div>
7        <Button variant="contained" color="primary">
8          Clique aqui
9        </Button>
10     </div>
11   );
12 }
13
14 export default App;
```

Listagem 8.24: Uso do componente *Button* do Material-UI

Por fim, o exemplo a seguir demonstra como importar e utilizar um componente do *Shadcn* em um ambiente *React*. A Listagem 8.25 apresenta o componente *Card*, incorporado a uma função que renderiza uma estrutura com cabeçalho e corpo. Esse exemplo evidencia a modularização da interface, permitindo que o componente seja reutilizado em diferentes partes do projeto. Essa abordagem promove um código mais organizado e limpo, além de destacar a eficiência no desenvolvimento de interfaces ao aproveitar componentes prontos.

```
import React from 'react';
import { Card } from 'shadcn';

function DashboardCard() {
  return (
    <Card>
      <Card.Header>
        <h2>Dashboard</h2>
      </Card.Header>
      <Card.Body>
        Conteúdo do cartão.
      </Card.Body>
    </Card>
  );
}

export default DashboardCard;
```

Listagem 8.25: Uso do componente *Card* do Shadcn

8.15 Integração com APIs

Inicialmente, torna-se necessário compreender os fundamentos que permitem a comunicação entre uma *Representational State Transfer Application Programming Interface* (API REST) e um *front-end* desenvolvido em *React* (REDHAT, 2023). Para isso, é imprescindível definir o conceito de API REST e esclarecer o funcionamento do *AJAX* (REDHAT, 2023). Esses conceitos formam a base para a integração entre aplicações e possibilitam a criação de sistemas robustos e escaláveis (REDHAT, 2023).

Uma API REST é uma interface de programação que segue um conjunto de restrições arquiteturais, como a arquitetura cliente/servidor, requisições HTTP e comunicação sem estado (REDHAT, 2023). Além disso, destaca-se a importância de uma interface uniforme, a existência de um sistema de camadas e hierarquias, bem como o uso de cache para otimização. Esses atributos padronizados facilitam a interação entre diferentes sistemas, promovendo uma comunicação eficiente e estruturada.

O termo *AJAX* significa *Asynchronous JavaScript e XML* (MOZILLA, 2023a). Embora o *XML* tenha sido o formato predominante inicialmente, atualmente o JSON é amplamente utilizado por ser mais leve e simples. Essa abordagem assíncrona permite que dados sejam atualizados na interface sem a necessidade de recarregar toda a página, contribuindo para uma experiência de usuário mais fluida e interativa.

Para simplificar o desenvolvimento e aprimorar a compreensão, opta-se pelo uso do cliente *AXIOS* [14]. Essa biblioteca *JavaScript* permite o envio de requisições HTTP, o tratamento de promessas e a manipulação de recursos de *AJAX*, convertendo automaticamente as respostas das API para JSON (AXIOS, 2023). Devido a essas características, o *AXIOS* revela-se uma escolha interessante para uso tanto no *front-end* quanto no *back-end* (KHOLMATOV, 2023).

Como exemplo prático, a Listagem 8.26 apresenta o comando necessário para instalar a biblioteca AXIOS em um projeto *React*. Esse comando deve ser executado no terminal, utilizando o gerenciador de pacotes escolhido pelo desenvolvedor, o que facilita a integração do *AXIOS* no projeto.

```
1 npm install axios
```

Listagem 8.26: Comandos para instalar o *AXIOS*

Ademais, a Listagem 8.27 apresenta a implementação de uma função que obtém os dados de um usuário por meio da API do GitHub. No código, o *AXIOS* é configurado para acessar o *endpoint* de usuários na Linha 3. Em seguida, são realizadas múltiplas requisições (da Linha 9 até a Linha 13) para capturar informações relevantes, como repositórios, seguidores e outros dados, que são armazenados em um estado, conforme definido na Linha 22. Este exemplo ilustra, de forma prática, como integrar o *front-end* a uma API utilizando o *AXIOS*.

[14] AXIOS disponível em: ⟨https://axios-http.com/ptbr/docs/intro⟩

```
import axios from "axios";

const apiGithub = axios.create({
baseURL: "https://api.github.com/users",
});

const fetchData = async (userName) => {
try {
  const result = await apiGithub.get(`/${userName}`);
  const stars = await apiGithub.get(`/${userName}/starred`);
  const repos = await apiGithub.get(`/${userName}/repos`);
  const followers = await apiGithub.get(`/${userName}/followers`);
  const following = await apiGithub.get(`/${userName}/following`);
  const userData = {
    ...result.data,
    stars: stars.data.length,
    repos_list: repos.data,
    stars_list: stars.data,
    followers_list: followers.data,
    following_list: following.data,
  };
  setData(userData);
} catch (e) {
  console.log(e);
  setData({});
}
};
```

Listagem 8.27: Consulta na API do Github com *AXIOS*

8.16 Exemplo prático: O jogo da forca no React

Nesta seção, é apresentado um exemplo prático do jogo da forca implementado com *React* e TypeScript, demonstrando, de forma didática, a criação e a organização dos componentes que compõem a aplicação. Cada componente foi desenvolvido em um arquivo *.tsx* distinto, o que facilita a manutenção e a compreensão da estrutura do projeto.

Para iniciar o projeto, utiliza-se o Vite, uma ferramenta que permite a criação e a execução de projetos *React* com suporte a TypeScript. Os arquivos *.tsx* permite a escrita de JSX com tipagem estática, contribuindo para a detecção antecipada de erros. Os comandos para a criação e a execução do projeto encontram-se na Listagem 8.28. Dessa forma, a configuração inicial já orienta o leitor sobre como estruturar o ambiente de desenvolvimento.

Além disso, o projeto integra o Tailwind CSS para aprimorar a estilização da aplicação, proporcionando um design moderno e responsivo. Inicialmente, as dependências são instaladas e os arquivos de configuração são gerados, conforme demonstrado na Listagem 8.29. Em seguida, o arquivo *tailwind.config.js* é configurado

para que o Tailwind identifique os arquivos do projeto (Listagem 8.30). Por fim, as diretivas do Tailwind são incluídas no arquivo de estilos, conforme mostrado na Listagem 8.31. Estes passos garantem que as classes utilitárias do Tailwind possam ser utilizadas ao longo do projeto, oferecendo flexibilidade adicional na construção da interface.

No arquivo *constants.tsx* (Listagem 8.32), são definidas as constantes essenciais para o funcionamento do jogo, como a lista de palavras (*WORDS*) e o número máximo de erros permitidos (*MAX_MISTAKES*). A constante *WORDS* é definida entre as linhas 1 e 22, contendo a lista de termos que poderão ser utilizados durante a partida. Já a constante *MAX_MISTAKES*, definida na linha 24, especifica o limite de tentativas incorretas. A separação desses dados em um arquivo único facilita a manutenção, permitindo que esses valores sejam facilmente reutilizados em diferentes componentes da aplicação.

```
npm create vite@latest hangman-game --template react-ts
cd hangman-game
npm install
npm run dev
```

Listagem 8.28: Comando para inicializar o projeto React com Vite e TypeScript

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

Listagem 8.29: Instalação das dependências do Tailwind CSS e criação dos arquivos de configuração

```
module.exports = {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Listagem 8.30: Configuração do Tailwind CSS para identificação dos arquivos do projeto

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Listagem 8.31: Inclusão das diretivas do Tailwind CSS no arquivo de estilos

```
export const WORDS = [
  "REACT",
  "JAVASCRIPT",
  "COMPONENT",
  "FUNCTION",
  "HOOK",
  "STATE",
  "EFFECT",
  "CONTEXT",
  "PROPS",
  "VIRTUAL",
  "RENDER",
  "ELEMENT",
  "FRAGMENT",
  "MEMO",
  "REDUCER",
  "CALLBACK",
  "REFERENCE",
  "PORTAL",
  "SUSPENSE",
  "LAZY",
];

export const MAX_MISTAKES = 6;
```

Listagem 8.32: Definição das constantes em *constants.tsx*

O componente principal, *HangmanGame*, foi dividido em duas partes para melhorar a legibilidade do código. Na Parte I, apresentada em duas listagens, encontram-se os *imports* necessários, a definição dos estados e a implementação das funções que gerenciam o início do jogo e verificam as condições de vitória. Na Listagem 8.33 são realizados os *imports* do *React* e das constantes, além de ser definido o estado que armazenará a palavra, as letras adivinhadas, o número de erros e o status do jogo. No segundo, a Listagem 8.34, são implementados os *useEffect* e as funções que iniciam um novo jogo e monitoram se o jogador atingiu as condições de vitória.

```
import React, { useState, useEffect, useCallback } from "react";
import { WORDS, MAX_MISTAKES } from "./constants";

export default function HangmanGame() {
  const [word, setWord] = useState("");
  const [guessedLetters, setGuessedLetters] =
   useState<string[]>([]);
  const [mistakes, setMistakes] = useState(0);
  const [gameStatus, setGameStatus] =
   useState<"playing" | "won" | "lost">("playing");
```

Listagem 8.33: Imports e definição dos estados do componente *HangmanGame*

```
  useEffect(() => {
    startNewGame();
  }, []);

  const startNewGame = () => {
    const randomWord =
     WORDS[Math.floor(Math.random() * WORDS.length)];
    setWord(randomWord);
    setGuessedLetters([]);
    setMistakes(0);
    setGameStatus("playing");
  };

  useEffect(() => {
    if (word && gameStatus === "playing") {
      const hasWon =
       [...word].every((letter) =>
        guessedLetters.includes(letter));
      if (hasWon) {
        setGameStatus("won");
      }
    }
  }, [word, guessedLetters, gameStatus]);

  const handleGuess = useCallback(
    (letter: string) => {
      if (gameStatus !== "playing" ||
       guessedLetters.includes(letter)) return;
      const newGuessedLetters = [...guessedLetters, letter];
      setGuessedLetters(newGuessedLetters);
      if (!word.includes(letter)) {
        const newMistakes = mistakes + 1;
        setMistakes(newMistakes);
        if (newMistakes >= MAX_MISTAKES) setGameStatus("lost");
      }
    },
    [gameStatus, guessedLetters, word, mistakes]
  );
}
```

Listagem 8.34: Implementação das funções do componente *HangmanGame*

Na Parte II do componente *HangmanGame* (Listagem 8.35), o retorno do componente é apresentado. Nesta parte, os elementos visuais são organizados com

classes do Tailwind CSS, integrando os componentes auxiliares da interface do jogo. O layout define uma interface intuitiva e responsiva, na qual os componentes *HangmanDrawing*, *WordDisplay* e *GameStatus* são exibidos juntamente com o componente *Keyboard*. Adicionalmente, um botão para reiniciar o jogo é renderizado quando o status não está em "*playing*". Essa estrutura demonstra claramente como os componentes se integram para formar a aplicação, facilitando a compreensão do leitor iniciante.

```
  return (
    <div className="flex flex-col items-center justify-center
     min-h-screen p-4 bg-gray-100">
      <h1 className="text-3xl font-bold mb-8">Jogo da Forca</h1>
      <div className="flex flex-col md:flex-row gap-8 w-full
       max-w-4xl">
        <div className="flex-1">
          <HangmanDrawing mistakes={mistakes} />
        </div>
        <div className="flex-1 flex flex-col items-center
         justify-between">
          <WordDisplay word={word} guessedLetters={guessedLetters}
           gameOver={gameStatus !== "playing"} />
          <GameStatus gameStatus={gameStatus} mistakes={mistakes}
           maxMistakes={MAX_MISTAKES} word={word} />
          <Keyboard guessedLetters={guessedLetters}
           handleGuess={handleGuess} disabled={gameStatus !==
            "playing"} />
          {gameStatus !== "playing" && (
            <button
              onClick={startNewGame}
              className="mt-6 px-6 py-2 bg-blue-600 text-white
               rounded-md hover:bg-blue-700 transition-colors"
            >
              Jogar Novamente
            </button>
          )}
        </div>
      </div>
    </div>
  );
}
```

Listagem 8.35: Retorno do componente *HangmanGame*

O componente auxiliar *Keyboard* foi dividido em duas partes para melhorar a visualização e a compreensão do código. Na primeira parte (Listagem 8.36), define-se a estrutura dos grupos de letras que compõem o teclado virtual. Essa estrutura organiza as letras em três linhas, facilitando a disposição dos botões na interface. Em seguida, na segunda parte (Listagem 8.37), o componente itera sobre esses grupos e renderiza um botão para cada letra. Durante esse mapeamento, cada botão é desabilitado caso a letra já tenha sido selecionada ou o jogo esteja finalizado, por meio de uma verificação da presença da letra em *guessedLetters*. Essa abordagem modular permite que o código seja mais legível e de fácil manutenção, orientando o leitor sobre a lógica de renderização dos elementos da interface.

```
import React from "react";

export function Keyboard({
  guessedLetters,
  handleGuess,
  disabled,
}: {
  guessedLetters: string[];
  handleGuess: (letter: string) => void;
  disabled: boolean;
}) {
  const rows = [
    ["Q", "W", "E", "R", "T", "Y", "U", "I", "O", "P"],
    ["A", "S", "D", "F", "G", "H", "J", "K", "L"],
    ["Z", "X", "C", "V", "B", "N", "M"],
  ];
```

Listagem 8.36: Definição dos grupos de letras do componente *Keyboard*

Na segunda parte (Listagem 8.37), o componente itera sobre os grupos de letras para renderizar os botões correspondentes. Cada botão é desabilitado se a letra já tiver sido selecionada ou se o jogo estiver finalizado. A verificação que define o estilo de cada botão, baseada na presença da letra em *guessedLetters*, é realizada durante o mapeamento dos grupos.

```
  return (
    <div className="w-full max-w-md">
      {rows.map((row, rowIndex) => (
        <div key={rowIndex} className="flex justify-center
         gap-1 my-1">
          {row.map((letter) => {
            const isGuessed = guessedLetters.includes(letter);
            return (
              <button
                key={letter}
                onClick={() => handleGuess(letter)}
                disabled={isGuessed || disabled}
                className={`w-8 h-10 text-center font-medium
                rounded ${
                   isGuessed ? "bg-gray-300 text-gray-500" :
                   "bg-gray-200 hover:bg-gray-300 text-black"
                   } ${disabled && !isGuessed ? "opacity-50
                   cursor-not-allowed" : ""}`}
              >
                {letter}
              </button>
            );
          })}
        </div>
      ))}
    </div>
  );
}
```

Listagem 8.37: Renderização dos botões do teclado no componente *Keyboard*

8.17 Visão geral sobre o Angular

O Angular é uma plataforma e framework *JavaScript* de código aberto baseada em *TypeScript*, projetada para a construção de aplicações de página única (*Single Page Applications* - SPA) por meio de HTML e *TypeScript* (Angular Team, 2025d). Mantido pelo Google, o Angular oferece um ecossistema completo de ferramentas, bibliotecas e APIs para criação de aplicações escaláveis e de alto desempenho (Angular Team, 2025d).

Lançado inicialmente em setembro de 2016 como uma reescrita completa do AngularJS, o Angular (também referido como Angular 2+ ou simplesmente Angular) introduziu um novo paradigma de desenvolvimento baseado em componentes, abandonando conceitos como *scope* e *controllers* em favor de hierarquias de componentes modularizados (AGULAR, 2025c).

A arquitetura do Angular fundamenta-se em componentes, cada um encapsulando HTML, CSS e lógica em TypeScript, o que facilita a reutilização e o teste de partes da interface do usuário (GeeksforGeeks, 2023). O framework adota injeção de dependências para promover modularidade e testabilidade e estende o HTML com *directives* que permitem modificar o comportamento e a aparência dos elementos do Document Object Model (DOM) (GeeksforGeeks, 2023). Além disso, o Angular integra um sistema de roteamento robusto para navegação em SPAs, que suporta parâmetros de rota, *lazy loading* de módulos e guardas de rota (Angular Team, 2025d).

O *Angular CLI* (*Command Line Interface*) fornece ferramentas para criação, construção, execução de testes e deployment de aplicações Angular, acelerando o fluxo de trabalho do desenvolvedor (Angular Team, 2025d). No ecossistema do Angular, existem bibliotecas como o Angular Material, que implementa componentes com base no Material Design do Google (AGULAR, 2025a), e Angular Universal, que possibilita renderização do lado do servidor (SSR) para melhorar desempenho e SEO (AGULAR, 2025b).

8.18 Iniciando um Projeto com o *Angular CLI*

O *Angular CLI* é a ferramenta oficial disponibilizada pela equipe do Angular que permite a criação, configuração e gerenciamento padronizado de projetos Angular diretamente pelo terminal (KODALI, 2024). Com base em *Schematics*, a CLI automatiza tarefas repetitivas, como a geração de componentes e serviços, além de fornecer comandos para a compilação e a execução do projeto em ambiente de desenvolvimento (KODALI, 2024).

Antes de iniciar um projeto com o *Angular CLI*, é fundamental ter instalado o *Node.js* e o `npm`[15]. O *Node.js* funciona como ambiente de execução JavaScript no

[15] *Node.js* disponível em: ⟨https://nodejs.org/⟩; `npm` disponível em: ⟨https://docs.npmjs.com/downloading-and-installing-node-js-and-npm⟩

servidor, enquanto o `npm` é responsável por gerenciar as dependências e pacotes do projeto (OpenJS Foundation, 2025)(npm, Inc., 2023).

Para criar um novo projeto Angular, utiliza-se o comando interativo *ng new*, que inicializa o *workspace* e gera a estrutura básica de arquivos e pastas conforme as melhores práticas recomendadas pela equipe Angular (Angular Team, 2025o). Durante a execução, é possível optar por configurações como o uso de testes unitários. Após a criação, o comando *ng serve* inicia um servidor de desenvolvimento, permitindo a visualização imediata das alterações no navegador em ⟨http://localhost:4200⟩ (Angular Team, 2025a).

```
1 npm install -g @angular/cli
2 ng new nome-do-projeto
3 cd nome-do-projeto
4 ng serve
```

Listagem 8.38: Passos para iniciar um projeto com *Angular CLI*

8.19 Criando o componente "Hello World" em Angular

O Angular utiliza uma arquitetura baseada em componentes para estruturar aplicações web. Cada componente no Angular é definido por uma classe em TypeScript, um *template* em HTML e um conjunto opcional de folhas de estilo (Angular Team, 2025b). O componente inicial "Hello World" serve como ponto de partida para compreender o fluxo de criação, compilação e exibição de componentes na aplicação. Nesta seção, é apresentado, passo a passo, o processo de criação e configuração do componente utilizando o *Angular CLI*.

Para gerar o componente "Hello World", utiliza-se o comando da Linha 1 na Listagem 8.39. O *Angular CLI* provê o gerador de componentes por meio do subcomando *generate component* (Angular Team, 2025a). O nome "hello-world"especificado no comando determina o diretório e os arquivos iniciais do componente. Internamente, o Angular aplica um esquema que gera arquivos de código TypeScript, *template* HTML e estilos CSS padronizados. Essa abordagem agiliza a criação de componentes padronizados e segue as convenções definidas pela equipe do Angular. O uso da CLI é essencial para manter consistência e evitar erros manuais na configuração inicial do componente (Angular Team, 2025a).

```
1 ng generate component hello-world
```

Listagem 8.39: Criação do componente HelloWorld via Angular CLI

O arquivo *hello-world.component.ts* define a classe principal do componente, conforme mostrado na Listagem 8.40. A importação de *Component* e *OnInit* no

início define o *decorator* e o ciclo de vida do componente (Angular Team, 2025b). No *decorator* *@Component*, declarado entre as Linhas 3 e 7, são especificados o seletor, o *template* e as folhas de estilo associadas. A propriedade *selector* determina a tag HTML personalizada que representará o componente na aplicação. Os campos *templateUrl* e *styleUrls* indicam, respectivamente, o arquivo de marcação e as folhas de estilo aplicadas ao componente. Por fim, a classe *HelloWorldComponent* implementa *OnInit* e define métodos de inicialização, ainda sem lógica adicional (Angular Team, 2025b).

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-hello-world',
5    templateUrl: './hello-world.component.html',
6    styleUrls: ['./hello-world.component.css']
7  })
8  export class HelloWorldComponent implements OnInit {
9    constructor() { }
10   ngOnInit(): void { }
11 }
```

Listagem 8.40: Código-fonte TypeScript do componente HelloWorld

O *template* HTML do componente, mostrado na Listagem 8.41, contém a marcação fundamental para exibir a mensagem. Este arquivo é referenciado pelo *templateUrl* configurado no *decorator* do componente. A separação entre a lógica em TypeScript e a marcação HTML segue o princípio de responsabilidades únicas na arquitetura de software (FREEMAN; SANDERSON, 2024). Essa separação facilita testes isolados e manutenção da interface do usuário (FREEMAN; SANDERSON, 2024).

```
1  <h1>Hello World!</h1>
```

Listagem 8.41: Template HTML do componente HelloWorld

Para que o Angular reconheça o componente, é necessário registrá-lo em um módulo, conforme ilustrado na Listagem 8.42. Na Linha 4, a classe *HelloWorld-Component* é explicitamente importada para ser disponibilizada no módulo raiz da aplicação. As declarações dentro do array *declarations*, entre as Linhas 7 e 10, informam ao compilador do Angular quais componentes pertencem ao módulo. O módulo principal, *AppModule*, importa também o *BrowserModule*, essencial para aplicações web (Angular Team, 2025b). A configuração de *bootstrap*, na Linha 15, define o componente responsável pela inicialização da aplicação. Com essa abordagem modular, novos componentes podem ser facilmente adicionados e gerenciados na aplicação (Angular Team, 2025b).

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { AppComponent } from './app.component';
4  import { HelloWorldComponent } from
'./hello-world/hello-world.component';
5
6  @NgModule({
7    declarations: [
8      AppComponent,
9      HelloWorldComponent
10   ],
11   imports: [
12     BrowserModule
13   ],
14   providers: [],
15   bootstrap: [AppComponent]
16 })
17 export class AppModule { }
```

Listagem 8.42: Registro do componente HelloWorld em AppModule

Finalmente, para exibir o componente "Hello World" na aplicação, a tag customizada correspondente é adicionada em *app.component.html*. A Linha 1 da Listagem 8.43 demonstra a inclusão de `<app-hello-world></app-hello-world>`. O Angular identifica essa tag e instancia a classe *HelloWorldComponent* no local correspondente do DOM.

```
1  <app-hello-world></app-hello-world>
```

Listagem 8.43: Inclusão do componente HelloWorld em AppComponent

Esse processo de renderização ocorre durante o ciclo de *bootstrap* da aplicação, substituindo a tag pela saída do *template*. Assim, o texto "Hello World!" é apresentado ao usuário de forma dinâmica e reativa. Esse fluxo completo demonstra como componentes são criados, registrados e exibidos no Angular (Angular Team, 2025b).

8.20 Editando a Classe de um Componente

O Angular organiza aplicações web por meio de componentes, cada um definido por uma classe TypeScript, um template HTML e, opcionalmente, folhas de estilo(Angular Team, 2025e; FREEMAN; SANDERSON, 2024). A classe do componente é onde se declaram sua lógica, propriedades e métodos, separando responsabilidades (Angular Team, 2025b). Atualizar essa classe permite oferecer funcionalidades dinâmicas sem misturar código de apresentação com lógica de negócios(Angular Team, 2025d).

Nesta seção, é demonstrado como adicionar uma propriedade à classe *AppComponent* para exibir dados dinâmicos no *template*. A propriedade *developer* será declarada dentro da classe, aproveitando a inferência de tipos do TypeScript, dispensando a especificação explícita de *: string*. Essa abordagem simplifica o código e mantém a clareza da implementação.

Para incluir a propriedade, edita-se o arquivo *app.component.ts* e, em seguida, atualiza-se a definição da classe conforme a Listagem 8.44. O valor inicial atribuído à propriedade é *Developer*, mas pode ser ajustado ou proveniente de serviços externos em cenários reais.

```
export class AppComponent {
  developer = 'Developer';
}
```

Listagem 8.44: Declaração da propriedade *developer* na classe

Com a propriedade disponível, utiliza-se a sintaxe de interpolação para exibir seu valor no *template*. Basta envolver a expressão entre chaves duplas {{ }}, conforme ilustrado no Listagem 8.45. Essa técnica mantém a separação de camadas e atualiza o DOM automaticamente(Angular Team, 2025u).

```
@Component({
  selector: 'app-root',
  template: `Hello {{ developer }}`,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  developer = 'Developer';
}
```

Listagem 8.45: Interpolação de propriedade no template inline

Como exercício adicional, é demonstrada a interpolação de expressões aritméticas simples no *template*. A inserção de {{ *1 + 1* }} permite calcular e exibir resultados diretamente, reforçando o poder de ligação unidirecional de dados do Angular (Listagem 8.46)(Angular Team, 2025u). A expressão final exibe "Hello Developer, 2"conforme mostrado na Listagem 8.46.

```
@Component({
  selector: 'app-root',
  template: `Hello {{ developer }}, {{ 1 + 1 }}`,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  developer = 'Developer';
}
```

Listagem 8.46: Interpolação de expressão no template

A interpolação no Angular também suporta chamadas a métodos públicos da classe e exibição de objetos complexos, mantendo clara a distinção entre lógica de negócios e apresentação(Angular Team, 2025t). Essa flexibilidade é essencial para desenvolver interfaces reativas e de fácil manutenção.

8.21 Agrupando Componentes

No Angular, cada componente é identificado por um seletor que funciona como uma tag HTML personalizada. Antes de compor dois componentes, deve-se definir cada um separadamente (Angular Team, 2025f). O exemplo na Listagem 8.47 apresenta dois componentes: *ProfileComponent* exibe o nome de usuário, enquanto *DashboardComponent* não contém marcação inicial. Ambos são declarados sem referências mútuas, funcionando de forma isolada.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-profile',
  template: `
    User: {{ username }}
  `
})
export class ProfileComponent {
  username = 'developer';
}

@Component({
  selector: 'app-dashboard',
  template: ``,
  imports: []
})
export class DashboardComponent { }
```

Listagem 8.47: Definição independente de *ProfileComponent* e *DashboardComponent*

Para agrupar componentes, é necessário importar o componente secundário no decorador do componente pai. No exemplo, o *ProfileComponent* é adicionado ao *array imports* de *DashboardComponent*. Além disso, modifica-se o *template inline* para incluir o seletor `<app-profile />` dentro de uma seção HTML, conforme Listagem 8.48. Essas alterações habilitam a renderização do conteúdo de *ProfileComponent* no contexto do *DashboardComponent*.

```
import { Component } from '@angular/core';
import { ProfileComponent } from './profile.component';

@Component({
  selector: 'app-dashboard',
  template: `
    <section>
      <app-profile />
    </section>
  `,
  imports: [ProfileComponent]
})
export class DashboardComponent { }
```

Listagem 8.48: Composição de *ProfileComponent* dentro de *DashboardComponent*

Ao agrupar componentes dessa forma, o Angular mantém cada unidade de interface modular, ao mesmo tempo em que permite a construção de interfaces mais complexas e coesas. As *tags* personalizadas podem ser combinadas com elementos HTML nativos, diretivas de estrutura e estilos, promovendo a escalabilidade da aplicação e a clareza na organização do código (Angular Team, 2025b).

A prática de compor componentes é fundamental para explorar todo o potencial do *framework*. Com ela, desenvolvedores conseguem criar layouts sofisticados, aproveitar a reutilização de código e manter uma arquitetura limpa e fácil de evoluir (Angular Team, 2025e).

8.22 Renderização Condicional

No Angular, a *renderização condicional* permite exibir ou ocultar trechos de interface com base em condições lógicas. Esse recurso é essencial para criar layouts dinâmicos que respondam ao estado da aplicação. A técnica se apoia em diretivas estruturais, que alteram a estrutura do DOM em tempo de execução. Para iniciantes, entender como e quando usar cada diretiva evita *code smells* e facilita a manutenção. Esta seção apresenta algumas formas de condicionar a renderização de componentes e elementos.

Uma diretiva usada no Angular é **ngIf*, que inclui ou remove um elemento conforme uma expressão booleana. No *template*, basta preceder o elemento com o asterisco e a condição desejada. Quando a expressão é avaliada como *true*, o

elemento é renderizado; caso contrário, é removido do DOM. Essa abordagem melhora o desempenho, pois não apenas oculta o elemento, mas também evita sua criação. O exemplo na Listagem 8.49 ilustra seu uso básico (Angular Team, 2025h).

```
<button *ngIf="isLoggedIn">Logout</button>
<p *ngIf="!isLoggedIn">Faça login para continuar</p>
```

Listagem 8.49: Exemplo de uso de **ngIf* para exibir botão de logout

Em contraste, a sintaxe com *@if* apresentada na Listagem 8.50 simula lógica *server-side* de autenticação. Nesse caso, o código decide se inclui trechos de HTML conforme o usuário já tenha se autenticado no servidor antes do envio da página (Angular Team, 2025i). O *@if* é processado no servidor, gerando HTML estático que não responde a mudanças no navegador. Já o **ngIf* do Angular atua no *runtime* do *browser*, oferecendo reatividade em aplicações SPA (Angular Team, 2025h).

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    @if (isUserAuthenticatedServerSide) {
      <div>Bem-vindo, usuário autenticado no servidor!</div>
    } @else {
      <div>Por favor, faça login para acessar o sistema.</div>
    }
  `,
})
export class AppComponent {
  isUserAuthenticatedServerSide = false;
}
```

Listagem 8.50: Exemplo hipotético de uso de @if em template server-side para controle de login

Outra diretiva estrutural importante no Angular é *@switch*, que avalia uma expressão e renderiza apenas o bloco correspondente. Nas *engines server-side* como Razor no ASP.NET, a diretiva condicional usa a sintaxe *@switch*. O *@switch* avalia uma expressão e renderiza o bloco *@case* correspondente, com *fallback* em *@default* (Microsoft Docs, 2025). Essa lógica ocorre no servidor antes de enviar o HTML ao cliente, produzindo saída estática. A Listagem 8.51 apresenta um exemplo de um cenário de navegação baseado em abas no servidor (Angular Team, 2025j).

```
@switch (currentTab)
{
    @case "home":
        <div>Conteúdo da Home (renderizado server-side)</div>
        @break;
    @case "profile":
        <div>Conteúdo do Perfil (renderizado server-side)</div>
        @break;
    @case "settings":
        <div>Conteúdo das Configurações (renderizado server
        -side)</div>
        @break;
    @default:
        <div>Selecione uma aba</div>
        @break;
}
```

Listagem 8.51: Exemplo hipotético de uso de @switch em template server-side para seleção de abas

A prática de renderização condicional contribui para aplicações mais reativas e modulares. Ao dominar tais práticas de renderização, iniciantes podem construir interfaces que refletem fielmente o estado da aplicação. Essas diretivas, aliadas às boas práticas de arquitetura, tornam o código mais legível e performático (Angular Team, 2025g).

8.23 Iteração sobre elementos do DOM

No desenvolvimento com Angular, a *iteração* sobre os elementos do DOM é fundamental para aplicar estilos, atributos ou comportamentos de forma dinâmica e eficiente. Em aplicações *real-time*, muitas vezes é necessário percorrer um conjunto de componentes renderizados para sincronizar estados ou introduzir micro-interações sem sobrecarregar o *change detection*. A demonstração prática deste conceito encontra-se na Listagem 8.52, que ilustra o uso das diretivas.

Para acessar múltiplos elementos do *template*, o Angular disponibiliza a diretiva estrutural *@ViewChildren*. Ao declará-la em um componente, o framework injeta uma instância de `QueryList<ElementRef>`, que agrupa todas as referências aos elementos marcados com a mesma *template reference variable*. No exemplo (Listagem 8.52), observa-se a declaração de `@ViewChildren('itemElement') itens!:QueryList<ElementRef>;` logo após a lista de itens, o que garante a coleta de todas as referências antes da manipulação.

A manipulação do DOM deve ser realizada preferencialmente por meio do serviço *Renderer2* para assegurar compatibilidade com plataformas não-DOM (por exemplo, *NativeScript*). No *hook ngAfterViewInit*, que é acionado após a inicialização completa da *view*, o componente itera sobre cada *ElementRef* usando o método

forEach. Conforme na Listagem 8.52, o estilo de cor de cada elemento alterna entre verde e azul de acordo com seu índice, graças à chamada *this.renderer.setStyle(...)*.

```
import { Component, ViewChildren, QueryList, ElementRef,
         AfterViewInit, Renderer2 } from '@angular/core';

@Component({
  selector: 'app-list',
  template:
    <li #itemElement *ngFor="let item of items">{{ item }}</li>
})
export class ListComponent implements AfterViewInit {
  @ViewChildren('itemElement') itens!: QueryList<ElementRef>;
  items = ['Angular', 'DOM', 'Iteração'];

  constructor(private renderer: Renderer2) {}

  ngAfterViewInit() {
    this.itens.forEach((el: ElementRef, idx: number) => {
      this.renderer.setStyle(el.nativeElement, 'color',
        idx % 2 ? 'blue' : 'green');
    });
  }
}
```

Listagem 8.52: Exemplo de uso de @ViewChildren e Renderer2

Além de *@ViewChildren*, o Angular também permite iterar diretamente *arrays* de dados no *template* com a diretiva **ngFor*. Embora o **ngFor* não exponha *ElementRefs*, é possível combinar seu uso com *template references* para capturar cada instância de elemento gerado (Listagem 8.53). Outra abordagem envolve a criação de *directives* customizadas que implementam a interface *DoCheck* ou escutam eventos de *MutationObserver* para reagir a mudanças no DOM de forma granular, sem necessidade de recorrer ao *forEach* manual no componente.

```
// item-list.component.html
<ul>
  <li *ngFor="let item of items; let i = index">
    <span #itemRef>{{ item.name }}</span>
  </li>
</ul>

// item-list.component.ts
import { Component, AfterViewInit, QueryList, ElementRef,
ViewChildren } from '@angular/core';

@Component({
  selector: 'app-item-list',
  templateUrl: './item-list.component.html'
})
export class ItemListComponent implements AfterViewInit {
  @ViewChildren('itemRef') itemRefs!: QueryList<ElementRef>;
  items = [{ name: 'Item A' }, { name: 'Item B' },
  { name: 'Item C' }];

  ngAfterViewInit() {
    this.itemRefs.forEach((ref: ElementRef, index: number) => {
      console.log(`Elemento ${index}:`, ref.nativeElement);
    });
  }
}
```

Listagem 8.53: Exemplo de uso de *ngFor com template reference e @ViewChildren

O código apresentado na Listagem 8.53 demonstra uma solução para capturar elementos gerados dinamicamente pelo *ngFor*. Nele, a diretiva *ngFor* repete um elemento de lista para cada item do array *items*, e cada ¡span¿ interno é marcado com a diretiva de referência *template #itemRef*.

Ademais, o Angular 17 introduziu o bloco de controle de fluxo *@for*, permitindo iterar coleções de forma nativa no *template* sem a necessidade de importar a diretiva *NgFor* de *@angular/common*. Na Listagem 8.54 é apresentado um exemplo de uso desse bloco: um componente que define uma lista de frutas e, no *template*, percorre o *array fruits*, exibindo o nome de cada objeto. A sintaxe é inspirada no laço *for* do JavaScript, tornando-a mais familiar e concisa: a expressão de iteração (*fruit of fruits*) e a função de *tracking* (*track fruit.id*) são declaradas entre parênteses, seguidas pelo bloco de conteúdo a ser repetido.

No mesmo exemplo (Listagem 8.54), é apresentado também o bloco *@empty*, que renderiza uma mensagem alternativa quando a coleção está vazia, substituindo o padrão anterior de combinar **ngIf* e *else*. Dessa forma, o *@for* configura-se

como uma solução mais elegante, legível e alinhada ao estilo de *loops* do próprio JavaScript nos *templates* do Angular.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template:
    <section>
      <h2>Lista de Frutas</h2>
      @for (fruit of fruits; track fruit.id) {
        <p>{{ fruit.name }}</p>
      }
      @empty {
        <p>Nenhuma fruta disponível.</p>
      }
    </section>

})
export class AppComponent {
  fruits = [
    { id: 0, name: 'Maçã' },
    { id: 1, name: 'Banana' },
    { id: 2, name: 'Laranja' },
    { id: 3, name: 'Manga' },
    { id: 4, name: 'Uva' },
  ];
}
```

Listagem 8.54: Implementação de *@for* para lista de frutas

A função de *tracking* é obrigatória no *@for*, garantindo que o Angular possa identificar mudanças na coleção sem re-renderizar todo o bloco, o que melhora o desempenho em listas grandes. Caso os elementos não possuam um identificador único, pode-se usar o índice da iteração (*$index*), mas isso só é recomendado em cenários simples. Além disso, o bloco *@empty* permite renderizar conteúdo alternativo quando a coleção estiver vazia, substituindo os padrões anteriores de **ngIf* combinados com *else*. Assim, o *@for* configura-se como uma solução elegante, legível e alinhada ao estilo de *loops* do próprio JavaScript para *templates* do Angular.

8.24 Como o Angular Envia Dados para o HTML

Na Seção 8.20 foi usada uma das formas de enviar dados de forma dinâmica para o HTML de um componente feito com classe; essa técnica se chama *binding* por interpolação, um dos tipos de *binding* que o Angular proporciona. A interação entre a lógica em TypeScript e o *template* em HTML no Angular ocorre por meio de

diferentes mecanismos de *binding* (FREEMAN; SANDERSON, 2024). Esta seção apresenta outras duas abordagens que viabilizam essa comunicação: o *Property Binding* e o *Event Handling*. Cada técnica estabelece um canal específico para passagem de dados ou eventos, conferindo reatividade e organização ao *template*, sem recorrer a manipulações imperativas do DOM (Angular Team, 2025p; Angular Team, 2025l).

O *Property Binding* estabelece um vínculo direto entre expressões do componente e propriedades de elementos do DOM, encapsulando-as entre colchetes. Essa abordagem viabiliza a atualização automática sempre que o valor de origem for alterado, reforçando a consistência entre o estado e a apresentação visual e aproveitando a checagem de tipos em compilação (Angular Team, 2025p). A Listagem 8.55 ilustra um componente no qual as propriedades *isDisabled*, *user.avatarUrl* e *user.name* são atribuídas aos atributos *disabled*, *src* e *value*, respectivamente.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-exemplo-property-binding',
  template: `
    <button [disabled]="isDisabled">Enviar</button>
    <img [src]="user.avatarUrl" alt="Avatar do usuário">
    <input type="text" [value]="user.name">
  `
})
export class ExemploPropertyBindingComponent {
  isDisabled: boolean = true;
  user = { avatarUrl: 'https://example.com/avatar.png', name:
  'Usuário' };
}
```

Listagem 8.55: Componente com Property Binding

O *Event Handling*, ou *Event Binding*, permite que ações do usuário disparem métodos do componente, utilizando a sintaxe de parênteses ao redor do evento. Essa técnica introduz reatividade de entrada, viabilizando interação direta, como no exemplo da Listagem 8.56, em que o clique em um botão aciona o método *onEnviar()*, responsável por executar a lógica associada (Angular Team, 2025l).

```
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-exemplo-event-handling',
5    template: `
6      <button (click)="onEnviar()">Enviar</button>
7    `
8  })
9  export class ExemploEventHandlingComponent {
10   onEnviar(): void {
11     console.log('Evento enviado');
12   }
13 }
```

Listagem 8.56: Componente com Event Handling

O uso de *Property Binding* ou *Event Handling* estabelece uma ligação entre o *template* HTML e os dados presentes no código, alinhando-se aos princípios de arquitetura baseada em componentes (FREEMAN; SANDERSON, 2024). Ademais, o Angular disponibiliza outras formas de *binding* que ampliam a interação e a personalização do *template*, tais como *Two-Way Binding*[16], *Attribute Binding*[17], e *Class Binding* e *Style Binding*[18].

8.25 As *"props"* de um Componente no Angular

Em Angular, as *props* estabelecem mecanismos padronizados para que um componente filho receba parâmetros do componente pai e, simultaneamente, envie eventos de volta por meio de canais explícitos. Ao se trabalhar com decoradores como *@Input* e *@Output*, o desenvolvedor mantém clara a responsabilidade de cada entidade no fluxo de dados, reforçando uma arquitetura de componentes que preza pela legibilidade e pela escalabilidade. Esta seção apresenta de forma progressiva o desenvolvimento de um componente, partindo de uma implementação inicial sem os decoradores até a aplicação dos decoradores que definem as *props* do componente.

Na Listagem 8.57 a partir da linha 4 observa-se um componente cuja propriedade interna *titulo* foi definida diretamente na classe, sem exposição formal para o pai, e um método *acao()* que dispara uma rotina interna. Apesar de funcionar isoladamente, essa abordagem carece de um contrato explícito para troca de informações entre componentes. Ainda na Listagem 8.57, a linha 24 demonstra esse ponto de partida, ilustrando o comportamento de *props* sem o uso de *@Input* e *@Output*.

[16] Guia para *Two-Way Binding* disponível em: ⟨https://v17.angular.io/guide/two-way-binding⟩
[17] Guia para *Attribute Binding* disponível em: ⟨https://v17.angular.io/guide/attribute-binding⟩
[18] Guia para *Class Binding* e *Style Binding* disponível em: ⟨https://v17.angular.io/guide/class-binding⟩

```
/* filho.component.ts */
import { Component } from '@angular/core';

@Component({
  selector: 'app-filho',
  template: `
    <h2>{{ titulo }}</h2>
    <button (click)="acao()">Clique</button>
  `
})
export class FilhoComponent {
  titulo: string = 'Título padrão';
  acao(): void {
    console.log('Ação interna');
  }
}

/* pai.component.ts */
import { Component } from '@angular/core';

@Component({
  selector: 'app-pai',
  template: `
    <app-filho titulo="valorExterno" (click)="metodoPai()">
    </app-filho>
  `
})
export class PaiComponent {
  metodoPai(): void {
    console.log('Evento no pai');
  }
}
```

Listagem 8.57: Implementação inicial sem *@Input* e *@Output*

Para formalizar o contrato de comunicação, aplica-se *@Input* na propriedade que receberá valores e *@Output* em conjunto com *EventEmitter* para emitir eventos customizados. A Listagem 8.58, linhas 13 e 14 apresentam a definição aprimorada, na qual o filho expõe a *@Input() titulo* e dispara o evento *@Output() clicado* para o pai, que responde por meio de *(clicado)*. Esta configuração favorece a manutenção do código e a previsibilidade no fluxo de dados.

```
/* filho.component.ts */
import { Component, Input, Output, EventEmitter } from
'@angular/core';

@Component({
  selector: 'app-filho',
  template: `
    <h2>{{ titulo }}</h2>
    <button (click)="acao()">Clique</button>
  `
})
export class FilhoComponent {
  @Input() titulo!: string;
  @Output() clicado = new EventEmitter<void>();

  acao(): void {
    this.clicado.emit();
  }
}

/* pai.component.ts */
import { Component } from '@angular/core';

@Component({
  selector: 'app-pai',
  template: `
    <app-filho
      [titulo]="tituloDoPai"
      (clicado)="metodoPai()">
    </app-filho>
  `
})
export class PaiComponent {
  tituloDoPai: string = 'Título vindo do pai';
  metodoPai(): void {
    console.log('Evento recebido no pai');
  }
}
```

Listagem 8.58: Implementação com *@Input* e *@Output*

Dessa forma, as *props* configuram-se como elementos fundamentais para estabelecer um fluxo de dados unidirecional e eventos diretos entre componentes, respeitando os princípios do Angular e promovendo um código modular e coeso. A adoção de *@Input* e *@Output* reforça a clareza das interfaces.

8.26 Deferrable Views

O uso de *Deferrable Views* em aplicações Angular permite adiar a renderização de partes menos críticas até que estejam visíveis na tela, otimizando o tempo de renderização inicial e melhorando a experiência do usuário (Angular Team, 2025k). Essa técnica emprega diretivas especiais como *@defer*, *@placeholder* e *@loading*, que controlam, respectivamente, quando adiar a exibição, o conteúdo estático substituto e o indicador de carregamento mínimo. Ao adotar esse padrão, o desenvolvedor separa lógica de apresentação imediata de funcionalidades complementares, garantindo interfaces mais responsivas sem sacrificar a clareza do código.

Para ilustrar a configuração básica, considere o arquivo *app.ts* definido na Listagem 8.59. Nela, o componente raiz é anotado com *@defer* no interior do *template inline*, referenciando diretamente o ponto de exibição (*viewport*). O bloco *@placeholder* apresenta um conteúdo estático enquanto o componente adiado não está pronto, e o bloco *@loading* assegura um tempo mínimo de 2 segundos antes de renderizar o conteúdo final, evitando piscadas visuais (Angular Team, 2025k).

```
import { Component } from '@angular/core';
import { Comments } from './comments';

@Component({
  selector: 'app-root',
  template: `
    <div>
      <h1>Ultimas noticias</h1>
      <article>
        <p>
          Lorem Ipsum is simply dummy text of the printing and
          typesetting industry...
        </p>
      </article>
      @defer (on viewport) {
        <topics/>
      }
      @placeholder {
        <p>Tópicos futuros</p>
      }
      @loading (minimum 2s) {
        <p>Carregando...</p>
      }
    </div>
  `,
  imports: [Comments],
})
export class App {}
```

Listagem 8.59: Configuração de *Deferrable Views* em *app.ts*

Em seguida, define-se o componente responsável pelos dados adiados, separando sua lógica em outro módulo (*topics.ts* na Listagem 8.60). Esta abordagem modular reforça a manutenibilidade e o reaproveitamento de código, permitindo que o componente seja carregado apenas quando efetivamente necessário (Angular Team, 2025b).

```
import { Component } from '@angular/core';

@Component({
  selector: 'topics',
  template: `
    <div>
      <div>Principais notícias de 2025!!</div>
      <div>EXTRA EXTRA!</div>
      <div>ANGULAR OU REACT !?</div>
    </div>
  `,
})
export class Comments {}
```

Listagem 8.60: Implementação do componente adiado *Comments* em *topics.ts*

Após a configuração, o Angular identifica automaticamente quando o componente entra no *viewport* e inicia o carregamento, primeiro exibindo o *placeholder*, depois mantendo o indicador de *loading* por pelo menos dois segundos e, finalmente, renderizando o conteúdo real.

8.27 Rotas em *Angular*

O uso de *routing* no *Angular* permite organizar a navegação entre diferentes partes da aplicação de forma declarativa e modular. A definição de rotas centraliza os caminhos de acesso a componentes, promovendo coesão e reduzindo o acoplamento entre módulos(Angular Team, 2025c; Angular Team, 2025e). Além disso, a abordagem baseada em rotas facilita a manutenção e o carregamento (*lazy loading*), melhorando a performance(Angular Team, 2025m). Nesta seção, será apresentado, passo a passo, como configurar e utilizar as principais funcionalidades de roteamento no *Angular*.

O exemplo na Listagem 8.61 demonstra a criação de três rotas principais: *home*, *about* e *contact*. Para configurar as rotas básicas, é criado um módulo dedicado chamado *AppRoutingModule*, onde é importado o *RouterModule* e são registrados os caminhos desejados. A função *forRoot()* do *RouterModule* recebe um *array* de objetos *Route*, cada um mapeando um *path* a um *component*(Angular Team, 2025s). O módulo de roteamento deve ser exportado para que o roteamento fique disponível na raiz da aplicação.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
import { ContactComponent } from './contact/contact.component';

const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Listagem 8.61: Definição do *AppRoutingModule*

Com o módulo de roteamento configurado, é possível criar links de navegação no *template* por meio da diretiva *routerLink*. Essa diretiva gera âncoras que disparam a mudança de rota sem recarregar a página, preservando o estado da aplicação (Angular Team, 2025r). No exemplo da Listagem 8.62, são definidos links no componente de navegação, permitindo acesso rápido às seções. A sintaxe é simples: basta passar o caminho desejado como valor de *routerLink*. Essa estratégia reforça a separação entre a lógica de roteamento e a de apresentação.

```
<nav>
  <ul>
    <li><a routerLink="/home">Home</a></li>
    <li><a routerLink="/about">About</a></li>
    <li><a routerLink="/contact">Contact</a></li>
  </ul>
</nav>
<router-outlet></router-outlet>
```

Listagem 8.62: Uso de *routerLink* no *template*

Para navegação programática, o *Angular* disponibiliza o serviço *Router*, que permite, em tempo de execução, redirecionar o usuário e manipular o histórico de rotas(Angular Team, 2025n). Essa abordagem é útil em cenários que exigem lógica antes da transição, como a autenticação ou a validação de dados. No exemplo da Listagem 8.63, após o envio de um formulário, o método *navigate()* redireciona para a rota */home*. Essa técnica oferece maior controle sobre o fluxo de navegação da aplicação.

```typescript
import { Component } from '@angular/core';
import { Router } from '@angular/router';

@Component({ /* ... */ })
export class LoginComponent {
  constructor(private router: Router) { }

  onSubmit() {
    // lógica de autenticação ...
    this.router.navigate(['/home']);
  }
}
```

Listagem 8.63: Navegação programática com *Router*

Por fim, as rotas podem receber parâmetros dinâmicos, acessíveis pelo serviço *ActivatedRoute*. Isso possibilita criar URLs parametrizadas, como detalhes de um item baseados em seu *id*(Angular Team, 2025q). Na Listagem 8.64, a rota *product/:id* é definida e, no componente, o *ActivatedRoute* é usado para extrair o valor de *id*. Essa abordagem é fundamental para aplicações que apresentam dados detalhados de forma dinâmica.

```typescript
const routes: Routes = [
  { path: 'product/:id', component: ProductDetailComponent }
];

// No ProductDetailComponent
import { ActivatedRoute } from '@angular/router';

export class ProductDetailComponent {
  productId: string;

  constructor(private route: ActivatedRoute) {
    this.productId = this.route.snapshot.paramMap.get('id')!;
  }
}
```

Listagem 8.64: Uso de parâmetros de rota com *ActivatedRoute*

8.28 Exemplo prático: Jogo-da-velha

O *Jogo-da-velha* é um clássico *game* de estratégia para dois jogadores, tradicionalmente representado em uma grade 3×3, em que cada participante marca alternadamente as células com *X* ou *O*. O objetivo é alinhar três símbolos iguais

em linha reta, seja na horizontal, na vertical ou na diagonal, antes do adversário, evitando ao mesmo tempo que o oponente alcance essa combinação.

Para iniciar o desenvolvimento do *Jogo-da-velha* com Angular, utiliza-se o *CLI* oficial: basta executar *npm install -g @angular/cli* e, em seguida, *ng new jogo-da-velha* com as *flags* *–standalone* e *–routing=false* para gerar um *project* enxuto focado em componentes independentes(Angular Team, 2025a). Após navegar até o diretório do *project*, a estrutura já conta com *AppComponent* pronto para ser adaptado.

Na Listagem 8.65 apresenta-se o componente central que compõe o *template* principal da aplicação, integrando os subcomponentes *BoardComponent*, *StatusComponent*, *ScoreBoardComponent* e *ControlsComponent* a partir da linha 8, além do provedor *GameService* responsável, na Linha 10, pela lógica de *state management*. No *template* central da aplicação, localizado a partir da Linha 11, encontram-se as *tags* dos componentes importados. Finalmente, a partir da Linha 20, encontram-se as estilizações em CSS do componente central, como a definição de classes, fontes e *media-queries*.

```
import { Component } from "@angular/core";
import { CommonModule } from "@angular/common";
/* Importações Javascript dos componentes principais e do service */
@Component({
  selector: "app-root",
  standalone: true,
  imports: [CommonModule, BoardComponent, StatusComponent,
  ScoreBoardComponent, ControlsComponent],
  providers: [GameService],
  template: `
    <div class="game-container">
      <h1 class="title">Jogo da Velha</h1>
      <app-score-board></app-score-board>
      <app-status></app-status>
      <app-board></app-board>
      <app-controls></app-controls>
    </div>
  `,
  styles: [
    `
      :host {
        display: block;
        font-family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
        color: #1e293b;
        background-color: #f8fafc;
        min-height: 100vh;
        padding: 20px;
      }

      .game-container {
        max-width: 500px;
        margin: 0 auto;
        padding: 20px;
      }

      .title {
        text-align: center;
        font-size: 2.5rem;
        font-weight: 700;
        margin-bottom: 20px;
        color: #0f172a;
        background: linear-gradient(90deg, #3b82f6, #f97316);
        -webkit-background-clip: text;
        -webkit-text-fill-color: transparent;
        background-clip: text;
        text-fill-color: transparent;
      }

      @media (max-width: 600px) {
        .game-container {
          padding: 10px;
        }

        .title {
          font-size: 2rem;
        }
      }
    `,
  ],
})
export class AppComponent {}
```

Listagem 8.65: Componente central *AppComponent* do Jogo-da-Velha

8.28.1 Criando um *Service* no Angular

Para estruturar funcionalidades compartilhadas entre vários componentes, o Angular utiliza o conceito de *services*, classes que podem ser injetadas via *dependency injection* e mantêm lógica de negócios ou *state management*. Para gerar um *service* utilizando o *CLI*, é necessário executar o comando na Listagem 8.66.

```
ng generate service services/game
```

Listagem 8.66: Gerando um serviço com Angular CLI

O comando da Listagem 8.66 cria automaticamente o arquivo *game.service.ts* e registra o decorador *@Injectable{providedIn: 'root'}*, garantindo que o serviço seja *singleton* em toda a aplicação. A injeção como *providedIn: 'root'* dispensa a inclusão manual em *providers* de módulos específicos, simplificando a manutenção da infraestrutura de injeção de dependências.

Em seguida, o desenvolvedor implementa a lógica desejada dentro da classe *GameService*, aproveitando recursos do *RxJS* para expor fluxos de dados reativos. A prática de isolar responsabilidades nesse *service* ajuda a manter os componentes focados exclusivamente na *view*, reforçando a separação de camadas recomendada pelo Angular(Angular Team, 2025e).

8.28.2 Análise do *GameService* do Jogo-da-Velha

O *GameService* define primeiramente a constante *WINNING_COMBINATIONS*, matriz que lista todas combinações de índices possíveis para vitória em uma grade 3×3, fundamental para a função de verificação de resultado. Em seguida, *initialState* configura o estado inicial (*board* vazio, jogador atual como *X*, placar zerado e flags de término de jogo), servindo de referência para *resetGame()* e *resetScores()*.

Para manter o estado reativo, utiliza-se `BehaviorSubject<GameState>`, que emite imediatamente seu último valor a novos assinantes. O atributo *gameState$* expõe o *Observable* para componentes, permitindo que alterem suas *views* ao receberem novas emissões. A propriedade *gameState* fornece acesso síncrono ao valor corrente do estado, facilitando validações antes da emissão de atualizações.

A função *makeMove(index: number)* realiza diversas etapas: 1. Valida se o jogo já terminou ou se a célula está ocupada. 2. Cria um novo *array* de *board* com a marcação do jogador atual. 3. Executa *checkWinner()*, que percorre *WINNING_COMBINATIONS* e identifica se há três símbolos iguais alinhados. 4. Determina *isDraw* e *isGameOver* e atualiza o placar conforme vitória ou empate. 5. Emite o novo estado via *gameStateSubject.next(...)*, alternando o jogador atual automaticamente.

Os métodos *resetGame()* e *resetScores()* reutilizam *initialState* para recomeçar o tabuleiro, preservando ou zerando o placar, respectivamente. Por fim, *checkWinner(board: Player[])* encapsula a lógica de detecção de vitória, retornando o símbolo vencedor ou *null* quando não há combinação vencedora.

```
@Injectable({ providedIn: 'root' })
export class GameService {
  private readonly WINNING_COMBINATIONS = [ /* ... */ ];
  private initialState: GameState = { /* ... */ };
  private gameStateSubject = new BehaviorSubject<GameState>(
  {...this.initialState});
  gameState$ = this.gameStateSubject.asObservable();

  makeMove(index: number): void { }
  resetGame(): void {  }
  resetScores(): void {  }
  private checkWinner(board: Player[]): Player {  }
}
```

Listagem 8.67: Trecho principal do *GameService*

A arquitetura adotada garante que toda a lógica de negócio permaneça encapsulada no *service*, enquanto os componentes apenas se inscrevem em *gameState$* e disparam métodos como *makeMove()* e *resetGame()*, promovendo reaproveitamento e testabilidade.

Capítulo 9

Engenharia de Requisitos para Web

Joana Sthefanny Gomes Costa dos Santos (UFC)

Michele Faria de Oliveira (UFC)

Evilasio Costa Junior (UFC)

Fischer Ferreira (Unifei)

Neste capítulo, serão discutidos os principais aspectos da engenharia de requisitos no contexto de aplicações web, abordando o ciclo de vida dos requisitos, as técnicas de elicitação mais utilizadas, a importância da validação e verificação, além da classificação dos requisitos em funcionais e não funcionais. Além disso, serão exploradas metodologias de priorização e ferramentas de gerenciamento que auxiliam na organização e controle dos requisitos ao longo do desenvolvimento do sistema. Essas discussões permitirão compreender como a especificação de requisitos pode ser adaptada às particularidades dos sistemas web, garantindo um processo mais eficiente e alinhado às necessidades do usuário final.

9.1 Requisitos em aplicação Web

O processo de desenvolvimento de aplicações propõe uma série de fases e atividades ao longo de seu ciclo de vida, estabelecendo um fluxo estruturado para garantir que os requisitos do sistema sejam corretamente definidos e implementados. Em geral, esse processo possui três fases principais: Definição, Desenvolvimento e Operação conforme a Figura 9.1



Figura 9.1: Fluxo de Desenvolvimento Web

Na fase de definição, é utilizada a Engenharia de Requisitos, sendo responsável por entender o funcionamento da aplicação web, como será a experiência do usuário final e como o sistema irá influenciar os negócios do cliente. O entendimento correto dos requisitos é uma etapa crucial no desenvolvimento de software, pois tem impacto direto na qualidade final do produto. Segundo Sommerville (2020), a obtenção de requisitos é definida como a descrição das funcionalidades e restrições operacionais de um sistema. Esses requisitos representam as necessidades dos clientes que buscam resolver problemas por meio do sistema, permitindo definir as restrições e características que o sistema deve apresentar.

Ademais, para atender a esses requisitos, o processo deve coletar informações sobre os requisitos de usuário e de sistema que a aplicação a ser desenvolvida deverá atender. Isso envolve identificar e documentar as necessidades, expectativas e restrições dos usuários e das partes interessadas no projeto. Essas informações são obtidas por meio da interação com as partes interessadas, que pode ocorrer por meio de reuniões, entrevistas e observações. A execução inadequada dessa fase é frequentemente responsável pelo fracasso de projetos, o que destaca a importância de conduzi-la corretamente. Esse problema pode estar relacionado à dificuldade dos profissionais em realizar essa etapa de forma adequada, o que indica falta de conhecimento necessário para sua execução.

Diferentemente dos sistemas tradicionais, as aplicações web demandam que o levantamento de requisitos considere fatores específicos desse ambiente, como a necessidade de responsividade, a integração contínua com serviços externos e práticas que favoreçam a visibilidade nos mecanismos de busca. Conforme Marcotte (MARCOTTE, 2011) e Eccher (ECCHER, 2015), requisitos relacionados ao design adaptável e à experiência do usuário tornaram-se indispensáveis, exigindo que os desenvolvedores antecipem múltiplos contextos de uso. Além disso, práticas de SEO, devem ser incorporadas ainda na definição inicial do sistema, pois afetam diretamente o alcance e a performance do produto (DOVER; DAFFORN, 2011). A integração com APIs de terceiros e a necessidade de garantir a segurança em cada troca de informações reforçam a ideia de que requisitos técnicos e não funcionais são fundamentais nesse tipo de aplicação (AMUNDSEN, 2023; JACOBSON et al., 2011). Essas particularidades evidenciam que a engenharia de requisitos, quando aplicada a sistemas web, deve ir além da elicitação tradicional, incorporando práticas capazes de lidar com mudanças rápidas, diversidade tecnológica e demandas de segurança e escalabilidade.

9.2 Engenharia de Requisitos

A engenharia de requisitos desempenha um papel crucial na jornada de um desenvolvedor web de sucesso, pois estabelece uma base sólida para qualquer projeto. Ela proporciona uma compreensão minuciosa das necessidades e expectativas do cliente, o que, por sua vez, se traduz diretamente na criação de um produto perfeitamente alinhado às demandas reais. Por outro lado, a ausência de uma

abordagem adequada de engenharia de requisitos resulta na falta de entendimento dessas necessidades, o que pode levar a um desenvolvimento desorganizado do produto. Isso, por sua vez, gera a correção de muitos erros que poderiam ter sido evitados, além de atrasos e custos adicionais consideráveis (PRESSMAN; MAXIM, 2021).

Sendo uma disciplina essencial no campo da engenharia de software e no desenvolvimento de aplicações web, a engenharia de requisitos é um estudo que se concentra em auxiliar a comunicação entre o desenvolvedor e todos os stakeholders, com o intuito de que o desenvolvedor trabalhe de forma eficiente com as informações coletadas ao longo de todas as etapas da concepção do produto, entregando qualidade e alto custo-benefício (RAMOS, 2019). Dessa forma, a engenharia de requisitos trabalha com a identificação, análise, documentação e gerenciamento de requisitos, contribuindo para um produto bem-sucedido, pois garante que as metas do projeto sejam atendidas de forma eficaz (PRESSMAN; MAXIM, 2021).

9.2.1 Planejamento e Preparação

A fase de planejamento é o estágio em que se estabelecem as bases para o sucesso de qualquer projeto web, definindo seus objetivos, escopo e direção geral. Por ser a etapa inicial, assume um papel determinante para o bom andamento do desenvolvimento, pois envolve decisões estratégicas sobre como o sistema será organizado, quais serão suas principais funcionalidades e como os diferentes componentes se relacionarão entre si. Nesse contexto, destaca-se a relevância de uma análise aprofundada da concepção do projeto, cujo propósito principal é fornecer fundamentos sólidos para o desenvolvimento de uma aplicação web.

Durante a preparação, o foco recai sobre a identificação das necessidades e expectativas do cliente, bem como sobre a definição clara dos objetivos aos quais o sistema deve atender. Para alcançar qualidade, é necessário ir além do conhecimento superficial dos usuários finais, compreendendo suas perspectivas e incorporando essa compreensão em todas as fases do desenvolvimento, de modo a criar soluções eficazes. Esse processo envolve diversas etapas, cada uma contribuindo para a definição inicial do escopo e das características da aplicação web, com impacto que se estende até a conclusão do projeto.

A comunicação eficiente é outro aspecto de grande importância em todas as etapas do levantamento de requisitos. Ela não se limita à interação entre desenvolvedores, mas também exige uma troca clara e objetiva com os *stakeholders* (RAMOS, 2019). Esse alinhamento garante que os requisitos coletados representam fielmente as necessidades do negócio e as expectativas dos usuários. Além disso, promover a colaboração contínua entre todos os membros da equipe é um fator decisivo para o sucesso do projeto.

A revisão periódica dos objetivos e o ajuste das estratégias à medida que o projeto evolui permitem que a equipe responda às mudanças de forma ágil e eficaz. O processo de levantamento de requisitos é dinâmico, e manter uma comunicação aberta contribui para que todos os envolvidos permaneçam sempre alinhados às

prioridades do projeto. Para entender melhor como essas interações se desdobram no desenvolvimento de sistemas, podemos observar o ciclo de vida dos requisitos, que organiza e estrutura cada fase desse processo de forma iterativa e adaptável.

9.3 Ciclo de vida dos requisitos

O ciclo de vida dos requisitos é um conjunto de atividades fundamentais envolvidas na gestão e no desenvolvimento de requisitos para sistemas ou aplicações. Esse ciclo é iterativo, o que significa que as etapas podem ser revisitadas e ajustadas ao longo do projeto, conforme novos requisitos surgem ou mudanças se tornam necessárias. Na Figura 9.2, é possível visualizar as principais fases desse ciclo. Cada uma dessas etapas é essencial para garantir que o sistema final atenda às expectativas dos *stakeholders*, mantendo-se dentro dos limites de tempo, orçamento e qualidade, e para permitir que o ciclo se adapte às incertezas e mudanças típicas dos projetos.



Figura 9.2: O ciclo de vida dos requisitos
Adaptado de Heath (2020)

Este ciclo garante a rastreabilidade dos requisitos, permitindo que cada mudança seja monitorada e o impacto seja avaliado. Dessa forma, o projeto se mantém

alinhado aos objetivos iniciais, com maior controle sobre a entrega final e a satisfação do cliente.

A seguir, são descritas as principais etapas do ciclo de vida dos requisitos:

- **Elicitação**: Consiste na coleta de informações sobre o que o sistema deve fazer, com base nas necessidades dos *stakeholders*. Envolve o uso de técnicas como entrevistas, questionários, observações e workshops.
- **Validação**: Busca garantir que os requisitos levantados estejam corretos, completos, consistentes e que representem fielmente as necessidades dos usuários e demais partes interessadas.
- **Modelagem**: Representa os requisitos de forma estruturada, utilizando diagramas ou modelos (como casos de uso ou diagramas de atividades), com o objetivo de facilitar a compreensão e comunicação entre os envolvidos no projeto.
- **Especificação**: Trata da descrição formal dos requisitos, tanto funcionais quanto não funcionais, com clareza, precisão e sem ambiguidades, servindo como base para o desenvolvimento do sistema.
- **Documentação**: Envolve o registro organizado dos requisitos em artefatos formais, como o Documento de Requisitos de Software (DRS), garantindo a rastreabilidade e o reuso.
- **Rastreabilidade**: Refere-se à capacidade de acompanhar cada requisito desde sua origem até sua implementação e testes, permitindo a gestão eficiente de mudanças e validação das entregas.
- **Classificação**: Os requisitos podem ser categorizados por tipo (funcional, não funcional), por prioridade (alta, média, baixa) ou por origem, facilitando a análise e organização durante o projeto.
- **Priorização**: Visa definir quais requisitos devem ser atendidos primeiro, considerando fatores como o valor de negócio, a urgência, o custo e a complexidade técnica.
- **Verificação**: Avalia se os requisitos foram corretamente implementados no sistema, por meio de revisões, inspeções e testes. Garante que o que foi desenvolvido está de acordo com o especificado.
- **Gestão de mudanças**: Trata do controle de alterações nos requisitos ao longo do projeto, incluindo a análise de impacto, a aprovação e a atualização da documentação, assegurando que o sistema se mantenha coerente e alinhado aos objetivos.

9.3.1 Coleta de Informações (Elicitação)

Na fase de coleta de informações, é fundamental realizar uma análise prévia bem estruturada, dado que esse planejamento deve considerar que o tempo é um recurso limitado. Essa preparação é particularmente crucial ao planejar estratégias para interagir eficazmente com *stakeholders*, garantindo que todas as expectativas e necessidades sejam compreendidas e adequadamente incorporadas ao desenvolvimento do projeto (RAMOS, 2019).

A identificação das necessidades do usuário, por sua vez, é um aspecto crítico que não pode ser negligenciado, no qual o envolvimento dos *stakeholders* deve ser cuidadosamente planejado. Eles fornecem informações que ajudam a moldar o desenvolvimento web, tornando-o funcional, eficiente e alinhado com as expectativas de todos os envolvidos. Logo, profissionais como analistas de negócios e engenheiros de requisitos desempenham um papel crucial nesse processo, colaborando e integrando elementos específicos de suas respectivas áreas de atuação.

A elicitação de requisitos é um passo que ajuda a entender as necessidades dos *stakeholders*, garantindo que o sistema desenvolvido atenda às expectativas. A Figura 9.3 mostra um mapa mental com diferentes categorias de Técnicas de Elicitação de Requisitos, que são ferramentas usadas para identificar e capturar os requisitos necessários para o desenvolvimento de um sistema ou aplicação, seguido por uma breve explicação de cada uma delas:

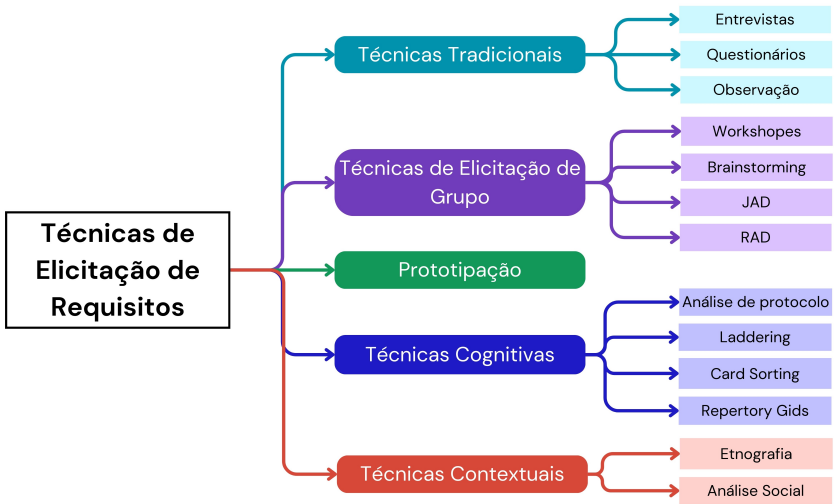


Figura 9.3: Técnicas de Elicitação de Requisitos
Adaptado de Rego (2023)

- **Técnicas Tradicionais:**

  - **Entrevistas:** Consistem em uma conversa estruturada ou semiestruturada com os *stakeholders* para identificar necessidades e expectativas do sistema.
  - **Questionários:** Um método de coleta de dados em larga escala, no qual perguntas padronizadas são feitas aos *stakeholders* para capturar requisitos.
  - **Observação:** Consiste em observar como os usuários interagem com o sistema ou realizam suas tarefas, a fim de entender requisitos não verbalizados.

- **Técnicas de Elicitação de Grupo:**
  - **Workshops:** Sessões colaborativas em que *stakeholders* e a equipe de desenvolvimento se reúnem para discutir e identificar os requisitos do sistema.
  - **Brainstorming:** Método criativo de geração de ideias, no qual os participantes propõem soluções e identificam requisitos sem restrições iniciais.
  - **JAD (Joint Application Development):** Reuniões focadas na colaboração entre os *stakeholders* e os desenvolvedores, facilitadas para a captura eficiente de requisitos.
  - **RAD (Rapid Application Development):** Técnica de desenvolvimento rápido que envolve ciclos curtos de *feedback* e prototipagem para ajustar os requisitos.

- **Prototipação:** A criação de protótipos do sistema permite que os *stakeholders* interajam com uma versão inicial e forneçam *feedback*, o que ajuda a ajustar e refinar os requisitos.

- **Técnicas Cognitivas:**
  - **Análise de protocolo:** O usuário descreve em voz alta o que está fazendo durante uma tarefa, o que permite extrair requisitos com base nas suas ações.
  - **Laddering:** Um método de entrevistas que explora as cadeias de raciocínio do usuário para entender seus valores e como esses valores influenciam os requisitos.
  - **Card Sorting:** Técnica onde os usuários organizam cartões com termos ou funcionalidades, ajudando a identificar como categorizam informações e requisitos.
  - **Repertory Grids:** Uma técnica para comparar diferentes elementos do sistema e entender as percepções dos usuários sobre eles, o que resulta em requisitos mais claros.

- **Técnicas Contextuais:**
  - **Etnografia:** Envolve a imersão no ambiente de trabalho dos usuários para observar como interagem com suas tarefas diárias, o que ajuda a identificar requisitos implícitos.
  - **Análise Social:** Examina as interações sociais e o contexto em que o sistema será utilizado, ajudando a capturar requisitos relacionados à dinâmica social e de trabalho.

9.3.2 Verificação e Validação dos Requisitos

A validação de requisitos garante que as especificações atendam às necessidades dos *stakeholders* e estejam livres de falhas que possam comprometer a entrega do produto final, sendo essencial verificar se a comunicação entre as partes envolvidas é eficaz e se todos os requisitos foram corretamente compreendidos e documentados (SOMMERVILLE, 2020). A validação envolve técnicas e processos que ajudam a identificar e corrigir problemas antes que se tornem mais difíceis de resolver, o que resulta em custos elevados. A seguir, a Figura 9.4 ilustra um processo de verificação e validação de requisitos em uma aplicação web, que envolve diferentes papéis na área de desenvolvimento: Designer de Interface/UX, Desenvolvedores, Analista de Requisitos e Testador de Qualidade (QA).

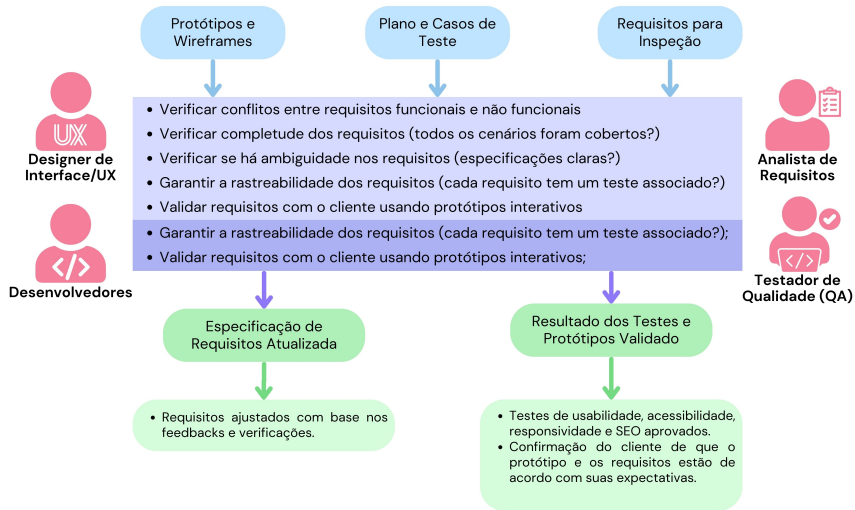


Figura 9.4: Exemplo de Verificação e Validação de Requisitos

O processo inicia-se com a elaboração de protótipos e *wireframes*, utilizados para identificar conflitos entre requisitos funcionais e não funcionais, avaliar a completude das especificações, detectar ambiguidades e garantir a rastreabilidade (assegurando

que cada requisito tenha um teste associado). Nessa etapa, a validação com o cliente também ocorre por meio de protótipos interativos. Em seguida, são definidos planos e casos de teste, que reforçam a verificação contínua da rastreabilidade e a validação dos requisitos. Paralelamente, ocorre a etapa de inspeção, na qual analistas e QA avaliam a consistência, a clareza e a testabilidade dos requisitos. Os resultados dessas atividades se traduzem em dois produtos principais:

- **Especificação de Requisitos Atualizada**: requisitos ajustados com base nos feedbacks e nas verificações.
- **Resultado dos Testes e Protótipos Validados**: confirmação de que os testes de usabilidade, acessibilidade, responsividade e SEO foram aprovados e de que o cliente validou que o protótipo atende às suas expectativas.

Esse fluxo iterativo assegura que os requisitos permaneçam claros, rastreáveis e alinhados às necessidades do projeto, contribuindo para a qualidade final do sistema. Dessa forma, a fase de validação de requisitos visa garantir que a especificação atenda às expectativas do cliente e reflita fielmente suas necessidades. A revisão de modelos ou descrições textuais de requisitos é uma técnica comum na indústria, sendo fundamental para garantir a correção, a completude e a consistência da especificação. Estima-se que até 60% (SUKUMARAN et al., 2006) dos defeitos encontrados em sistemas possam ser atribuídos a problemas no levantamento de requisitos, como informações incompletas ou inconsistentes. Portanto, uma validação adequada é crítica, especialmente para sistemas que precisam atender a altos padrões de confiabilidade e certificação (DARGHAM; SEMAAN, 2008).

9.3.3 Modelagem

A modelagem de requisitos, tanto funcionais quanto não funcionais, é um processo que envolve revisões contínuas ao longo do ciclo de vida do projeto. Esse processo abrange a definição e a estruturação dos serviços e das funcionalidades esperados de um sistema. A modelagem deve ser bem estruturada, especialmente em sistemas mais complexos, como os serviços médicos baseados na web (ANZBÖCK; DUSTDAR, 2005). Outro ponto importante é o uso de padrões já estabelecidos, como, por exemplo, o UML *(Unified Modeling Language)*, uma linguagem de modelagem visual amplamente utilizada na engenharia de software para especificar, visualizar, construir e documentar os artefatos de sistemas complexos, e o BPEL *(Business Process Execution Language)*, uma linguagem padrão usada para a orquestração e automação de processos de negócios baseados em serviços web (PATHAK et al., 2006).

A Figura 9.5 representa o processo de Especificação e Modelagem de Requisitos, destacando as principais etapas e artefatos envolvidos na definição detalhada de requisitos para um projeto de desenvolvimento web. A seguir, descrevemos os principais elementos do fluxo de trabalho do Analista de Requisitos, destacando suas funções e os artefatos gerados ao longo do processo. Como ilustrado no

diagrama, o fluxo se inicia com a definição de Diretrizes de Negócio (em azul claro, à esquerda), que especificam o comportamento esperado da aplicação web; um Documento de Visão (em azul claro, à direita), que detalha os objetivos do projeto, focando em aspectos de usabilidade, escalabilidade e compatibilidade; e o Glossário (em azul claro, ao centro), que define termos técnicos e conceitos-chave relacionados à experiência do usuário e tecnologias envolvidas. O Analista de Requisitos tem o papel (destacado em roxo) de:

- Elicitar Requisitos Funcionais e Não Funcionais;
- Especificar e validar casos de uso ou histórias de usuário;
- Especificar Requisitos Não Funcionais;
- Analisar e Validar Requisitos.

Essas atividades resultam em três principais artefatos (destacados em verde-claro, que são descritos logo abaixo):

- Casos de Uso ou Histórias de Usuário e Especificações Suplementares (à esquerda);
- Plano e Casos de Teste (ao centro);
- Requisitos para Inspeção (à direita);

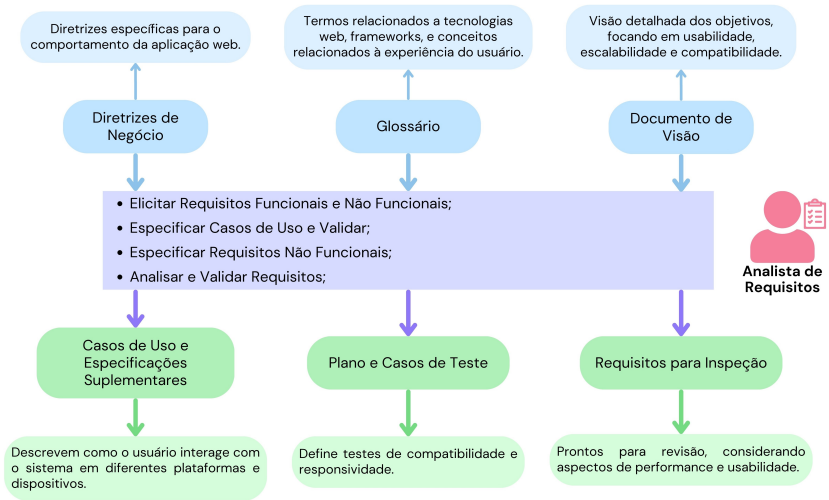


Figura 9.5: Especificação e Modelagem de Requisitos
Adaptado Lombardo ()

Esse diagrama mostra como as Diretrizes de Negócio, o Glossário e o Documento de Visão (todos em azul claro) se transformam em um conjunto claro de requisitos que podem ser validados e implementados como funcionalidades no sistema, assegurando que todas as partes envolvidas compartilhem uma visão comum dos objetivos e funcionalidades do software.

A Figura 9.6 apresenta um mapa mental de Modelagem de Requisitos, descrevendo as etapas que se seguem à fase de levantamento de requisitos. Ela é dividida em seis principais áreas:

- **Especificação de Requisitos**: Divide-se em requisitos funcionais e não funcionais. Requisitos funcionais descrevem as funcionalidades que o sistema deve oferecer, enquanto requisitos não funcionais se referem a restrições e qualidades do sistema, como desempenho e segurança.

- **Caso de Uso**: Representa como os usuários interagem com o sistema. Um caso de uso descreve o comportamento do sistema do ponto de vista de um usuário, ajudando a entender os requisitos de interação. A seta verde que conecta os "Casos de Uso"aos "Requisitos Funcionais", indica que os casos de uso são derivados diretamente dos requisitos funcionais, ou seja, cada funcionalidade especificada deve refletir uma interação entre o usuário e o sistema.

- **Objetivos**: Aqui, são definidos os principais objetivos do sistema, que orientam a modelagem e desenvolvimento, assegurando que as expectativas das partes interessadas sejam atendidas. A seta verde que conecta "Objetivos"a "Especificação de Requisitos"ressalta que os objetivos definidos no início do projeto são essenciais para a especificação de requisitos, garantindo que os requisitos estejam alinhados às metas do projeto.

- **Métricas de Requisitos**: Determinam como medir o sucesso ou a completude dos requisitos, garantindo que sejam verificáveis e atendam às expectativas e necessidades estabelecidas.

- **Gestão**: Foca na gestão dos requisitos ao longo do ciclo de vida do projeto. A gestão eficiente assegura que mudanças e revisões nos requisitos sejam controladas e documentadas adequadamente.

- **Técnicas de Levantamento**: Estas são as técnicas utilizadas para coletar os requisitos iniciais. Incluem métodos como análise de documentos, entrevistas, reuniões, observações, dentre outros, que garantem que todas as necessidades dos stakeholders sejam capturadas corretamente.

O objetivo da modelagem de requisitos, conforme descrito na figura, é consolidar e detalhar todos os aspectos do sistema, garantindo que cada requisito seja compreendido e validado antes da implementação. Este processo é crucial para assegurar que o produto final esteja alinhado às expectativas dos *stakeholders*. Dessa

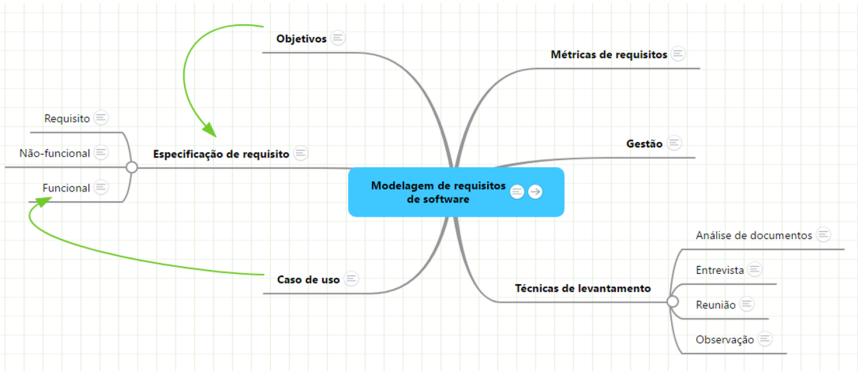


Figura 9.6: Mapa mental de Modelagem de Requisitos
Adaptado de Guerra (2009)

forma, conclui-se que a modelagem de requisitos não deve se limitar às funcionalidades básicas, mas também deve considerar aspectos críticos, como segurança, eficiência e a evolução contínua dos requisitos ao longo do desenvolvimento da aplicação, que são essenciais para garantir a interoperabilidade e a confiabilidade do sistema (ANZBÖCK; DUSTDAR, 2005).

9.3.4 Documentação dos requisitos

A documentação de requisitos desempenha um papel fundamental no sucesso de um projeto, pois garante que as necessidades identificadas durante a elicitação sejam registradas de forma clara, organizada e acessível a todos os envolvidos no desenvolvimento. Além de formalizar as expectativas do cliente, ela atua como um guia contínuo ao longo do ciclo de vida do sistema, reduzindo ambiguidades e minimizando o risco de retrabalho (SOMMERVILLE, 2020).

Diversas abordagens podem ser utilizadas para representar e organizar esses requisitos, variando de modelos mais simples, como listas e descrições textuais, a técnicas visuais que facilitam o entendimento e a validação com os *stakeholders*. Entre as práticas mais adotadas no contexto do desenvolvimento web estão as histórias de usuários, que descrevem funcionalidades sob a perspectiva do usuário final, e os diagramas de casos de uso, que oferecem uma visão estruturada das interações entre usuários e o sistema. A seguir, serão apresentadas essas duas técnicas, destacando suas características, objetivos e como contribuem para a documentação dos requisitos.

9.4 História de usuários

Uma das abordagens mais utilizadas para facilitar a documentação do levantamento de requisitos no processo de desenvolvimento é o uso de "histórias de usuário", que representam cenários baseados nas experiências e necessidades do usuário final, com o cliente colaborando diretamente com a equipe de desenvolvimento para criar "cartões de história", que descrevem, de forma resumida, um cenário relevante para o usuário. Esses cartões são utilizados pelo time de desenvolvimento para planejar e implementar funcionalidades em versões futuras do sistema. Durante o planejamento das iterações, as histórias de usuário são detalhadas e divididas em tarefas, com o objetivo de mensurar o esforço e os recursos necessários à sua implementação. Além disso, as histórias são priorizadas pelo cliente, que decide quais delas trarão mais valor imediato ao negócio (SOMMERVILLE, 2020).

A Figura 9.7 ilustra o fluxo de organização do desenvolvimento de funcionalidades. Esse processo é dividido em diferentes níveis, começando pela Iniciativa, que representa um objetivo do projeto e, posteriormente, será dividida em Épicos, grandes blocos de funcionalidades relacionados ao objetivo principal. Cada épico engloba uma série de histórias de usuário que descrevem funcionalidades específicas a serem implementadas para atender às necessidades dos usuários. Em cada história, são identificadas subtarefas, que correspondem a ações ou atividades específicas necessárias à conclusão da história. Essas subtarefas facilitam o planejamento e a execução de cada funcionalidade de forma mais organizada e eficiente.

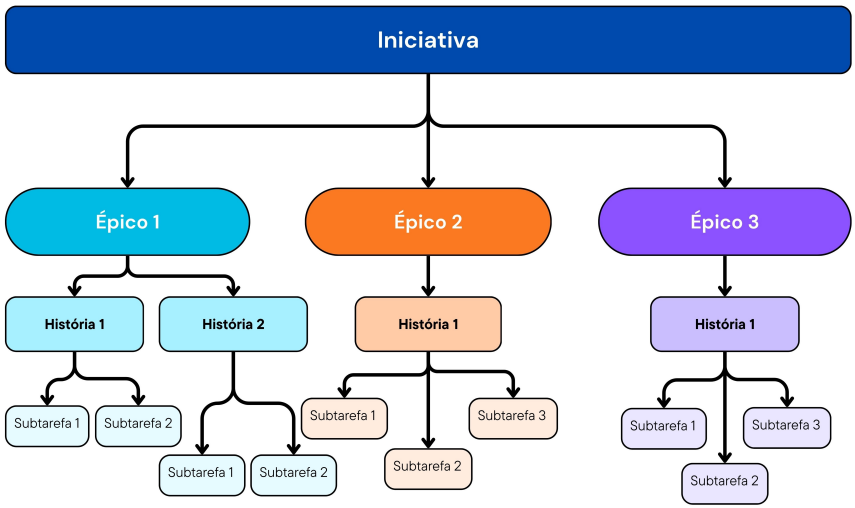


Figura 9.7: Modelo de histórias de usuários

A Figura 9.8 apresenta a estrutura das histórias de usuário no contexto da construção de uma plataforma de *e-commerce*. O fluxo segue a organização de

Iniciativa > Épico > Histórias de Usuário > Subtarefas, similar ao modelo da figura anterior, porém com um exemplo aplicado, detalhando como uma iniciativa pode ser dividida em partes menores e mais gerenciáveis, facilitando o planejamento e a execução do projeto. A iniciativa principal é a "Construção de uma plataforma de *e-commerce* completa". A partir dessa iniciativa, surgem três grandes Épicos:

- **Gerenciamento de Produtos**: Como administrador, o objetivo é adicionar novos produtos com descrição, preço e imagens para que os clientes possam visualizá-los e comprá-los.
- **Processamento de Pagamentos**: Como cliente, a meta é adicionar itens ao carrinho e efetuar o pagamento para finalizar a compra.
- **Integração com Sistema de Entrega**: Como cliente, o objetivo é escolher a opção de entrega que melhor atenda às suas necessidades.

Cada épico é dividido em histórias de usuário. Por exemplo, no Gerenciamento de Produtos, uma história de usuário pode descrever o desejo do administrador de adicionar produtos corretamente, enquanto, no Processamento de Pagamentos, a história do cliente envolve a necessidade de adicionar itens ao carrinho e finalizar a compra por meio de APIs de pagamento. No nível seguinte, surgem as subtarefas, que detalham as ações necessárias para atender a essas histórias. Por exemplo, no Gerenciamento de Produtos, devem ser incluídas a implementação de validações e a criação de um formulário para inserção de produtos.

9.5 Casos de uso

O diagrama de casos de uso desempenha um papel fundamental na modelagem do escopo inicial de um projeto, com o objetivo de descrever quem realiza determinadas ações no sistema e estabelecer uma representação visual das interações entre os atores e os eventos que requerem resposta do sistema. Cada caso de uso está associado a uma tarefa que beneficia o ator e é de sua responsabilidade, destacando os resultados entregues pelo sistema (SOMMERVILLE, 2020). Esse tipo de diagrama é útil para definir as interações principais e as entidades envolvidas, mas não se aprofunda em detalhes como a sequência de passos necessários, as regras de negócio aplicáveis ou a sequência de casos de uso que compõem um processo de negócio completo (VAZQUEZ; SIMõES, 2020).

Além de descrever as interações principais entre o sistema e seus atores, os casos de uso também são ferramentas eficazes para identificar macroprocessos que englobam objetivos colaborativos, mas que ainda não têm escopo claramente definido. À medida que o desenvolvimento avança e as decisões sobre quais processos serão automatizados são tomadas, os casos de uso devem ser constantemente atualizados para refletir o novo escopo do sistema. Essa evolução garante que o diagrama permaneça relevante e alinhado aos requisitos do projeto, permitindo uma visão

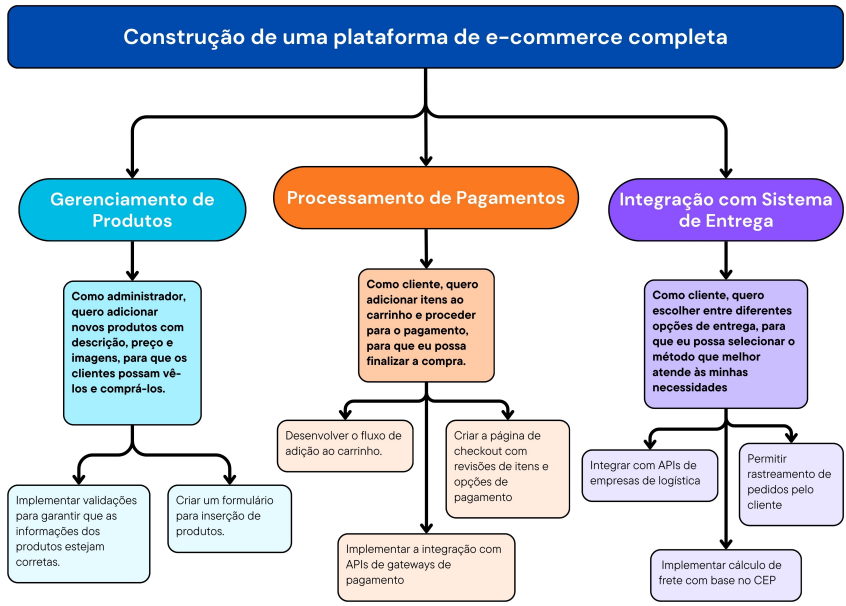


Figura 9.8: Exemplo de histórias de usuário

progressiva das funções a serem implementadas. Portanto, a qualidade do diagrama de casos de uso está diretamente relacionada ao momento em que ele é produzido e atualizado, sendo fundamental que ele acompanhe o ciclo de vida do software e as decisões que impactam o escopo do sistema (VAZQUEZ; SIMõES, 2020).

O diagrama da Figura 9.9 mostra como usuários e administradores de um site interagem com as diferentes funcionalidades. Ele usa os relacionamentos `include` para indicar que algumas ações dependem de outras (como baixar um documento após pesquisá-lo) e `extend` para indicar que algumas ações são uma extensão de funcionalidades semelhantes realizadas por outro ator (por exemplo, o upload de documentos por um usuário ou por um administrador do site).

9.5.1 Rastreabilidade (*traceability*)

O termo rastreabilidade refere-se à capacidade de vincular requisitos a todos os artefatos gerados ao longo do desenvolvimento de uma aplicação. Isso inclui a relação entre documentos de requisitos, especificações, modelos de análise e de design, código-fonte, casos de teste e outros artefatos produzidos ao longo do ciclo de vida do desenvolvimento (JIRAPANTHONG, 2015). Em aplicações web, os requisitos tendem a evoluir ao longo do tempo devido a diversos fatores, como o surgimento de novas necessidades, o que torna o processo de atualização de requisitos difícil. Neste contexto, a rastreabilidade é essencial, pois permite avaliar o impacto de

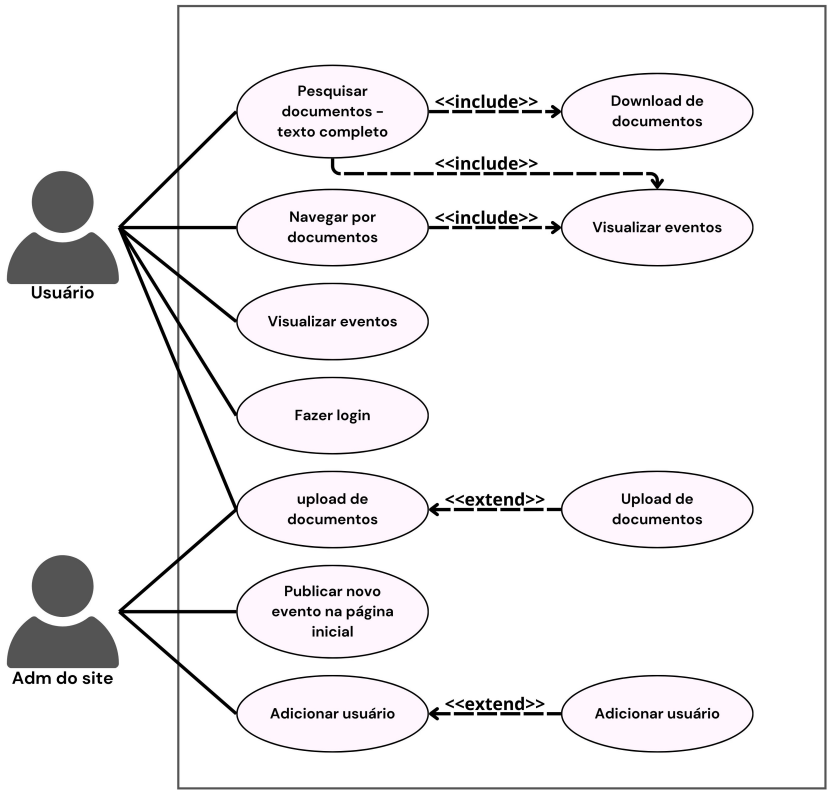


Figura 9.9: Diagrama de caso de uso de um site

mudanças e facilitar as atividades de manutenção. O uso de rastreabilidade tem sido amplamente reconhecido como uma característica fundamental de uma aplicação bem projetada, contribuindo para a qualidade e a eficiência no desenvolvimento e na evolução contínua de aplicações web (GARCIA; PAIVA, 2016).

Basicamente, rastrear significa estabelecer um vínculo entre essas informações para garantir que os requisitos possam ser monitorados e gerenciados ao longo do projeto. A rastreabilidade é essencial para qualquer tipo de desenvolvimento, pois uma grande quantidade de informações é utilizada e gerada, e todas elas precisam ser mantidas relacionadas para garantir a consistência e o controle do projeto (PINHEIRO, 2004).

De acordo com a norma IEEE 29148:2011, um requisito é considerado rastreável quando é possível documentar sua origem e acompanhar seu progresso, garantindo que atenda aos objetivos dos *stakeholders* e que seu gerenciamento ao longo do ciclo de vida seja facilitado. Dessa forma, a rastreabilidade dos requisitos deve

ser estabelecida e mantida para documentar como os requisitos foram projetados para atender aos objetivos dos *stakeholders* (INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS (IEEE), 2011).

No contexto do desenvolvimento web, a rastreabilidade ajuda a garantir que os requisitos de *stakeholders* coletados na fase de levantamento sejam continuamente monitorados e atualizados. O uso de ferramentas de gerenciamento de requisitos facilita esse processo, garantindo que os requisitos estejam associados aos elementos arquitetônicos do sistema, às interfaces e aos critérios de verificação e validação. À medida que a aplicação avança pelas fases do ciclo de vida, a rastreabilidade também deve ser mantida, assegurando que cada requisito tenha um identificador único para facilitar seu controle e acompanhamento (GARCIA; PAIVA, 2016).

9.6 Classificação dos requisitos

Após a identificação das necessidades e expectativas dos *stakeholders*, o próximo passo é classificar essas necessidades em dois tipos principais de requisitos: funcionais e não funcionais.

9.6.1 Requisitos Funcionais

Os requisitos funcionais estão diretamente ligados aos serviços que o sistema deve oferecer, ou seja, definem as funcionalidades específicas que o software precisa executar para satisfazer as necessidades dos usuários e atingir os objetivos do projeto. Esses requisitos são fundamentais para assegurar que o sistema atenda às expectativas dos usuários e cumpra seu propósito. Além disso, eles fornecem uma orientação clara para o desenvolvimento e a testagem do sistema, permitindo que as equipes de projeto compreendam com precisão o que deve ser implementado e validado (SOMMERVILLE, 2020). Na Figura 9.10 é possível observar alguns exemplos de requisitos funcionais voltados para desenvolvimento web:

9.6.2 Requisitos Não Funcionais

Os requisitos não funcionais, também conhecidos como atributos de qualidade, são características fundamentais que definem o comportamento do sistema além de suas funcionalidades básicas. Eles abrangem fatores como segurança, desempenho, confiabilidade e escalabilidade,e influenciam diretamente as decisões de design e arquitetura (SOMMERVILLE, 2020). Ignorar esses requisitos pode gerar custos elevados, pois alterações após a implementação do sistema podem ser complexas. Por exemplo, a decisão de modularizar um sistema para facilitar sua manutenção pode impactar negativamente o desempenho. Portanto, é fundamental priorizar esses requisitos nas fases iniciais do desenvolvimento para evitar conflitos futuros (ANTONIO, 2008). Na Figura 9.11 é possível observar alguns exemplos de requisitos não funcionais voltados para desenvolvimento web:

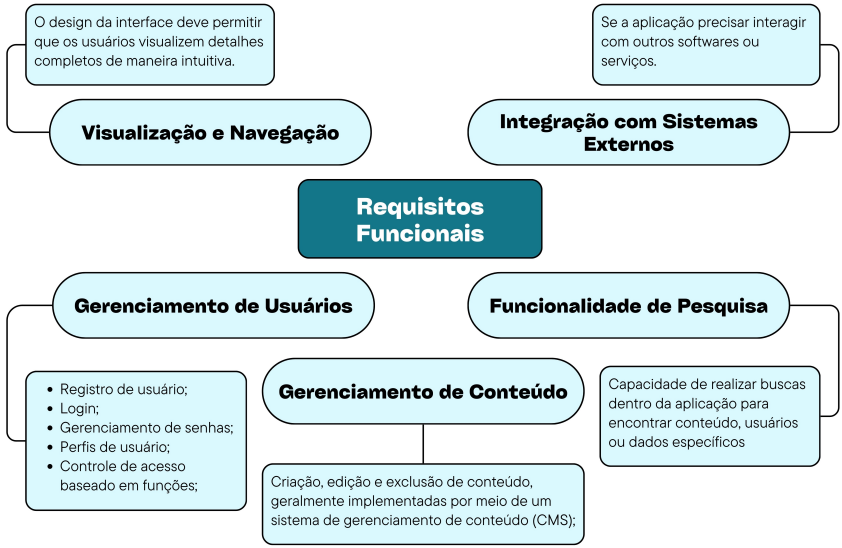


Figura 9.10: Diagrama de caso de uso de um site

9.7 Requisitos de sistemas comuns em aplicações Web

Além dos requisitos funcionais e não funcionais já discutidos, existem outros aspectos que são frequentemente encontrados no desenvolvimento de aplicações web. Esses requisitos estão relacionados a funcionalidades específicas que garantem a usabilidade, o desempenho e a segurança das aplicações, além de atender às expectativas dos usuários. A seguir, serão apresentados alguns desses requisitos comuns:

9.7.1 Responsividade e Adaptação a Diferentes Telas

O design responsivo é um dos principais aspectos a serem considerados no desenvolvimento web. Conforme descrito por Marcotte (2011), um layout responsivo permite que uma aplicação se adapte a diferentes tamanhos de tela, proporcionando uma experiência consistente para usuários em desktops, tablets e dispositivos móveis. Essa abordagem é baseada em três pilares fundamentais:

- **Grades Flexíveis**: O uso de unidades relativas, como porcentagens em vez de pixels fixos, garante que os elementos da interface se ajustem dinamicamente ao espaço disponível.



Figura 9.11: Exemplos de Requisitos Não Funcionais em uma aplicação web

- **Imagens Flexíveis**: A aplicação de propriedades como `max-width: 100%` impede que as imagens ultrapassem os limites dos contêineres, mantendo a proporção adequada em telas menores.
- **Media Queries**: Permitem definir regras de estilo específicas para diferentes tamanhos de tela, garantindo que a interface seja ajustada conforme necessário.

```
@media screen and (max-width: 768px) {
    body {
        font-size: 100%;
    }
}
```

Listagem 9.1: Consulta de mídia para responsividade.

A Listagem 9.1 apresenta um exemplo simples de *media query* em CSS, utilizado para tornar uma aplicação web responsiva. Nesse caso, a regra `@media screen and (max-width: 768px)` define que os estilos dentro desse bloco serão aplicados apenas quando a largura da tela do dispositivo for igual ou inferior a 768 pixels, valor comumente associado a tablets e dispositivos móveis. Dentro do bloco, o seletor `body` ajusta o tamanho da fonte para `100%`, garantindo que o texto seja legível em

telas menores. Essa abordagem permite que o layout se adapte dinamicamente a diferentes resoluções, melhorando a experiência do usuário em dispositivos variados.

Eccher (2015) reforça a importância do uso de frameworks como **Bootstrap e Tailwind CSS**, bem como tecnologias nativas como **CSS Grid e Flexbox**, para estruturar layouts flexíveis e responsivos. Essas ferramentas facilitam a organização dos elementos da página e melhoram a compatibilidade entre diferentes dispositivos. Além disso, a definição de *breakpoints* adequados é essencial para evitar problemas como a sobreposição de elementos e a necessidade de rolagem horizontal (MARCOTTE, 2011). Um design responsivo bem estruturado melhora não apenas a experiência do usuário, mas também o desempenho e a acessibilidade da aplicação. Por fim, garantir a responsividade de uma aplicação exige testes em múltiplos dispositivos e resoluções, de forma a identificar possíveis inconsistências e otimizar a apresentação do conteúdo (ECCHER, 2015; MARCOTTE, 2011) .

9.7.2 SEO (Seach Engine Optimization)

A indexação de páginas é essencial para que um site apareça nos resultados dos mecanismos de busca. Isso significa que os buscadores precisam encontrar o site, entender seu conteúdo e armazená-lo para exibição quando alguém fizer uma pesquisa relacionada. Para facilitar esse processo, algumas boas práticas são recomendadas (BRASIL, 2020; ENGE et al., 2022):

- **Estruturação de títulos:** O site deve ter títulos bem organizados, com cabeçalhos estruturados, como `H1` e `H2`, para facilitar a leitura do conteúdo pelos mecanismos de busca.
- **Uso do sitemap.xml:** Criar um arquivo chamado `sitemap.xml` e enviá-lo ao Google Search Console, garantindo que os buscadores consigam encontrar e indexar todas as páginas do site;
- **Configuração do robots.txt:** Evitar bloqueios desnecessários no arquivo `robots.txt`, pois ele pode impedir que os mecanismos de busca acessem certas partes do site e prejudiquem a indexação.

O arquivo `sitemap.xml` é essencial para a indexação de sites nos mecanismos de busca, pois funciona como um "mapa"que lista as páginas mais relevantes, facilitando o rastreamento pelos buscadores. O Google e outras ferramentas de pesquisa utilizam esse arquivo para localizar e indexar páginas de forma mais eficiente, principalmente em sites grandes ou com conteúdos que não estão diretamente acessíveis pelo menu principal (DOVER; DAFFORN, 2011). Para otimizar o processo, é recomendável manter o `sitemap.xml` sempre atualizado e, no caso de sites grandes, dividi-lo em múltiplos arquivos para melhorar a leitura pelos mecanismos de busca (DOVER; DAFFORN, 2011; BRASIL, 2020). Sua estrutura segue um formato padronizado em XML, conforme exemplificado no Código-Fonte 9.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
   <url>
      <loc>https://www.exemplo.com/</loc>
      <lastmod>2024-03-19</lastmod>
      <priority>1.0</priority>
   </url>
   <url>
      <loc>https://www.exemplo.com/produtos</loc>
      <lastmod>2024-03-19</lastmod>
      <priority>0.8</priority>
   </url>
</urlset>
```

Listagem 9.2: Exemplo de sitemap.xml

- **Declaração XML:** A primeira linha (`<?xml version="1.0"encoding="UTF-8"?>`) define a versão do XML e a codificação dos caracteres, garantindo compatibilidade com diferentes idiomas e símbolos especiais.
- **Elemento raiz (`<urlset>`):** Abrange todas as URLs do sitemap e utiliza um namespace padronizado (`xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"`) para indicar que o arquivo segue o protocolo oficial de Sitemaps.
- **Tag `<url>`:** Cada bloco `<url>` representa uma página do site a ser indexada pelos mecanismos de busca.
- **Tag `<loc>`:** Defina a URL da página a ser indexada. No exemplo, foram listadas duas URLs: a página inicial (`https://www.exemplo.com/`) e a página de produtos (`https://www.exemplo.com/produtos`).
- **Tag `<lastmod>`:** Indica a última data de modificação da página. Isso ajuda os motores de busca a identificar se o conteúdo foi atualizado recentemente.
- **Tag `<priority>`:** Define a prioridade da página para os mecanismos de busca. Os valores variam de 0.0 (menos importante) a 1.0 (mais importante). No exemplo, a página principal tem prioridade 1.0, enquanto a página de produtos tem 0.8.
- **Encerramento do arquivo:** O `</urlset>` fecha o documento, indicando o fim do sitemap.

O `robots.txt` é um arquivo de configuração que informa aos mecanismos de busca quais páginas do site podem ou não ser rastreadas. Ele deve ser colocado na raiz do site (ex: `https://www.exemplo.com/robots.txt`) e segue um formato padronizado para definir permissões de acesso (DOVER; DAFFORN, 2011). Embora o `robots.txt` seja útil para restringir o rastreamento, ele não impede que páginas sejam indexadas caso outros sites apontem links para elas. Para evitar a indexação

completa, recomenda-se o uso da meta tag `noindex` diretamente no HTML da página (DOVER; DAFFORN, 2011). Um exemplo básico de um arquivo `robots.txt` pode ser visto na Listagem 9.3:

```
User-agent: *
Disallow: /admin/
Disallow: /private/
Allow: /public/
Sitemap: https://www.exemplo.com/sitemap.xml
```

Listagem 9.3: Exemplo de robots.txt

- `User-agent: *`: Significa que as regras valem para todos os mecanismos de busca.
- `Disallow: /admin/`: Impede que os buscadores acessem a pasta "admin".
- `Allow: /public/`: Permite acesso à pasta "public".
- `Sitemap: https://www.exemplo.com/sitemap.xml`: Indica a localização do sitemap do site.

Além disso, é essencial que as páginas do site tenham **URLs amigáveis**, ou seja, endereços fáceis de entender. Em vez de uma URL confusa, cheia de números e caracteres estranhos, o ideal é que ela descreva claramente o conteúdo da página. Por exemplo, uma URL como `www.exemplo.com/produto/tenis-esportivo` é muito melhor do que `www.exemplo.com/p=123?ref=xyz`. Para isso, é importante usar palavras-chave relevantes, separar palavras com hífens e evitar caracteres especiais e acentos (DOVER; DAFFORN, 2011; BRASIL, 2020).

Outro ponto fundamental para otimizar um site para os buscadores é o uso de **meta tags**, que são informações adicionadas no código da página para ajudar os mecanismos de busca a entenderem melhor o seu conteúdo. Uma dessas tags é a `meta description`, que aparece nos resultados de pesquisa e deve ser curta e descrever bem o que a página oferece. Usar palavras-chave nessa descrição pode aumentar as chances de mais pessoas clicarem no link. Além disso, recomenda-se configurar `meta titles` (títulos da página) e `alt text` nas imagens, o que ajuda tanto na indexação quanto na acessibilidade para usuários com deficiência visual (ENGE et al., 2022; BRASIL, 2020).

Por fim, um detalhe técnico que pode melhorar a visibilidade do site nos mecanismos de busca é a implementação de *Server-Side Rendering (SSR)*. Em muitos sites modernos, o conteúdo é carregado pelo JavaScript no navegador do usuário, o que pode dificultar a leitura pelos buscadores. O SSR resolve esse problema ao gerar as páginas no servidor antes de enviá-las ao usuário, garantindo que todo o conteúdo já esteja disponível para indexação adequada. Isso melhora não apenas a indexação, mas também a velocidade do site, o que é um fator importante para o ranqueamento no Google (ENGE et al., 2022).

9.7.3 Integração com APIs de terceiros

Muitas aplicações web precisam se conectar a serviços externos, como mapas interativos, sistemas de pagamento ou redes sociais. Isso é possível graças à **integração com APIs de terceiros**, que permite que um site ou aplicativo troque informações com outros sistemas de forma segura e eficiente. Para fazer isso corretamente, é importante entender alguns conceitos essenciais, como os tipos de APIs disponíveis, como proteger essas conexões e como evitar problemas ao usá-las.

A comunicação entre sistemas pode ocorrer de diferentes formas, sendo as mais comuns as APIs *REST* e *GraphQL*. O modelo *REST* utiliza um conjunto padronizado de métodos HTTP, como GET, POST, PUT e DELETE, permitindo a troca estruturada de informações entre serviços. Já o *GraphQL* permite que o cliente especifique exatamente quais dados deseja receber, reduzindo a transferência de dados desnecessários e tornando as consultas mais eficientes (AMUNDSEN, 2023).

Como essas APIs podem lidar com informações sensíveis, como dados de pagamento ou dados pessoais dos usuários, é essencial garantir que apenas pessoas e sistemas autorizados possam acessá-las. Para isso, existe o *OAuth 2.0*, um sistema que permite que aplicativos se conectem a serviços externos sem precisar armazenar senhas. Um exemplo prático de sua aplicação ocorre em plataformas de e-commerce, onde os usuários podem realizar login utilizando suas contas do Google ou Facebook. Em vez de solicitar diretamente o e-mail e a senha do usuário, o site redireciona para a página do provedor de autenticação, onde o usuário concede permissão de acesso às suas informações. O provedor, então, envia um *token* de acesso temporário, permitindo que o e-commerce autentique o usuário sem armazenar suas credenciais (JACOBSON et al., 2011).

Além da autenticação, o *OAuth 2.0* também é amplamente utilizado para pagamentos online. Um e-commerce pode, por exemplo, permitir que os clientes utilizem o PayPal ou Stripe para realizar compras. Dessa forma, em vez de inserir os dados do cartão diretamente no site, o usuário é redirecionado para a plataforma de pagamento, onde a transação é realizada com segurança. Esse modelo reduz os riscos de vazamento de informações financeiras e melhora a confiabilidade do sistema (JACOBSON et al., 2011).

Outro desafio na integração de APIs de terceiros é garantir que elas não sejam sobrecarregadas. Se um grande número de usuários fizer muitas requisições simultaneamente, a API pode ficar lenta ou até parar de funcionar. Para evitar isso, muitas APIs utilizam uma técnica chamada *Rate Limiting*, que limita o número de requisições que um usuário pode fazer em um determinado período. Um exemplo disso é o Twitter, que restringe o número de verificações de timeline permitidas por hora. Outras estratégias incluem o *Spike Arresting*, que impede aumentos súbitos de acessos que possam prejudicar o sistema, e o *Throttling*, que reduz a velocidade das requisições em vez de bloqueá-las completamente (AMUNDSEN, 2023). Por fim, erros podem ocorrer ao se comunicar com APIs de terceiros, e é fundamental que o sistema saiba lidar com essas situações de forma adequada. Algumas estratégias importantes incluem:

- ***Mensagens de erro padronizadas*:** Permitem que o sistema informe claramente o que ocorreu e como o problema pode ser resolvido.
- ***Retry Strategy*:** Faz com que o sistema tente novamente caso a requisição falhe temporariamente, evitando falhas decorrentes de instabilidades momentâneas.
- ***Fallback Mechanism*:** Caso a API principal esteja fora do ar, o sistema pode recorrer a um serviço alternativo para continuar funcionando (AMUNDSEN, 2023).

A adoção dessas boas práticas permite que as APIs externas sejam integradas de forma segura e eficiente, garantindo que os usuários tenham uma experiência estável e sem complicações. Plataformas que seguem essas diretrizes conseguem evitar falhas inesperadas e melhorar a confiabilidade dos serviços oferecidos (AMUNDSEN, 2023; JACOBSON et al., 2011).

9.8 Análise e priorização

A priorização de requisitos é essencial para concentrar os esforços do projeto nos itens mais críticos, utilizando critérios como risco, valor para o negócio e custo. Essa abordagem contribui para a mitigação de riscos e para a definição da ordem de análise e de implementação dos requisitos (VAZQUEZ; SIMõES, 2020). No desenvolvimento de aplicações web, a priorização é ainda mais essencial devido à natureza dinâmica e incerta desse ambiente. Mudanças rápidas nas estratégias e nas regras de negócio exigem adaptações constantes, e a gerência, muitas vezes, requer respostas imediatas, mesmo diante de demandas complexas ou atípicas. Nesse cenário, a entrega rápida de uma aplicação web tende a ser mais valorizada do que o esforço necessário para desenvolvê-la. Portanto, as equipes de engenharia web devem adotar uma abordagem ágil para responder a essas demandas com eficiência (PRESSMAN; MAXIM, 2009).

Dessa forma, a participação dos *stakeholders* é essencial nesse processo, pois são eles que definem a urgência de cada requisito. No entanto, em muitos projetos reais, a priorização é frequentemente negligenciada ou atribuída unicamente à equipe de desenvolvimento, gerando listas de tarefas desorganizadas, nas quais tudo é tratado como urgente, o que pode levar a uma distribuição incorreta de recursos (VAZQUEZ; SIMõES, 2020). Dessa forma, para evitar esse problema, métodos de priorização, como GUT, BASICO e RICE, podem ser aplicados, permitindo uma comparação objetiva dos requisitos. Esses modelos ajudam a garantir que as decisões sejam fundamentadas e alinhadas com os objetivos do projeto, promovendo uma priorização mais eficaz. A seguir, será falado a respeito de cada um desses métodos.

9.8.1 Matriz GUT

O método Gravidade, Urgência e Tendência (GUT) usa uma escala de 1 a 5 para priorizar projetos com base em três critérios: Gravidade (impacto nas atividades), Urgência (tempo restante para resolução) e Tendência (velocidade com que um problema pode piorar).Para aplicar o método, deve-se montar uma tabela, como ilustrado na Figura 9.12, na qual se atribuem notas a cada critério dos projetos. A soma dessas notas determina a prioridade, em que projetos com as maiores pontuações são priorizados (KEPNER; TREGOE, 1972).

Importância = G x U x T		
G	Gravidade	É o fator impacto financeiro ou qualquer outro dependendo dos objetivos da instituição
U	Urgência	É o fator Tempo
T	Tendência	É o fator Tendência (Padrão de desenvolvimento)

Figura 9.12: Matriz GUT

9.8.2 RICE

A matriz *Reach, Impact, Confidence e Effort* (RICE) classifica projetos com base em quatro critérios: *Reach* (alcance), *Impact* (impacto), *Confidence* (confiança) e *Effort* (esforço). Cada critério é avaliado e recebe uma nota, e a decisão sobre a escala de notas é flexível conforme a necessidade (DAVIS et al., 2001). Cada critério recebe uma nota e a importância do projeto é calculada pela seguinte fórmula, onde R é o alcance, I é o impacto, C é a confiança e E é o esforço:

$$\text{Pontuação} = \frac{R \times I \times C}{E}$$

O resultado dessa fórmula ajuda a comparar e priorizar projetos, com os que obtêm maiores pontuações sendo considerados de maior prioridade. A Figura 9.13 ilustra o funcionamento dessa matriz:

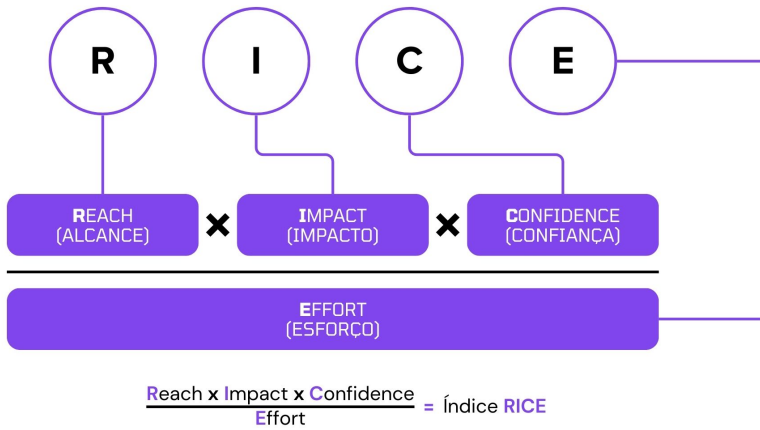


Figura 9.13: Matriz GUT

9.8.3 Matriz BASICO

A Matriz Benefícios, Abrangência, Satisfação, Investimento, Cliente e Operacionalidade (BASICO) é uma ferramenta que auxilia na tomada de decisões importantes, classificando tarefas e ações por prioridade. Utilizando uma escala de 1 a 5, onde 1 representa o pior cenário e 5 o melhor, você avalia cada tarefa com base em seis critérios: Benefícios para a organização, Abrangência da solução, Satisfação dos usuários, Investimento necessário, Cliente externo satisfeito, Impacto sobre os clientes e capacidade de Operacionalização. Após atribuir notas a cada critério, somam-se os valores e compara-se a pontuação total com a de outras tarefas para determinar a prioridade (KEPNER; TREGOE, 1972).

O resultado dessa fórmula facilita a comparação e priorização de projetos, destacando aqueles com maiores pontuações como os de maior prioridade. A Figura 9.14 ilustra o funcionamento da matriz BASICO:



Figura 9.14: Matriz BASICO

9.9 Gerenciamento de Mudanças

Uma das maiores dificuldades no processo de levantamento de requisitos é a sua constante mudança ao longo do ciclo de vida do projeto. Essa instabilidade é frequentemente relatada por analistas de requisitos, e muitas alterações poderiam ser evitadas com um trabalho mais sólido nas etapas iniciais. Em muitos casos, as mudanças decorrem de falhas na elicitação, como a postura passiva do analista ou a comunicação ineficaz com os *stakeholders*, o que resulta em uma compreensão incompleta do escopo do projeto (VAZQUEZ; SIMõES, 2020).

Sommerville destaca que, especialmente em sistemas de grande porte, os requisitos estão em constante evolução, já que os problemas que o sistema busca resolver raramente podem ser completamente definidos desde o início (SOMMERVILLE, 2020). Conforme o projeto avança, os *stakeholders* podem modificar suas necessidades ou percepções, o que exige que os requisitos acompanhem essas mudanças. Além disso, fatores externos, como novas demandas de mercado, alterações legislativas ou reestruturações organizacionais, influenciam diretamente a necessidade de adaptar os requisitos. Mesmo em metodologias ágeis, em que a flexibilidade é um princípio

central, é indispensável manter um processo estruturado para registrar, analisar e incorporar mudanças de forma controlada.

No contexto do desenvolvimento de aplicações web modernas, essa necessidade de adaptação constante também se reflete na forma como o software é entregue e atualizado. Espera-se que os usuários tenham acesso imediato às versões mais recentes do sistema, sem necessidade de instalar atualizações manualmente. Essa expectativa é viabilizada por práticas de entrega contínua, que permitem evoluir o software com baixo impacto para o usuário final, garantindo que novas funcionalidades, correções de segurança e melhorias sejam disponibilizadas com agilidade (HUMBLE; FARLEY, 2010).

Para isso, tornam-se essenciais as práticas de *Continuous Integration* e *Continuous Deployment* (CI/CD), que automatizam a integração de código, a execução de testes e a publicação de novas versões. Ferramentas como GitHub Actions, GitLab CI, Docker, Jenkins e CircleCI são amplamente utilizadas para configurar *pipelines* que reagem a eventos no repositório, como *commits* ou *pull requests*, disparando tarefas como *build*, *test* e *deploy*. Segundo Humble e Farley, a automação desses processos reduz erros manuais, acelera as entregas e aumenta a confiabilidade da liberação de software (HUMBLE; FARLEY, 2010).

Um *pipeline* eficaz de entrega contínua deve promover o mesmo artefato entre os ambientes de desenvolvimento, homologação e produção, sem recompilações, garantindo consistência e facilitando o rastreamento de falhas. Essa estratégia exige a separação entre código e configuração de ambiente, possibilitando o ajuste seguro de variáveis sensíveis, como URLs ou credenciais (HUMBLE et al., 2006).

Além disso, para mitigar riscos durante atualizações, técnicas como *rollback*, *blue-green deployments*, *canary releases* e *feature toggles* são empregadas. Essas abordagens possibilitam implantações progressivas, monitoramento em tempo real e reversão automática em caso de falhas, oferecendo maior controle sobre o comportamento do sistema em produção (HUMBLE et al., 2006).

A integração entre gerenciamento de mudanças e entrega contínua evidencia a importância de práticas que viabilizem a adaptação contínua do sistema, sem comprometer sua estabilidade. Ao automatizar processos e estruturar o controle de mudanças, as equipes de desenvolvimento ganham agilidade para responder a novas demandas e maior previsibilidade na operação. Com isso, o ciclo de vida do software torna-se mais sustentável e alinhado a contextos de entrega contínua de valor, conforme proposto pelo movimento DevOps (HUMBLE; FARLEY, 2010).

9.10 Ferramentas de gerenciamento de requisitos

Para garantir que os requisitos sejam documentados e rastreados de forma eficaz, diversas ferramentas podem ser empregadas. Nesta seção, serão apresentadas três ferramentas amplamente utilizadas para esse fim: JIRA, Helix RM e Visure, que servem para aprimorar a coleta e a gestão de informações.

9.10.1 JIRA

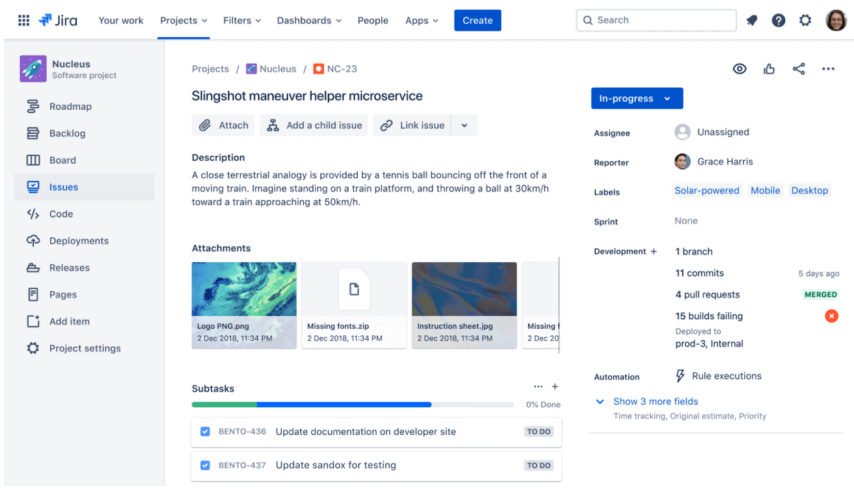


Figura 9.15: JIRA
Adaptado Parker (2024)

É uma ferramenta de gerenciamento de projetos e de rastreamento de problemas desenvolvida pela Atlassian. Muito utilizada no gerenciamento de requisitos e no desenvolvimento ágil. Também é bastante popular em equipes ágeis e na integração com outras ferramentas de desenvolvimento (PARKER, 2024).

- Prós
  - Suporte de Conformidade: Facilita o alinhamento com padrões e regulamentações, incluindo assinaturas eletrônicas e trilhas de auditoria;
  - Segurança: Oferece robustos controles de acesso e permissões para proteger dados confidenciais;
  - Boa integração com outras ferramentas;
- Contras:
  - Curva de aprendizado relativamente acentuada para novos usuários;
  - Custo elevado, especialmente para grandes equipes;

9.10.2 Helix RM

É uma ferramenta especializada para o gerenciamento de requisitos, desenvolvida pela Perforce Software, voltada à captura, organização e rastreamento ao longo do ciclo de vida do projeto. É mais adequada para organizações com projetos mais complexos (Visure Solutions, 2024).

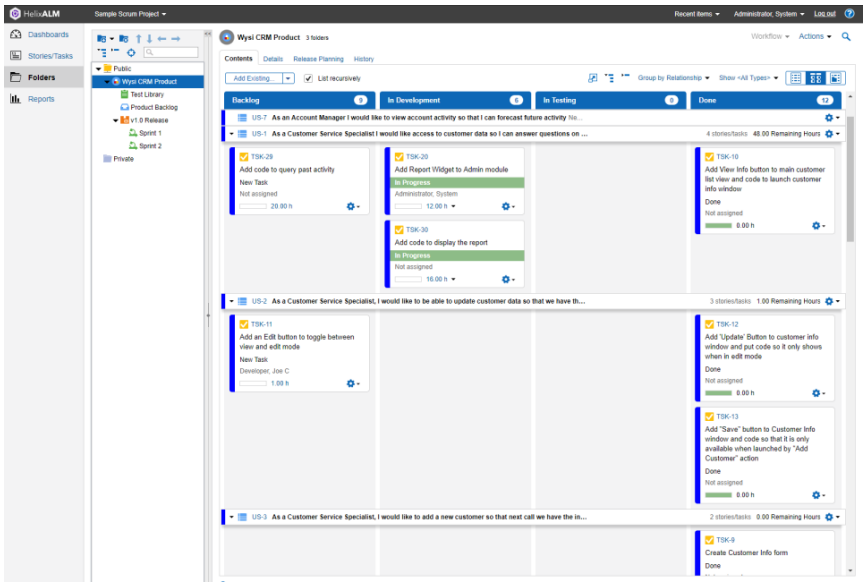


Figura 9.16: Helix RM
Adaptado Capterra (2024)

- Prós
  - Suporte robusto para conformidade;
  - Boa segurança e personalização;
  - Recursos avançados de relatórios e colaboração;
- Contras:
  - Curva de aprendizado relativamente acentuada para novos usuários;
  - Custo elevado;

9.10.3 Visure

É uma ferramenta de gerenciamento de requisitos altamente configurável e flexível, adequada para quem busca personalização em seus processos. Além de gerenciar requisitos, o Visure também oferece funcionalidades para gerenciamento de mudanças, testes, riscos, e rastreamento de problemas e defeitos (PARKER, 2024).

- Prós
  - Integra-se com softwares importantes como IBM DOORS, JIRA, MATLAB e VectorCAST;

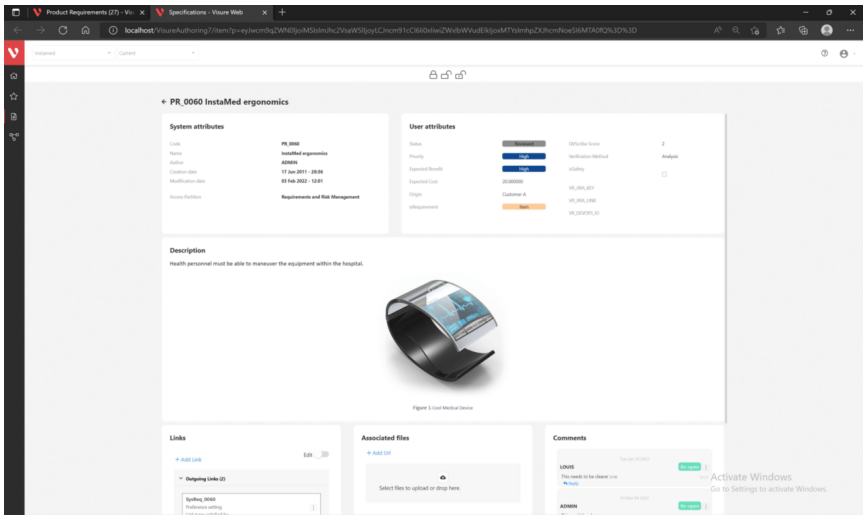


Figura 9.17: Visure
Adaptado Parker (2024)

  - Suporta várias metodologias de projeto, incluindo Agile, modelo V e cascata;
  - Oferece modelos específicos para diferentes setores;

- Contras:
  - Criação de modelos de relatórios pode ser tediosa e complicada;
  - A instalação e configuração são desafiadoras.

Capítulo 10

Web Semântica

Alanna Maria Machado Alves Paiva (UFC)

Fischer Ferreira (UNIFEI)

Este capítulo apresenta uma trajetória evolutiva da web, desde sua origem até os avanços proporcionados pela Web Semântica. Na primeira seção (Seção 10.1), aborda-se a World Wide Web (WWW), contextualizando sua criação, seu funcionamento e sua importância histórica. Em seguida, a seção Visão Geral sobre a Web Semântica apresenta os fundamentos teóricos e conceituais que orientam essa abordagem (Seção 10.2). A terceira seção, Ferramentas e Tecnologias da Web Semântica, explora os principais padrões e linguagens que possibilitam a representação, a integração e a consulta de dados de forma semântica (Seção 10.3). Por fim, a seção Web Semântica, no cenário atual, discute aplicações práticas, tendências recentes e o papel estratégico da Web Semântica em áreas como a educação e a inteligência artificial, evidenciando sua relevância crescente na sociedade contemporânea (Seção 10.4).

10.1 World Wide Web (WWW)

A World Wide Web, ou simplesmente web, é um dos principais componentes da internet, responsável por viabilizar o acesso e o compartilhamento de informações em formato de documentos multimídia, como textos, imagens e vídeos, por meio de navegadores. Criada por Tim Berners-Lee no início dos anos 1990, a web surgiu com o objetivo de permitir o compartilhamento autorizado de informações entre cientistas e instituições de pesquisa em diferentes partes do mundo (BERNERS-LEE, 1999).

A estrutura da web baseia-se no conceito de hiperlinks, que são conexões que interligam páginas e recursos distintos. Por meio desses links, os usuários podem navegar de forma não linear entre conteúdos diversos, explorando uma ampla rede de informações. Cada recurso disponível na web é identificado por um endereço único, chamado URL (Uniform Resource Locator), que possibilita sua localização e acesso específicos (BERNERS-LEE, 1992).

Berners-Lee também foi responsável pelo desenvolvimento do protocolo HTTP (Hypertext Transfer Protocol), que estabelece a comunicação entre servidores e clientes de forma leve e eficiente. Além disso, ele introduziu a linguagem HTML (Hypertext Markup Language), utilizada para estruturar os conteúdos exibidos em páginas web. Com esses componentes (URL, HTTP e HTML) formou-se a base tecnológica que tornou a web acessível e funcional em escala global.

O projeto da World Wide Web foi concebido com a visão de construir um espaço universal de informações, no qual cada item pudesse ser recuperado a partir de uma referência única. Desde então, a web transformou profundamente a forma como as pessoas acessam e compartilham informações, democratizando o conhecimento e facilitando a comunicação em escala planetária. Tornou-se uma plataforma essencial para diversas atividades cotidianas, como pesquisa, comunicação, educação, entretenimento e comércio eletrônico. Sua evolução contínua segue moldando a maneira como indivíduos, organizações e sistemas interagem na era digital (AGARWAL; SASTRY, 2022; GOYAL et al., 2023; CHAQFEH et al., 2023).

10.1.1 Hypertext Markup Language (HTML)

A web, em sua concepção original, não foi desenvolvida com foco na semântica. Contudo, com o crescimento acelerado da informação online e a necessidade de melhorar a organização, a acessibilidade e a indexação dos conteúdos, Tornou-se necessário adotar mecanismos que conferissem significado estrutural aos elementos das páginas. O HTML (Hypertext Markup Language) surgiu como uma linguagem padrão para atender a essa demanda, oferecendo uma base comum para a criação e interconexão de documentos hipertextuais. Projetado para ser simples e universal, o HTML permitiu definir a estrutura essencial das primeiras páginas da web. Por ser uma linguagem de marcação, sua principal função era definir a organização do conteúdo, utilizando elementos básicos como cabeçalhos, listas, parágrafos e divisões.

A Listagem 10.1 apresenta um exemplo de código em HTML na versão 4. A estrutura é iniciada pela diretiva `<!DOCTYPE>` (linha 1), que declara o tipo de documento e a versão da linguagem utilizada. Essa declaração passou por ajustes ao longo das versões do HTML para acompanhar as novas especificações. No corpo do código, diferentes elementos desempenham funções estruturais. No exemplo apresentado, as tags `<div>` são utilizadas para dividir a página em seções, mas não atribuem significado semântico ao conteúdo que encapsulam (BERNERS-LEE et al., 1992).

- `<head>` (linha 3): contém metadados do documento, incluindo título, codificação e outras informações não exibidas diretamente na página;
- `<title>` (linha 4): define o título que será exibido na aba do navegador;
- `<body>` (linha 6): envolve todo o conteúdo visível da página.

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
2  <html>
3  <head>
4      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"
           >
5      <title>Exemplo de Página HTML</title>
6  </head>
7  <body>
8      <div id="header">Bem-vindo à Página HTML</div>
9      <p>Esta é uma página de exemplo criada em HTML.</p>
10
11     <div id="menu">
12         <div class="post">Início</div>
13     </div>
14
15     <div id="footer">Rodapé da página</div>
16 </body>
17 </html>
```

Listagem 10.1: Exemplo de página HTML

O HTML4[1] apresenta uma estrutura simples e funcional, mas sua limitação reside na ausência de elementos com valor semântico explícito. Isso dificultava a interpretação automatizada do conteúdo, especialmente por motores de busca e tecnologias assistivas. Com o tempo, a linguagem evoluiu, incorporando elementos capazes de expressar melhor a função de cada parte da página. Essa evolução resultou no HTML5[2], versão mais recente e amplamente adotada.

A Web Semântica surgiu justamente da necessidade de tornar o conteúdo mais inteligível tanto para humanos quanto para máquinas. No HTML4, como ilustrado na Listagem 10.1, o uso predominante de elementos genéricos, como `<div>`, comprometia a compreensão do propósito do conteúdo. Embora tais elementos organizassem visualmente a página, não informavam à máquina se se encontravam em um menu, um rodapé ou um artigo, o que dificultava a extração de significado.

A introdução do HTML5 representou um avanço significativo nesse aspecto. O HTML5, embora não seja tecnicamente parte dos padrões de Web Semântica definidos pelo W3C (como RDF[3], OWL[4] e SPARQL[5]), é um facilitador da semântica no nível estrutural da web. Essa versão do HTML passou a oferecer elementos com valor semântico explícito, como `<header>`, `<footer>`, `<article>`, `<section>` e `<nav>` (PILGRIM, 2010). Com essas novas tags, o desenvolvedor consegue estruturar o conteúdo de forma mais expressiva e compreensível para diferentes agentes da web, como navegadores, mecanismos de busca e leitores de tela. A Figura 10.1 ilustra a estrutura típica de uma página em HTML5, com ênfase na distribuição semântica dos elementos (LAWSON; SHARP, 2011).

Para exemplificar a aplicação dos elementos semânticos, a Listagem 10.2 apresenta uma estrutura básica de uma página em HTML5. Nesse exemplo, observa-se a substituição de `<div>` genéricos por elementos com significado próprio, o que melhora a compreensão do conteúdo pelas máquinas.

- `<body>` (linha 8): abriga todo o conteúdo da página, agora estruturado com elementos semânticos;
- `<header>` (linha 9): representa o cabeçalho da página ou seção, podendo conter títulos, logotipos e informações introdutórias;
- `<nav>` (linha 12): identifica a área de navegação, contendo links para outras seções ou páginas;
- `<article>` (linha 15): delimita um conteúdo independente e autocontido, como uma notícia ou postagem;
- `<section>` (linha 16): agrupa conteúdo tematicamente relacionado dentro de um artigo ou página;

[1] ⟨https://www.w3.org/TR/html4/⟩
[2] ⟨https://html.spec.whatwg.org/multipage/⟩
[3] ⟨https://www.w3.org/TR/rdf11-concepts/⟩
[4] ⟨https://www.w3.org/TR/owl2-overview/⟩
[5] ⟨https://www.w3.org/TR/sparql11-query/⟩

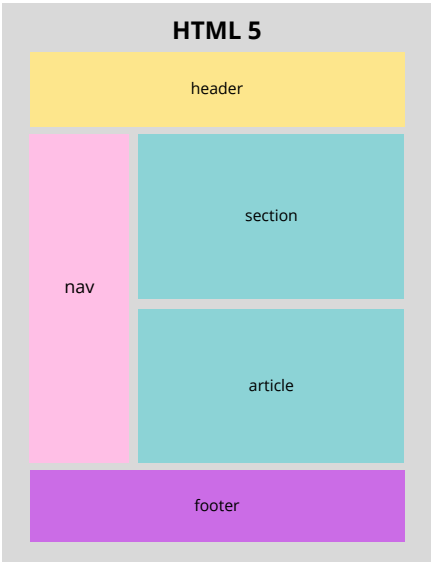


Figura 10.1: Estrutura do HTML5

- `<footer>` (linha 20): define o rodapé, geralmente contendo informações auxiliares como contato ou direitos autorais.

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-
            scale=1.0" />
        <title>Exemplo de Página HTML</title>
    </head>
    <body>
        <header class="header" id="header">
            Cabeçalho da página
        </header>
        <nav class="nav">
            Menu
        </nav>
        <article>
            <section id="inicio">
                <h2>Bem-vindo ao Meu Site!</h2>
            </section>
        </article>
        <footer>
            <p>Rodapé da página</p>
        </footer>
    </body>
</html>
```

Listagem 10.2: Exemplo de página HTML5

O exemplo evidencia como o HTML5 promove uma estruturação mais clara e informativa, favorecendo a interoperabilidade entre aplicações web. A adoção de elementos semânticos fortalece a capacidade de agentes automatizados de interpretar, classificar e reutilizar o conteúdo, além de melhorar significativamente a experiência de navegação para usuários que dependem de tecnologias assistivas. Essa transição é um dos pilares fundamentais da Web Semântica, ao permitir que o conteúdo seja não apenas exibido, mas também compreendido. Vale destacar que, embora o HTML5 tenha introduzido elementos semânticos importantes, atua principalmente na camada de estrutura da informação. A Web Semântica, conforme definida pelo W3C, também envolve a representação formal de conhecimento por meio de tecnologias como RDF, OWL e SPARQL.

10.1.2 W3C

O *World Wide Web Consortium* (W3C)[6] é um consórcio internacional responsável pelo desenvolvimento de padrões que orientam o funcionamento da web. Fundado por Tim Berners-Lee, o W3C tem como principal missão promover uma arquitetura técnica consistente, capaz de acompanhar a rápida evolução das tecnologias web e de garantir sua interoperabilidade (CONSORTIUM, 2023).

[6] ⟨https://www.w3.org/⟩

Os padrões definidos pelo W3C servem de referência para o desenvolvimento de sites, navegadores e dispositivos, assegurando que todos possam acessar os recursos da web de forma eficiente e compatível. Além disso, esses padrões oferecem uma base sólida para enfrentar desafios fundamentais, como a acessibilidade, a internacionalização, a privacidade e a segurança.

A criação do W3C também desempenhou um papel estratégico na coordenação do desenvolvimento da web. Diante do avanço acelerado das tecnologias, tornou-se necessário reunir pesquisadores, desenvolvedores e empresas em torno de um esforço comum, evitando fragmentações e promovendo a adoção de diretrizes unificadas (BERNERS-LEE et al., 1992; CONSORTIUM, 2023).

10.1.3 A Evolução da Web

A web passou por transformações significativas desde sua criação. Originalmente, a estrutura da chamada web tradicional (ou Web 1.0) foi projetada apenas para apresentar informações, sem mecanismos que permitissem sua interpretação por computadores. Nesse modelo, os conteúdos eram processados exclusivamente por seres humanos, o que limitava a automação e o uso inteligente das informações (BREITMAN, 2005).

Essa limitação impulsionou o desenvolvimento da Web Semântica, que busca atribuir significado aos dados por meio de metadados estruturados. A proposta consiste em descrever não apenas o conteúdo das páginas, mas também as relações entre os elementos, permitindo que agentes automatizados compreendam, relacionem e manipulem as informações de forma mais inteligente (SEGARAN et al., 2009).

Com essa abordagem, a web deixa de ser apenas um repositório de dados e passa a funcionar como uma rede de informações interconectadas e compreensíveis por sistemas computacionais. Isso amplia as possibilidades de automação, personalização e integração entre aplicações, tornando a web mais eficiente e acessível para diferentes públicos e dispositivos (AGARWAL; SASTRY, 2022; GOYAL et al., 2023; CHAQFEH et al., 2023).

10.2 Visão Geral sobre a Web Semântica

O conceito de Web Semântica foi introduzido em 2001 por Berners-Lee et al. (2001), o inventor da web, com o objetivo de propor uma nova forma de estruturar e interligar informações na internet. Diferentemente da web tradicional, voltada basicamente para usuários humanos, a Web Semântica, por meio de um conjunto de ferramentas, padrões e tecnologias, permite que máquinas processem, interpretem e façam inferências sobre dados estruturados publicados na web (EIS, 2017). Isso permite que as máquinas auxiliem os usuários na obtenção de respostas mais precisas e relevantes, reduzindo a necessidade de intervenção humana, como ocorre nos mecanismos de busca convencionais, como o Google.

Para introduzir o tema de forma prática, os Cenários das Figuras 10.2 e 10.3 apresentam duas situações reais de aplicação da Web Semântica no cotidiano de duas pessoas (Juliana e Ana), evidenciando seu potencial em diferentes contextos.

O Cenário da Figura 10.2 descreve o uso de um assistente virtual para realizar uma busca por restaurantes, utilizando tecnologias como RDF, SPARQL e vocabulários padronizados (por exemplo, schema.org em JSON-LD[7]). Essa tecnologia permite transformar dados desconexos em informações organizadas e úteis, respondendo de forma contextualizada e precisa à necessidade do usuário.

Cenário 1 — Assistente de voz em ambiente de saúde personalizado

Juliana está organizando um jantar com amigos e pergunta ao seu assistente virtual: *"Qual o restaurante japonês mais bem avaliado aberto agora perto de mim?"*
O assistente virtual, que utiliza tecnologias da Web Semântica, executa os seguintes passos:

1. Consulta fontes estruturadas como o Google Knowledge Graph, que utiliza dados em RDF extraídos de sites anotados com JSON-LD e schema.org.
2. Identifica restaurantes classificados como `schema:Restaurant` com `schema:cuisine` igual a "Japanese".
3. Verifica a geolocalização do usuário e filtra por restaurantes no raio de 5 km, abertos no momento.
4. Ordena os resultados com base na propriedade `schema:aggregateRating`.
5. Responde: *"O restaurante Sakura Sushi, com avaliação de 4 estrelas, está aberto agora e fica a 1,2 km de você. Deseja ligar ou fazer uma reserva?"*

Figura 10.2: Assistente de voz em ambiente de saúde personalizado

Já o Cenário da Figura 10.3 apresenta uma aplicação voltada à área da saúde. Nele, uma paciente com diabetes tipo 2 recebe recomendações personalizadas de um assistente virtual semântico. O sistema é capaz de analisar ontologias médicas, integrar dados de múltiplas fontes, sugerir ações clínicas e facilitar a comunicação entre paciente e profissionais de saúde. Essa abordagem demonstra o potencial da Web Semântica para transformar dados de saúde em decisões inteligentes e contextualizadas.

[7] ⟨https://www.w3.org/TR/json-ld11/⟩

Exemplo Atual — Web Semântica em Assistentes Virtuais

Ana, uma paciente com diabetes tipo 2, utiliza um aplicativo de assistente virtual no seu *smartphone*. Esse aplicativo, integrado com tecnologias de Web Semântica, é capaz de: O assistente virtual, que utiliza tecnologias da Web Semântica, executa os seguintes passos:

1. **Analisar ontologias médicas** (como SNOMED[a]) para interpretar corretamente os registros eletrônicos de saúde de Ana, incluindo exames laboratoriais recentes e histórico de prescrições.
2. **Integrar dados de diferentes fontes**: o assistente acessa dados de dispositivos vestíveis (como um relógio inteligente que mede glicose e frequência cardíaca), registros médicos armazenados em hospitais e dados de farmácias locais.
3. **Fazer recomendações personalizadas**: com base nas diretrizes clínicas extraídas de bases semânticas e em seu histórico, o sistema sugere que Ana faça um exame de hemoglobina glicada, pois detectou uma tendência de elevação nos níveis de glicose nas últimas semanas.
4. **Agendar automaticamente uma consulta**: o assistente usa dados vinculados (Linked Data) para encontrar endocrinologistas próximos, filtrar aqueles que aceitam o plano de saúde de Ana e verificar disponibilidade em sua agenda.
5. **Gerar relatórios compreensíveis para médicos e pacientes**: as informações são apresentadas em vocabulário padronizado, facilitando tanto a comunicação entre profissionais quanto a compreensão por parte de Ana.

[a] ⟨https://www.snomed.org/what-is-snomed-ct⟩

Figura 10.3: Web Semântica em Assistentes Virtuais

Esses exemplos evidenciam como a Web Semântica está presente em tecnologias contemporâneas, promovendo a integração de dados e a geração de informações inteligentes, contextualizadas e úteis aos usuários. Sejam em atividades cotidianas ou em setores estratégicos, como a saúde, essas tecnologias ampliam as possibilidades de personalização e automação, contribuindo para uma web mais conectada e orientada ao significado.

Os dois cenários ilustram como a Web Semântica pode transformar tarefas cotidianas ao permitir que os computadores compreendam o significado das informações e colaborem ativamente com os usuários. Para tornar esses cenários possíveis, é

necessário desenvolver uma infraestrutura capaz de organizar e inter-relacionar os dados de forma compreensível para as máquinas. Nesse contexto, surgem os modelos denominados ontologias, que organizam o conhecimento e descrevem as relações entre conceitos, funcionando como dicionários ou mapas semânticos.

Além disso, para compreender plenamente os fundamentos e a aplicação da Web Semântica, é importante considerar alguns temas centrais que compõem sua estrutura e funcionamento, tais como: Metadados, Ontologias, Agentes, *Web Services*, Linguagens e Ferramentas.

10.2.1 Metadados

Metadados são informações que descrevem outros dados. Eles têm papel central na Web Semântica, pois tornam possível organizar, localizar, descrever e gerenciar conteúdos de forma estruturada e compreensível para máquinas. Ao agregar contexto e significado aos recursos da web, os metadados permitem que sistemas automatizados interpretem o conteúdo das páginas com maior precisão, facilitando sua indexação, classificação e reutilização (ŘEZNÍK et al., 2022).

No contexto da web, o desenvolvimento e uso de metadados envolvem três aspectos principais. O primeiro é a *descrição dos recursos*, ou seja, a definição clara das informações a serem tratadas como metadados. O segundo refere-se à *produção desses metadados*, que, devido ao grande volume de dados disponíveis online, tende a ser realizada por meio de processos automatizados. A criação manual, embora mais precisa em alguns contextos, é inviável em larga escala. Por fim, há o *uso dos metadados*, que envolve sua leitura e processamento por sistemas computacionais, garantindo que o significado semântico original seja preservado durante a manipulação dos dados (DIAS; SANTOS, 2003).

Além desses aspectos, outras questões devem ser consideradas. A diversidade de padrões existentes permite que o mesmo recurso seja descrito por diferentes conjuntos de qualificadores. Por isso, é comum a adoção de padrões específicos para distintos tipos de recursos, como textos acadêmicos, imagens e produtos comerciais. Outro ponto relevante é a *internacionalização dos metadados*, uma vez que muitos qualificadores foram originalmente definidos em inglês, o que exige adaptações para uso em contextos multilíngues. A geração dos metadados pode ocorrer no momento em que o recurso é criado ou posteriormente, sendo necessária sua atualização sempre que o recurso sofre modificações. Em certos casos, como em resenhas ou comentários de usuários, os metadados podem ser gerados de forma independente. Vale ressaltar, ainda, que metadados são, por definição, dados e, por isso, enfrentam os mesmos desafios relacionados ao armazenamento, ao versionamento, ao acesso e à interpretação (IANNELLA; WAUGH, 1997).

A ideia de criar um padrão universal de metadados que abranja todas as áreas do conhecimento é frequentemente discutida, mas ainda está longe de ser viável. Isso se deve à necessidade de descrever domínios muito diversos, cada um com sua própria terminologia, estrutura e requisitos específicos. Uma padronização única

implicaria perda de expressividade e exigiria um alto grau de especialização para garantir uma descrição eficaz dos recursos (DIAS; SANTOS, 2003).

Diante dessa diversidade, surgiram padrões de metadados com finalidades distintas. Um dos mais amplamente utilizados na web é o *Dublin Core*[8], desenvolvido com apoio do W3C. Projetado para ser simples e aplicável a diferentes tipos de documentos eletrônicos, o *Dublin Core* permite a criação e o compartilhamento de metadados de forma padronizada, sem exigir conhecimentos técnicos avançados. Ele é composto por 15 elementos principais, como título, autor, data, formato e descrição, e é amplamente adotado em bibliotecas digitais, repositórios institucionais e catálogos acadêmicos (DZIEKANIAK; KIRINUS, 2004).

Cada elemento do *Dublin Core* possui três propriedades fundamentais: o nome (que o identifica), o atributo (forma como é representado nos metadados) e a definição (descrição do conceito). Por exemplo, em uma fotografia digital, os metadados podem incluir informações como a data e a hora da captura, o modelo do dispositivo, a localização geográfica e as configurações da câmera. Esses dados tornam o recurso mais facilmente pesquisável, classificável e compreensível tanto para usuários quanto para sistemas automatizados.

A Tabela 10.1 apresenta os 15 elementos do padrão *Dublin Core*, com seus respectivos atributos e descrições resumidas. Esses elementos podem ser utilizados de forma flexível para descrever diversos tipos de recursos digitais (Dublin Core Metadata Initiative (DCMI), 2020).

Embora o *Dublin Core* continue sendo amplamente utilizado em ambientes acadêmicos e documentais, os metadados na web moderna têm sido frequentemente implementados com o uso de *JSON-LD* (JavaScript Object Notation for Linked Data) (Schema.org, 2024), em conjunto com vocabulários padronizados como o *Schema.org*. Essa abordagem tornou-se o método preferencial de marcação semântica em páginas web, por ser compatível com mecanismos de busca, interoperável com diversos sistemas e sem interferir na apresentação visual do conteúdo. O *Schema.org*, mantido por organizações como Google, Microsoft e W3C, oferece uma ampla gama de tipos e propriedades para descrever recursos como artigos, pessoas, eventos, produtos e instituições.

A Listagem 10.4 apresenta um exemplo de metadados estruturados em *JSON-LD* para descrever um artigo acadêmico.

[8] ⟨https://www.dublincore.org/specifications/dublin-core/dces/⟩

Tabela 10.1: Elementos do padrão Dublin Core

Elemento	**Atributo**	**Descrição**
Title	title	Título do recurso.
Creator	creator	Pessoa ou entidade responsável pela criação do conteúdo.
Subject	subject	Tema ou assunto tratado no recurso.
Description	description	Resumo ou explicação do conteúdo.
Publisher	publisher	Entidade responsável pela disponibilização do recurso.
Contributor	contributor	Colaboradores que participaram da criação do recurso.
Date	date	Data associada ao recurso (criação, publicação etc.).
Type	type	Categoria ou tipo do recurso (ex: imagem, texto, vídeo).
Format	format	Formato físico ou digital (ex: PDF, JPEG, HTML).
Identifier	identifier	Identificador exclusivo do recurso, como uma URL ou DOI.
Source	source	Recurso original do qual este deriva.
Language	language	Idioma do conteúdo (ex: pt-BR, en-US).
Relation	relation	Relacionamentos com outros recursos.
Coverage	coverage	Escopo geográfico ou temporal do conteúdo.
Rights	rights	Informações sobre direitos autorais e uso do recurso.

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "ScholarlyArticle",
  "headline": "A Evolução da Web Semântica",
  "author": {
    "@type": "Person",
    "name": "Alanna Maria",
    "affiliation": {
      "@type": "Organization",
      "name": "Universidade Federal do Ceará"
    }
  },
  "publisher": {
    "@type": "Organization",
    "name": "Universidade Federal do Ceará"
  },
  "datePublished": "2025-07-31",
  "inLanguage": "pt-BR",
  "description": "Artigo introdutório sobre o uso de metadados e o
      padrão Dublin Core na Web Semântica.",
  "keywords": ["Web Semântica", "Metadados", "Dublin Core", "Schema.
      org"],
  "url": "https://www.famart.edu.br/artigos/web-semantica"
}
</script>
```

Listagem 10.3: Exemplo de metadados no formato JSON-LD

A Tabela 10.2 resume as principais diferenças entre o padrão *Dublin Core* e o vocabulário *Schema.org*, evidenciando como cada um atende a contextos específicos. Enquanto o *Dublin Core* se mantém como uma opção simples e eficaz em ambientes institucionais e acadêmicos, o *Schema.org*, especialmente em conjunto com *JSON-LD*, consolidou-se como padrão para aplicações web voltadas à visibilidade e interoperabilidade com sistemas de busca e agentes automatizados.

Em síntese, os metadados desempenham um papel estratégico na estruturação e na evolução da Web Semântica (ŘEZNÍK et al., 2022). Ao fornecer uma camada adicional de significado aos conteúdos digitais, eles promovem a interoperabilidade entre sistemas, aprimoram a busca e a recuperação da informação e tornam possível o desenvolvimento de aplicações mais inteligentes e sensíveis ao contexto. A escolha do padrão de metadados mais adequado depende do tipo de recurso, do público-alvo e dos objetivos da aplicação, sendo o *Dublin Core* e o *Schema.org* entre as soluções mais consolidadas e complementares nesse cenário.

10.2.2 Ontologias

As ontologias desempenham um papel central na Web Semântica, ao fornecer uma estrutura formal para a organização do conhecimento e promover a interoperabilidade entre diferentes sistemas. As ontologias têm desempenhado um papel

Tabela 10.2: Comparativo entre *Dublin Core* e *Schema.org*

Critério	**Dublin Core**	**Schema.org (com JSON-LD)**
Origem	Desenvolvido por DCMI (com apoio do W3C)	Criado por Google, Microsoft, Yahoo e Yandex; mantido com apoio do W3C
Objetivo	Descrição genérica de recursos digitais	Marcação semântica para interpretação automática por motores de busca e sistemas web
Formato típico	Metatags em HTML (`<meta name="dc.title"...>`)	JSON-LD (bloco `<script type="application/ld+json`≫)
Número de elementos	15 elementos principais (ex: title, creator, date)	Centenas de tipos e propriedades específicas (ex: Person, Article, Event)
Complexidade	Simples, de uso genérico	Mais detalhado e contextualizado, requer vocabulário mais amplo
Facilidade de uso	Fácil, especialmente para iniciantes	Requer familiaridade com JSON-LD e vocabulários semânticos
Compatibilidade com motores de busca	Limitada	Altamente compatível e recomendada por Google e Bing
Principais usos	Repositórios digitais, bibliotecas, catálogos acadêmicos	SEO, sites comerciais, artigos, produtos, eventos, currículos, instituições
Exemplo de aplicação	DSpace, bibliotecas, arquivos públicos	Google Scholar, sites institucionais, comércio eletrônico

cada vez mais relevante na representação formal do conhecimento, promovendo a interoperabilidade entre sistemas e servindo de base para o desenvolvimento de aplicações inteligentes (BIBI et al., 2023; STĂNESCU; OPREA, 2025). Segundo Gruber (1993), uma ontologia é definida como "uma especificação explícita de uma conceitualização compartilhada". Em termos práticos, isso significa oferecer um vocabulário comum para representar informações de forma estruturada e compreensível por máquinas, permitindo que diferentes agentes, sejam humanos ou computacionais, interpretem conceitos da mesma maneira (FENSEL et al., 2020).

Para ilustrar essa ideia, a Figura 10.4 apresenta uma representação gráfica de uma ontologia voltada para humanos. Nela, os conceitos de "Pessoa" e "Música" são conectados por atributos e relacionamentos. Por exemplo, uma instância de "Pessoa" pode possuir atributos como "Primeiro Nome", "Sobrenome" e "Email", e relacionar-se a "Música", representando o gênero musical de sua preferência. Essa estrutura hierárquica e lógica permite uma organização clara das informações, sem ambiguidades.

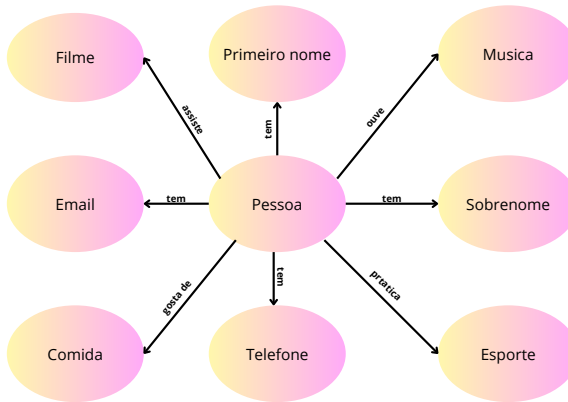


Figura 10.4: Representação gráfica de uma ontologia voltada para humanos

Em uma ontologia, diversos componentes trabalham em conjunto para estruturar o conhecimento:

Classes: categorias ou grupos de elementos com características comuns, como a classe "Cidade", que engloba todas as cidades com propriedades típicas.

Instâncias: exemplos concretos dessas classes. "Sobral" e "Fortaleza", por exemplo, são instâncias da classe "Cidade".

Atributos: características de uma classe ou de uma instância, como "nome", "população" e "área".

Propriedades: regras que definem o comportamento dos atributos, como garantir que "população" seja um número inteiro positivo.

Relacionamentos: conexões entre conceitos, como no exemplo entre "Cidade" e "Estado".

Axiomas: restrições lógicas que garantem a consistência da estrutura, como a exigência de que cada cidade pertença a exatamente um estado.

Além de organizarem o conhecimento de forma clara, as ontologias têm grande importância em contextos computacionais. Elas são empregadas na modelagem de sistemas complexos, representando entidades e suas relações de modo que possam ser processadas automaticamente. Um exemplo disso é o uso de triplas RDF (sujeito, predicado, objeto), que facilitam o intercâmbio semântico entre agentes. Tecnologias como OWL serão discutidas na Seção 10.3.5 e expandem essa lógica ao permitir a aplicação de regras e inferências mais sofisticadas (GUARINO et al., 2009).

A construção de uma ontologia requer um vocabulário bem definido, estruturado logicamente para representar uma realidade específica de forma clara e compartilhável (DIAS; SANTOS, 2003). Para isso, duas propriedades fundamentais são destacadas:

- **Compartilhamento**: garante que diferentes aplicações e sistemas utilizem o mesmo vocabulário semântico, promovendo consistência na comunicação.
- **Filtragem**: permite selecionar apenas as informações relevantes para um determinado contexto, abstraindo os detalhes não essenciais.

Essas características tornam as ontologias poderosas aliadas na busca semântica. Diferentemente dos mecanismos tradicionais, como o Google, que operam com base em palavras-chave, os sistemas baseados em ontologias interpretam o significado dos termos e suas relações. Por exemplo, ao pesquisar "curso de programação avançado", um buscador semântico entende que "programação" pode envolver linguagens como Python, Java ou JavaScript, e que "avançado" indica um nível de dificuldade. Essa compreensão é possível porque o sistema consulta uma ontologia que estrutura esses conceitos e suas inter-relações (TAYE, 2010).

Classe	Subclasse	Relação	Instância
Curso de Programação	Curso de Programação Avançado	é do tipo de linguagem	Python, JavaScript
Curso de Programação	Curso de Programação Avançado	tem nível de dificuldade	Avançado
Curso de Programação Avançado	-	é oferecido por	Universidade Federal do Ceará
Curso de Programação Avançado	-	tem pré-requisito	Conhecimento em algoritmo

Figura 10.5: Tabela de ontologia para o termo "Curso de Programação Avançado"

A ontologia apresentada na Figura 10.5 mostra como os conceitos de curso, linguagem, nível e modalidade estão interligados e organizados hierarquicamente, com subclasses e restrições associadas. Isso permite que os mecanismos semânticos

filtrem cursos com base em critérios específicos e recomendem os mais adequados ao perfil do usuário.

Para exemplificar a aplicação prática de uma ontologia na descrição de um curso, é possível utilizar a ontologia *Schema.org*, amplamente adotada por mecanismos de busca e plataformas de conteúdo. A Listagem 10.4 apresenta uma estrutura em *JSON-LD* sobre "Curso de Programação Avançado", incorporando os elementos da Figura 10.5. Nesse modelo, a classe principal "Course" define o curso, enquanto suas características são representadas por propriedades como "educationalLevel" (nível de dificuldade), "coursePrerequisites" (pré-requisitos) e "teaches" (linguagens ensinadas). A instituição ofertante é especificada como uma "EducationalOrganization", estabelecendo uma relação semântica com o provedor da formação.

Essa forma de representação permite que agentes computacionais compreendam com maior precisão o conteúdo, relacionem-se às intenções dos usuários e ofereçam resultados mais relevantes em buscas e em sistemas de recomendação. O uso de ontologias, como a do *Schema.org*, torna o conteúdo educacional mais interoperável, estruturado e acessível em diferentes plataformas na web.

```
{
  "@context": "https://schema.org",
  "@type": "Course",
  "name": "Curso de Programação Avançado",
  "description": "Curso voltado ao aprofundamento em linguagens de
      programação como Python e JavaScript, com foco em algoritmos e
      estruturas avançadas.",
  "provider": {
    "@type": "EducationalOrganization",
    "name": "Universidade Federal do Ceará",
    "url": "https://www.ufc.br"
  },
  "educationalCredentialAwarded": "Certificado de Conclusão",
  "coursePrerequisites": "Conhecimento em algoritmo",
  "hasCourseInstance": {
    "@type": "CourseInstance",
    "courseMode": "online",
    "instructor": {
      "@type": "Person",
      "name": "Prof. João Pedro"
    },
    "startDate": "2025-08-01",
    "endDate": "2025-12-15"
  },
  "audience": {
    "@type": "EducationalAudience",
    "educationalLevel": "Avançado"
  },
  "teaches": ["Python", "JavaScript"]
}
```

Listagem 10.4: Exemplo em JSON-LD para o Curso de Programação Avançado

A eficácia da Web Semântica será alcançada à medida que forem desenvolvidos programas capazes de realizar buscas precisas em diversas fontes disponíveis na web, correlacionando conteúdos com estruturas de conhecimento e regras de inferência. Com o aumento da disponibilidade de conteúdo estruturado, os agentes computacionais se tornam mais eficientes, capazes de compreender as intenções dos usuários e oferecer resultados mais alinhados às suas necessidades (DIAS; SANTOS, 2003; HITZLER, 2021; HENDLER, 2001). Assim, diferentes propostas de classificação de ontologias na literatura reforçam o objetivo comum de criar vocabulários compartilhados, que viabilizem a troca de informações de forma clara e sem ambiguidade.

10.2.3 Agentes Inteligentes

Para que a comunicação entre a web e o usuário ocorra de forma eficiente, é necessário um modelo que permita a troca de informações sobre uma tarefa específica. Nesse contexto, surgem os agentes inteligentes (ou agentes de software), que são programas responsáveis por coletar, processar e compartilhar informações para auxiliar os usuários. Em vez de executar todas as ações diretamente, esses agentes utilizam técnicas de inteligência artificial para apoiar o usuário em suas atividades, oferecendo sugestões, facilitando a tomada de decisões e promovendo uma interação mais eficiente com a web (SOUZA; ALVARENGA, 2004).

Hendler (1999), em seu artigo, apresenta uma analogia com agentes de viagem para ilustrar o funcionamento dos agentes inteligentes da web. Assim como um agente de viagens não realiza toda a tarefa por conta própria, mas auxilia na busca por opções, os agentes inteligentes interpretam os pedidos do usuário (como destino, datas e preferências), realizam buscas semânticas utilizando ontologias como FOAF (*Friend of a Friend*)[9] e *Schema.org*, consultam bases como *DBpedia* e *Wikidata*, e organizam os resultados de forma estruturada. A partir disso, sugerem rotas, meios de transporte, hospedagens e alternativas compatíveis com o perfil e orçamento do usuário.

Esses agentes empregam metadados para descrever e organizar as informações coletadas de diferentes fontes. Por exemplo, ao consultar uma base de dados de companhias aéreas, o agente acessa dados como preço, data, horário e política de bagagem, utilizando esses metadados para oferecer recomendações personalizadas. Além disso, agentes semânticos operam com base em regras e raciocínios complexos, utilizando ontologias e lógicas formais para interpretar o contexto e tomar decisões fundamentadas. Eles não apenas armazenam dados, mas também compreendem as relações entre si, possibilitando inferências que tornam as respostas mais relevantes e ajustadas às necessidades do usuário. Com isso, os agentes conseguem realizar associações sofisticadas, como:

- Identificar que um hotel está localizado próximo ao local de realização de um evento.

[9] ⟨http://xmlns.com/foaf/spec/⟩

- Reconhecer que as atividades culturais específicas estão alinhadas ao perfil do viajante.
- Entender que uma viagem em julho à Coreia exige atenção aos fatores climáticos e aos eventos sazonais.

A Figura 10.6 ilustra um exemplo de como essas conexões podem ser visualizadas em um grafo semântico:

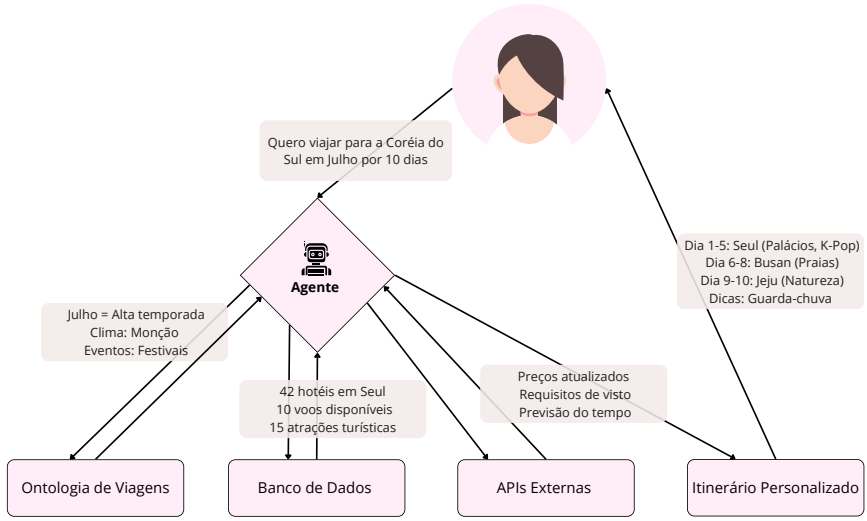


Figura 10.6: Representação de agentes inteligentes com base em grafos de conhecimento.

Essa abordagem permite que agentes inteligentes não apenas recuperem dados relevantes, mas também organizem e apresentem essas informações de forma contextualizada. Como resultado, as recomendações são mais precisas e personalizadas. A web, nesse contexto, deixa de ser apenas um repositório passivo de páginas para se tornar um ambiente dinâmico e proativo, no qual sistemas computacionais compreendem as intenções dos usuários e agem de forma colaborativa.

Em síntese, os agentes inteligentes têm como função transformar a navegação em uma experiência mais intuitiva, servindo como consultores digitais. Eles utilizam inferência semântica, dados interconectados e regras lógicas para sugerir alternativas que atendam às necessidades específicas de cada usuário, tornando a tomada de decisão mais informada e eficaz (HENDLER, 1999; SOUZA; ALVARENGA, 2004).

10.2.4 Web Services

Web services são componentes essenciais para a comunicação entre sistemas na web, possibilitando a automação de processos e a integração entre plataformas heterogêneas. Em termos gerais, são aplicações acessíveis pela internet que permitem a troca de dados entre softwares, usando protocolos e formatos padronizados como HTTP, XML, JSON, SOAP (Simple Object Access Protocol)[10] e REST (Representational State Transfer).

Na Web tradicional, os serviços funcionam com base em sintaxe, ou seja, exigem que os consumidores conheçam previamente as estruturas esperadas (parâmetros, formatos de retorno etc.). Já na Web Semântica, essa interação evolui para um nível mais sofisticado, pois os serviços podem ser descritos por meio de anotações semânticas que atribuem significado aos dados. Isso permite que agentes inteligentes descubram, interpretem, comparem e utilizem serviços de forma autônoma, sem a necessidade de codificação específica para cada caso (HENDLER, 2001).

A Figura 10.7 ilustra esse cenário, em que o agente atua como uma camada de inteligência entre o cliente e os diferentes serviços disponíveis. Ele interpreta uma solicitação complexa do usuário, consulta diversas fontes, converte formatos, agrega informações brutas e retorna uma resposta estruturada e personalizada. Essa funcionalidade só é possível devido ao uso de ontologias, metadados semânticos e raciocínio automatizado.

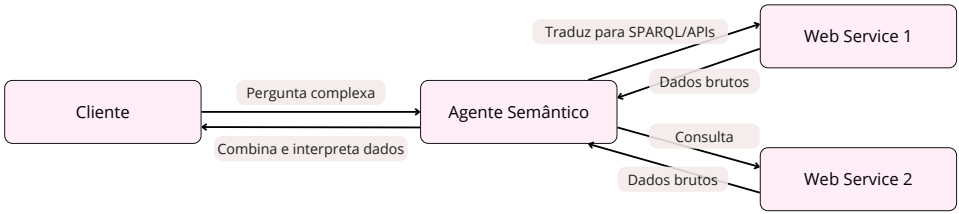


Figura 10.7: Agente como "Camada Inteligente" sobre web services

Para representar semanticamente os serviços, utilizam-se padrões como OWL-S (Web Ontology Language for Services)[11], que fornece uma ontologia em OWL para descrever serviços web em três aspectos: perfil (o que o serviço faz), modelo de processo (como ele funciona) e ligação (como acessá-lo). Outra abordagem comum é o uso de SAWSDL (Semantic Annotations for WSDL and XML Schema)[12], que permite associar conceitos ontológicos diretamente a descrições de WSDL existentes. Há ainda iniciativas, como o WSMO (Web Service Modeling Ontology), voltadas à modelagem semântica, à mediação e à execução de serviços.

Essas tecnologias tornam possível a criação de sistemas de descoberta automática de serviços baseados em significado. Por exemplo, um agente pode

[10] ⟨https://www.w3.org/TR/soap/⟩
[11] ⟨https://www.w3.org/submissions/OWL-S/⟩
[12] ⟨https://www.w3.org/TR/sawsdl/⟩

procurar serviços que ofereçam “hospedagem próxima a pontos turísticos” mesmo que o termo usado pelo fornecedor seja “alojamento perto de atrações locais”. A ontologia e os metadados semânticos resolvem essa diferença terminológica.

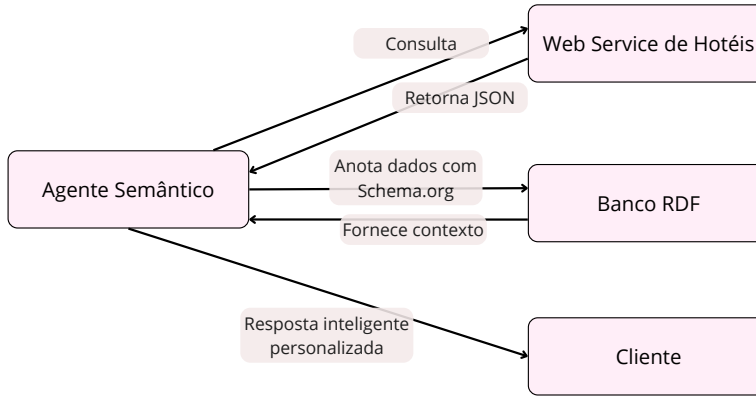


Figura 10.8: Web Service como fonte de dados para um agente

A Figura 10.8 mostra um exemplo de aplicação prática: um agente semântico consulta um Web Service de hotéis que retorna dados em JSON. O agente anota semanticamente essas informações com vocabulários como o Schema.org, armazena-as em um banco de dados RDF e combina-as com outras fontes para responder ao cliente de forma personalizada. Essa capacidade é essencial em contextos como o turismo, a saúde, a logística e o comércio eletrônico, em que múltiplas fontes e critérios precisam ser integrados dinamicamente.

Para tornar essas informações compreensíveis por máquinas, os serviços podem ser enriquecidos com metadados semânticos por meio de anotações como o vocabulário do Schema.org. Por exemplo, a Listagem 10.5 apresenta uma possível resposta anotada em JSON-LD por um Web Service de hotéis.

```
{
  "@context": "https://schema.org",
  "@type": "Hotel",
  "name": "Hotel Seul Garden",
  "address": {
    "@type": "PostalAddress",
    "addressLocality": "Seul",
    "addressCountry": "KR"
  },
  "starRating": {
    "@type": "Rating",
    "ratingValue": "4"
  },
  "amenityFeature": [
    {
      "@type": "LocationFeatureSpecification",
      "name": "Wi-Fi",
      "value": true
    },
    {
      "@type": "LocationFeatureSpecification",
      "name": "Café da manhã incluso",
      "value": true
    }
  ]
}
```

Listagem 10.5: Exemplo em JSON-LD de um Web Service no domínio de hotéis

Com essa anotação, um agente semântico pode identificar automaticamente que se trata de um hotel com 4 estrelas em Seul, que oferece Wi-Fi e café da manhã, mesmo que essas informações estejam expressas de formas diferentes na interface do serviço. Isso permite que o agente combine esses dados com as preferências do usuário e com outras fontes, oferecendo respostas mais relevantes e personalizadas.

Além disso, o uso de ontologias nas descrições dos serviços viabiliza o *matchmaking* semântico, ou seja, a correspondência inteligente entre as necessidades do usuário e as funcionalidades disponíveis. A Figura 10.9 exemplifica esse fluxo, destacando os papéis dos solicitantes, dos provedores e dos mecanismos de correspondência baseados em ontologias. Nessa arquitetura, tanto o consumidor quanto o fornecedor do serviço descrevem seus interesses ou capacidades por meio de anotações compatíveis com uma ontologia comum. O mecanismo de correspondência interpreta semanticamente essas descrições e identifica serviços relevantes mesmo com vocabulários distintos.

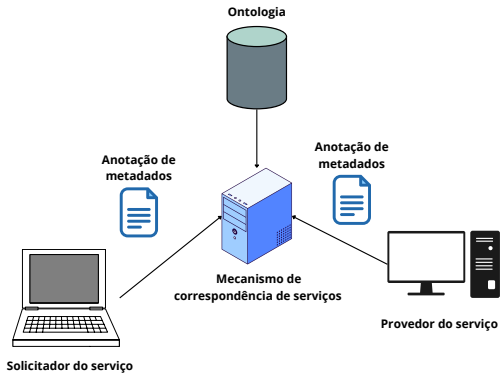


Figura 10.9: Exemplo de uso de ontologia na oferta de serviços na Web

Portanto, os web services semânticos representam um avanço significativo em relação aos modelos tradicionais. Eles permitem a descoberta, a composição e a invocação automática de serviços com base em significados compartilhados. Quando combinados com agentes inteligentes, possibilitam a criação de sistemas flexíveis, autônomos e interoperáveis, capazes de transformar a web em uma verdadeira plataforma de cooperação entre máquinas (HENDLER, 2001).

10.2.5 Linked Data

Linked Data, ou Dados Ligados, é uma abordagem que utiliza a web para estabelecer conexões estruturadas entre diferentes conjuntos de dados. Essas informações podem vir de diversas fontes, como bancos de dados de organizações situadas em locais distintos ou até mesmo de sistemas internos de uma mesma empresa, que antes não se comunicavam facilmente. A vasta quantidade de informações não estruturadas disponíveis na Internet impulsionou o desenvolvimento e a adoção de técnicas da Web Semântica, com o objetivo de tornar os dados mais interconectados, compreensíveis e úteis ao domínio público (AGUILAR et al., 2023).

De forma técnica, Linked Data refere-se a dados publicados na web de modo que possam ser processados automaticamente por máquinas. Esses dados possuem significados bem definidos, estão interligados a outros conjuntos de informações externas e podem ser acessados e integrados a novos conjuntos de dados, criando uma grande rede de conhecimento interconectado (BIZER et al., 2023).

Enquanto as unidades primárias da web de hipertexto são documentos HTML conectados por hiperlinks não tipados, o Linked Data usa documentos que contêm dados no formato RDF. No entanto, em vez de apenas conectar esses documentos, o Linked Data utiliza o RDF para criar afirmações tipadas que conectam entidades do mundo real. O resultado, frequentemente chamado de “Web de Dados”, pode

ser melhor definido como uma rede de coisas do mundo descritas por dados na web (BIZER et al., 2023).

Ele se baseia no uso de URIs, necessárias para identificar e vincular exclusivamente recursos na web. Um URI é, basicamente, uma *string* que identifica um recurso na web. Todos os recursos de informação, como páginas da WWW e documentos, possuem um URI, comumente conhecido como URL. Isso facilita a navegação e a descoberta de informações interligadas em diferentes fontes, promovendo a criação de uma rede de dados semânticos.

Conforme discutido no artigo *Linked Data - The Story So Far*, de Bizer et al. (2008), essa abordagem é fundamental para a construção de uma Web Semântica, na qual os dados podem ser facilmente acessados e integrados por meio de diferentes plataformas e contextos. Um dos conjuntos de dados mais conhecidos dessa abordagem é o *DBpedia*, um projeto que extrai dados estruturados da Wikipédia e os disponibiliza no formato de Linked Data. Isso permite que diferentes aplicações consultem e relacionem informações automaticamente. Por exemplo, ao consultar a DBpedia sobre um filme, é possível obter automaticamente dados sobre o elenco, o diretor e o ano de lançamento, sem depender de um sistema fechado.

Com o objetivo de estruturar essa publicação de dados de forma eficiente, Berners-Lee et al. (2006) estabeleceu um conjunto de princípios fundamentais para a disseminação de Linked Data. Esses princípios garantem que todos os dados publicados façam parte de um espaço de dados global único (HEATH, 2011):

- Utilizar URIs para identificar recursos: cada entidade do mundo real (como pessoas, lugares ou conceitos) deve possuir um identificador único na web.
- Utilizar URIs acessíveis via HTTP: essas URIs devem ser recuperáveis por meio do protocolo HTTP.
- Fornecer dados úteis ao acessar uma URI: o servidor deve retornar informações estruturadas em formatos padronizados, como RDF ou JSON-LD.
- Incluir links para outros dados: os dados disponibilizados devem conter referências a outros recursos, permitindo que humanos e agentes computacionais descubram novas informações automaticamente.

Essas diretrizes, conhecidas como os princípios do Linked Data, servem como um guia básico para a publicação e conexão de dados na web. Ao respeitar a arquitetura e os padrões da web, esses princípios possibilitam a criação de um ecossistema de dados interligados, promovendo maior acessibilidade e reutilização das informações.

A Figura 10.10 apresenta, como exemplo, a URI de identificador de recurso de um livro intitulado *Entendendo a Web*. A URI que identifica esse livro representa o recurso de forma única na web. Quando acessada por um navegador habilitado para RDF, essa URI redireciona para o documento RDF que contém dados estruturados

sobre o livro, incluindo propriedades como autor, data de publicação e relacionamentos com outras obras. Por outro lado, ao ser acessada por um navegador da web HTML, a URI redireciona para um documento HTML, como uma página da web sobre o livro no site da editora.

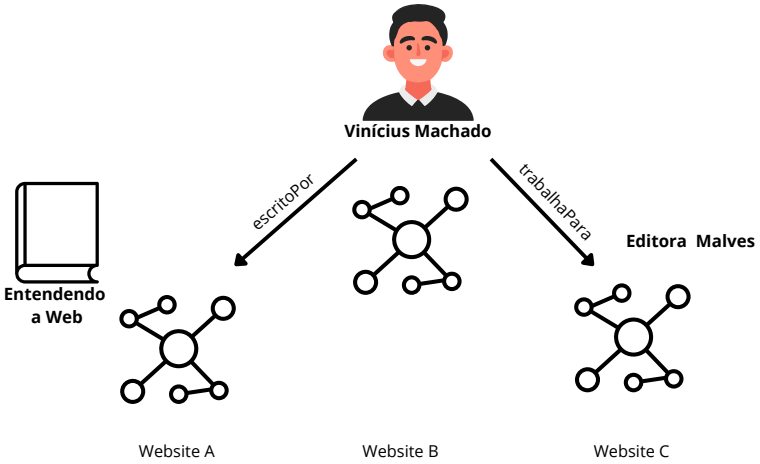


Figura 10.10: Funcionamento do Linked Data

O Linked Data destaca-se como um dos fundamentos principais para a organização e a recuperação estruturadas de informações na Web Semântica. Sua aplicação permite a interligação de fontes diversas, resultando em consultas mais precisas e contextualizadas. Diferentemente de bancos de dados isolados, essa abordagem facilita a descoberta de novos relacionamentos entre os dados, impulsionando a criação de serviços mais inteligentes e eficientes (HEATH, 2011). Dessa forma, o Linked Data não apenas melhora a acessibilidade da informação, mas também contribui para um ambiente digital mais integrado e dinâmico.

10.3 Ferramentas e Tecnologias da Web Semântica

A Web Semântica é fundamentada em uma série de tecnologias e padrões que possibilitam a descrição, a interconexão e o raciocínio sobre dados na web. Essas tecnologias tornam possível transformar dados brutos ou estruturados em conhecimento compreensível e processável por máquinas, promovendo maior interoperabilidade entre sistemas e agentes inteligentes.

Essas ferramentas podem ser organizadas em categorias complementares, conforme descrito a seguir:

1. **Modelos de dados**
   Esses modelos são responsáveis por estruturar as informações semânticas.

- **RDF (Resource Description Framework)**: modelo padrão de representação de dados, baseado em triplas (sujeito, predicado, objeto).
- **JSON-LD (JSON for Linking Data)**: formato baseado em JSON que incorpora contexto semântico por meio de vocabulários, como o da Schema.org.
- **XML**: amplamente utilizado para a serialização de dados, sendo também a base de formatos como RDF/XML.

2. **Linguagens de ontologia**
   Essas linguagens permitem descrever classes, propriedades e relacionamentos entre conceitos.

   - **RDFS (RDF Schema)**: fornece um vocabulário básico para a definição de hierarquias de classes e propriedades.
   - **OWL (Web Ontology Language)**: linguagem mais expressiva que permite descrever restrições, cardinalidades e equivalências entre conceitos.
   - **SKOS (Simple Knowledge Organization System)**: ideal para representar tesauros, taxonomias e outros sistemas de organização do conhecimento.

3. **Validação de dados**

   - **SHACL (Shapes Constraint Language)**: linguagem que define restrições e valida a conformidade dos dados RDF com uma estrutura esperada.

4. **Linguagens de consulta**

   - **SPARQL (SPARQL Protocol and RDF Query Language)**: linguagem de consulta equivalente ao SQL no contexto da Web Semântica, utilizada para recuperar e manipular dados em formato RDF.

Essas tecnologias constituem a base técnica da Web Semântica, permitindo que os sistemas compreendam o significado dos dados e realizem inferências, descobertas e integrações automatizadas.

10.3.1 Resource Description Framework (RDF)

O Resource Description Framework (RDF) é um modelo criado pelo W3C para organizar e descrever informações na web de forma compreensível tanto para pessoas quanto para máquinas (CYGANIAK et al., 2014). Ele constitui uma parte essencial da Web Semântica ao estruturar e atribuir significado aos

Tabela 10.3: Componentes de uma Tríplice RDF

Triplas	**Definição**	**Exemplo**
Sujeito	“Pessoa”, ou qualquer objeto, ideia, ou seja, qualquer coisa que possa ser reconhecida, nomeada ou encontrada.	“Pessoa”
Predicado	O predicado descreve a relação entre o sujeito e o objeto. Ele especifica como o sujeito está relacionado ao objeto. Os predicados são representados por URIs que apontam para termos definidos em vocabulários ou ontologias RDF.	”gosta de”
Objeto	O objeto é o valor que está sendo relacionado ao sujeito, podendo ser um valor direto, como um número, uma palavra ou uma referência para outro recurso.	”Comida”

dados, facilitando o processamento e a inter-relação das informações por sistemas computacionais (NECULA, 2020). A principal ideia do RDF é representar os dados como triplas, em vez de utilizar estruturas tradicionais, como as tabelas de bancos de dados relacionais. Nesse modelo, as informações são conectadas entre si por meio de relações, estruturadas em unidades compostas por sujeito, predicado e objeto, o que permite a formação de redes de conhecimento interligadas (DZIEKANIAK; KIRINUS, 2004).

Com base nos mesmos dados da Figura 10.4, que apresenta uma representação de ontologia voltada à interpretação humana, é possível construir uma versão formal para máquinas, expressa em triplas `<sujeito, predicado, objeto>`. Essa estrutura tripla, típica do RDF, permite a modelagem de conceitos e de seus relacionamentos, em conformidade com os princípios das ontologias da Web Semântica (MANOLA et al., 2004; DZIEKANIAK; KIRINUS, 2004). A Tabela 10.3 descreve os três componentes fundamentais de uma tripla RDF. Cada uma representa uma unidade de conhecimento, na qual o sujeito identifica um recurso, o predicado expressa uma propriedade ou relação, e o objeto representa o valor ou o recurso associado.

Essa estrutura permite a descrição de relações simples, como `‘‘Pessoa → gosta de → Comida’’`. Para um exemplo mais complexo sobre uma pessoa com múltiplas propriedades associadas, considere as seguintes triplas:

- `<‘‘Pessoa’’, ‘‘tem nome’’, ‘‘Alanna Maria’’>`
- `<‘‘Pessoa’’, ‘‘trabalha em’’, ‘‘Empresa XYZ’’>`
- `<‘‘Pessoa’’, ‘‘mora em’’, ‘‘Sobral’’>`

Essas triplas formam um grafo RDF que descreve a pessoa ”Alanna Maria”, permitindo interpretação automatizada por sistemas. O sujeito ”Pessoa”está relacionado a ”Sobral”por meio da propriedade ”mora em”, o que constrói uma

relação estruturada. Esse conjunto pode ser representado graficamente como um grafo orientado, em que os nós representam conceitos e as arestas representam os relacionamentos (MANOLA et al., 2004; GUTIERREZ et al., 2006).

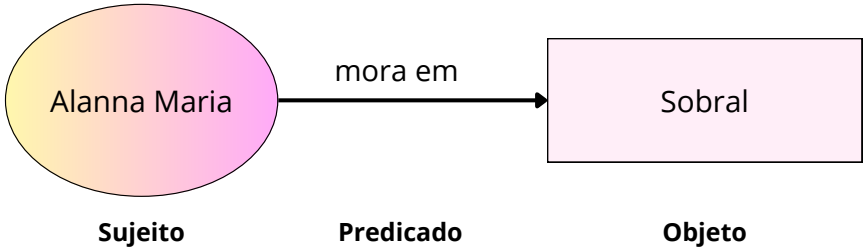


Figura 10.11: Estrutura RDF para representação de relações entre conceitos

Na Figura 10.11, os recursos são representados por elipses e os valores literais por retângulos. O RDF adota uma semântica bem definida, na qual cada tripla constitui um grafo orientado, com o sujeito como nó de origem, o objeto como nó de destino e o predicado como a aresta nomeada entre eles.

Além da representação gráfica, o RDF pode ser serializado em diferentes formatos, sendo o RDF/XML o primeiro padrão definido. Esse formato facilita a interoperabilidade entre sistemas ao padronizar a descrição de dados. Aplicações típicas incluem redes sociais semânticas, integração de contatos em sistemas distribuídos e processamento automático de informações pessoais (SOUZA; ALVARENGA, 2004).

Outra alternativa para representá-lo é o uso de formatos de serialização, sendo o RDF/XML o primeiro padrão estabelecido. Desenvolvido como sintaxe original para RDF, esse formato facilita a interoperabilidade entre diferentes sistemas ao descrever dados de forma padronizada e sem ambiguidades. Aplicações comuns incluem redes sociais semânticas, integração de dados de contato em sistemas distribuídos e aplicações que precisam processar automaticamente contatos de pessoas (SOUZA; ALVARENGA, 2004). A Listagem 10.6 apresenta um exemplo de um documento com anotações semânticas:

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/">

    <foaf:Person rdf:about="#alanna">
        <foaf:name>Alanna Maria</foaf:name>
        <foaf:mbox rdf:resource="mailto:alanna.maria@gmail.com"/>
        <foaf:age>25</foaf:age>
        <foaf:gender>female</foaf:gender>
    </foaf:Person>
</rdf:RDF>
```

Listagem 10.6: Exemplo de serialização RDF/XML usando vocabulário FOAF

Esse código apresenta:

- **Linha 1**: Declaração XML padrão.
- **Linhas 2-3**: Definição dos namespaces: `rdf:` - Namespace básico do RDF. `foaf:` - Vocabulário FOAF para descrição de pessoas.
- **Linha 4**: Instância de Pessoa `foaf:Person` com: `rdf:about` - URI único identificando o recurso.
- **Linha 5**: Propriedade `foaf:name` com valor literal, nome completo.
- **Linha 6**: Propriedade `foaf:mbox` vinculada a recurso externo via URI.
- **Linha 7**: Propriedade `foaf:age` com valor literal, idade.
- **Linha 8**: Propriedade `foaf:gender` com valor padronizado, gênero.
- **Linha 9**: Fechamento da estrutura RDF.

Apesar de sua utilidade, o RDF/XML apresenta desafios, como a verbosidade e a menor legibilidade em comparação com outras formas de serialização, como o Turtle.

A evolução dessa estrutura levou ao conceito de Linked Data, que se concentra na interligação de conjuntos de dados por meio de URIs. Essa abordagem permite conexões dinâmicas entre informações diversas, cada uma identificada de forma única na web. O projeto DBpedia, extraído da Wikipédia, é um dos mais conhecidos e serve como ponto central da Linked Open Data Cloud, com milhões de entidades e bilhões de triplas interconectadas (HITZLER, 2021).

O RDF é especialmente útil nesse contexto por sua capacidade de descrever qualquer entidade, desde que possua um identificador na web (DZIEKANIAK; KIRINUS, 2004; SILVA, 2015). Por exemplo:

- `<https://exemplo.com/recursos/Alanna>`, representa “Alanna”, a pessoa descrita.

- `<https://exemplo.com/propriedades/moraEm>`, define a propriedade “mora em” (propriedade).
- `<https://exemplo.com/recursos/Sobral>`, identifica a cidade “Sobral”, onde Alanna mora.

Assim, o RDF estabelece-se como um modelo simples, porém poderoso, para a descrição semântica de dados na web, permitindo a integração, o compartilhamento e a interpretação automatizada de informações em larga escala.

10.3.2 RDF Schema (RDFS)

O RDF Schema (RDFS) é uma extensão do RDF que permite definir vocabulários semânticos, possibilitando a criação de estruturas organizadas de classes, propriedades e hierarquias no modelo RDF. Ambas as tecnologias foram desenvolvidas com o propósito de promover o compartilhamento descentralizado de informações e vocabulários na web. Assim como qualquer usuário pode criar uma página na web e conectá-la a outras, o RDFS adota o mesmo princípio, permitindo que qualquer pessoa publique informações sobre qualquer tema de forma estruturada.

Enquanto o RDF se limita à descrição básica de recursos por meio de triplas `<sujeito, predicado, objeto>`, o RDFS introduz mecanismos para atribuir semântica a essas relações, incluindo a definição de tipos de entidades (classes), propriedades que conectam recursos e regras de herança, promovendo uma organização conceitual mais robusta (GUTIERREZ et al., 2006; MCBRIDE, 2004).

A Tabela 10.4 apresenta os principais elementos do RDFS. Esses componentes permitem criar vocabulários mais expressivos, definir hierarquias e adicionar descrições legíveis para humanos, favorecendo tanto a interpretação automática quanto a compreensão por parte de desenvolvedores e usuários.

Com base nesses elementos, é possível estruturar semanticamente uma descrição para a pessoa “Alanna Maria”. Primeiro, define-se a classe “Pessoa” com o uso de `rdfs:Class`. Em seguida, cria-se a instância `ex:Alanna` como um recurso do tipo “Pessoa” utilizando `rdf:type`. As propriedades como “hasName”, “hasAgeehasGender” são então associadas a essa instância, cada uma com seus domínios e intervalos definidos:

- A propriedade `hasName` tem como domínio a classe `Pessoa` e como intervalo um literal do tipo `xsd:string`;
- A propriedade `hasAge` tem como intervalo `xsd:integer`;
- A propriedade `hasGender` pode ter como intervalo uma instância de uma classe, por exemplo, `Gênero`.

Assim, ao criar o recurso `ex:Alanna`, atribui-se:

- `ex:Alanna rdf:type ex:Pessoa`

Tabela 10.4: Recursos e suas características em RDF

Elemento do RDFS	**Descrição**	**Exemplo(s)**
Classes	Permite categorizar recursos em diferentes tipos.	`rdfs:Class` define que um recurso é uma classe.
Propriedades	Definem relações entre os recursos.	`rdfs:domain` e `rdfs:range` especificam o tipo de recurso que pode ser usado como sujeito ou como objeto de uma propriedade.
Herança	Permite a definição de hierarquias de classes e propriedades.	`rdfs:subClassOf`: define que uma classe é uma subclasse de outra. `rdfs:subPropertyOf`: define que uma propriedade é uma subpropriedade de outra.
Rótulos e comentários	Proporciona descrições textuais e nomes legíveis para humanos.	`rdfs:label`: fornece um rótulo amigável para o recurso. `rdfs:comment`: permite adicionar comentários sobre o recurso ou propriedade.
Tipo de recurso	Define o tipo de um recurso específico.	`rdf:type` indica que um recurso é uma instância de uma classe.

- `ex:Alanna ex:hasName "Alanna Maria"`
- `ex:Alanna ex:hasAge "25"8sd:integer`
- `ex:Alanna ex:hasGender "Feminino"`

Esse modelo não apenas estrutura os dados de forma clara, mas também estabelece suas relações semânticas, tornando possível o compartilhamento e a reutilização dos dados em diferentes contextos e aplicações (DECKER et al., 2000).

10.3.3 JavaScript Object Notation for Linked Data (JSON-LD)

O JSON-LD (JavaScript Object Notation for Linked Data) é uma tecnologia projetada para representar dados estruturados com significado semântico, em conformidade com os princípios do Linked Data. Ele funciona como uma implementação prática desses princípios, desenvolvida para facilitar a publicação e o consumo de dados interligados na web. Utilizando o formato leve e legível do JSON, amplamente empregado em APIs e aplicações web, o JSON-LD oferece uma maneira eficiente de descrever informações interpretáveis por máquinas. Sua estrutura permite identificar recursos por meio de URIs e descrever as relações entre eles com base no modelo RDF, favorecendo a interoperabilidade entre sistemas e contribuindo para a construção de uma web mais conectada (CONSORTIUM, 2020).

Além disso, o JSON-LD permite a inclusão de metadados enriquecidos em páginas da web, como dados sobre produtos, eventos, receitas e perfis pessoais ou empresariais. Isso possibilita que mecanismos de busca, como o Google, compreendam melhor o conteúdo das páginas e apresentem informações estruturadas diretamente nos resultados de pesquisa, como avaliações, horários e localizações (CONSORTIUM, 2020). O uso dessa tecnologia torna os conceitos da Web Semântica mais acessíveis e aplicáveis no cotidiano dos desenvolvedores.

A Listagem 10.7 apresenta um exemplo prático de anotação semântica com JSON-LD, utilizando o vocabulário do Schema.org para descrever uma relação entre duas pessoas:

```
{
    "@context": "https://schema.org",
    "@type": "Person",
    "name": "Guiomar Machado",
    "knows": {
        "@type": "Person",
        "name": "Mairton Paiva"
    }
}
```

Listagem 10.7: Exemplo prático de JSON-LD aplicando princípios de Linked Data

10.3.4 Extensible Markup Language (XML)

De acordo com Consortium et al. (2006), em "Extensible Markup Language (XML) 1.1 (Second Edition)", a Linguagem de Marcação Extensível (XML) foi desenvolvida com o objetivo principal de representar dados de forma estruturada e hierárquica. Trata-se de uma linguagem utilizada para descrever documentos que armazenam e transferem dados entre diferentes sistemas. O XML também oferece diretrizes parciais sobre como os softwares devem processar esses documentos. O componente responsável por essa tarefa é o processador XML, encarregado de ler os documentos e de fornecer acesso à sua estrutura e ao seu conteúdo.

A estrutura do XML baseia-se em elementos organizados por meio de tags, em que cada tag representa um dado ou um agrupamento de dados com significado específico no contexto do documento. Apesar de sua ampla utilização na web, o XML não incorpora semântica própria, o que significa que a interpretação dos dados depende inteiramente do contexto da aplicação que os utiliza (ALMEIDA, 2002). Por isso, embora o XML possa ser empregado em conjunto com tecnologias da Web Semântica (como o RDF/XML), ele não é, por si só, uma tecnologia semântica. A Listagem 10.8 apresenta um exemplo simplificado de um fragmento XML:

```
<?xml version="1.0" encoding="UTF-8"?>
    <pessoa>
        <nome>Alanna Maria</nome>
        <email>alanna.maria@example.com</email>
    </pessoa>
```

Listagem 10.8: Exemplo simplificado de um fragmento XML

10.3.5 Web Ontology Language (OWL)

A Web Ontology Language (OWL) é uma linguagem fundamental para a representação de ontologias na Web Semântica. Seu formato original, baseado na sintaxe RDF/XML, embora amplamente adotado, apresenta desafios consideráveis de legibilidade devido à complexidade da estrutura.

Para contornar essas limitações, foram desenvolvidas alternativas sintáticas mais acessíveis. Uma delas é a sintaxe XML simplificada, que mantém a estrutura em tags, mas remove as regras rígidas do RDF, tornando-a mais intuitiva para leitura humana. Outra é a sintaxe abstrata, empregada na documentação formal da OWL, mais compacta e adequada a contextos técnicos. Além disso, há a sintaxe gráfica baseada na UML (Unified Modeling Language), que aproveita a familiaridade dos desenvolvedores com a modelagem para facilitar a compreensão e a comunicação entre os stakeholders. Essas variações atendem às necessidades de diferentes públicos, sem perder a expressividade dos construtos semânticos da OWL (ANTONIOU; HARMELEN, 2009).

Entre os principais componentes da OWL, destacam-se:

Classes: Estruturas hierárquicas que organizam conceitos por meio da propriedade `rdfs:subClassOf`. A Listagem 10.9 apresenta um exemplo em que "Professor" é definido como subclasse de "Pessoa", indicando que todo professor também é uma pessoa.

```
<owl:Class rdf:about="http://exemplo#Pessoa">
    <rdfs:label>Pessoa</rdfs:label>
</owl:Class>
<owl:Class rdf:about="http://exemplo#Professor">
    <rdfs:subClassOf rdf:resource="http://exemplo#Pessoa"/>
</owl:Class>
```

Listagem 10.9: Exemplo de Classes

Instâncias: Indivíduos concretos que pertencem a classes. A Listagem 10.10, apresenta um exemplo em que "Matheus" é uma instância da classe Pessoa.

```
<owl:NamedIndividual rdf:about="http://exemplo#Matheus">
        <rdf:type rdf:resource="#Pessoa"/>
        <rdfs:label>Manoel Matheus</rdfs:label>
    </owl:NamedIndividual>
```

Listagem 10.10: Instância de indivíduo em OWL

Propriedades de Objeto: Estabelecem relações entre indivíduos. Essas propriedades podem apresentar características específicas, como hierarquia. A Listagem 10.11 apresenta um exemplo da propriedade "temMãe", com domínio e intervalo definidos como "Pessoa".

```
<owl:ObjectProperty rdf:about="http://exemplo#temMãe">
        <rdfs:domain rdf:resource="http://exemplo#Pessoa"/>
        <rdfs:range rdf:resource="http://exemplo#Pessoa"/>
        <rdfs:label>tem mãe</rdfs:label>
    </owl:ObjectProperty>
```

Listagem 10.11: Propriedade de objeto em OWL

Propriedades de Dados: Ligam indivíduos a valores literais, como números, datas ou strings. A Listagem 10.12 apresenta um exemplo de propriedade que associa uma pessoa à sua idade, validando o tipo do dado com `xsd:integer`.

```
<owl:DatatypeProperty rdf:about="http://exemplo#temIdade">
        <rdfs:domain rdf:resource="http://exemplo#Pessoa"/>
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
            integer"/>
        <rdfs:label>tem idade</rdfs:label>
    </owl:DatatypeProperty>
```

Listagem 10.12: Propriedade de dados em OWL

Ao contrário de linguagens como XML ou até mesmo RDF, a OWL foi concebida para cenários em que as informações precisam ser processadas por softwares, não apenas visualizadas por humanos. Ela permite criar ontologias ricas e precisas, melhorando a capacidade das máquinas de compreender o conteúdo da web. Isso é possível graças a um vocabulário mais expressivo e a uma semântica formal que permite raciocínios automatizados sobre os dados (BECHHOFER et al., 2004).

A OWL apresenta três sublinguagens, cada uma mais expressiva do que a anterior: OWL Lite, OWL DL e OWL Full. Essas variantes expandem o vocabulário disponível para descrever propriedades e classes, incluindo relações entre classes, cardinalidade, igualdade, além de uma tipagem mais detalhada de propriedades e características enumeradas para propriedades e classes. Além disso, o OWL oferece uma semântica formal que permite que máquinas realizem raciocínios sobre o significado do conteúdo, um recurso ausente em XML ou RDF. Esse aspecto facilita o processamento e a integração automatizados de informações disponíveis na web, alinhando-se ao propósito central da Web Semântica (MCGUINNESS et al., 2004; BECHHOFER et al., 2004). A Tabela 10.5 contém uma breve descrição de cada um destes tipos de Sub-linguagem OWL:

A organização da OWL em diferentes níveis de expressividade permite que ela atenda a uma ampla gama de aplicações, desde as mais simples até as que exigem modelagens complexas. Em todos os casos, o objetivo é ampliar a capacidade das máquinas de interpretar, integrar e inferir conhecimento a partir dos dados disponíveis na web.

Tabela 10.5: Descrição das sublinguagens do OWL

Sub-linguagem	Descrição
OWL Lite	Voltada para aplicações simples, oferece suporte à hierarquia de classes e propriedades, a restrições básicas e a tipos de dados elementares. Equilibra expressividade e complexidade.
OWL DL	Intermediária em termos de expressividade. Preserva a decidibilidade e completude dos raciocínios, sendo adequada para aplicações mais robustas.
OWL Full	Sub-linguagem mais expressiva. Oferece máxima flexibilidade, permitindo o uso completo do RDF sem restrições sintáticas, embora sacrifique a garantia de decidibilidade.

10.3.6 Simple Knowledge Organization System (SKOS)

O Simple Knowledge Organization System (SKOS) é um modelo de dados desenvolvido pela W3C para representar sistemas de organização do conhecimento, como tesauros, taxonomias, esquemas de classificação e vocabulários controlados, de forma padronizada e interoperável na web. Seu propósito principal é facilitar a representação, reutilização e o compartilhamento desses esquemas, especialmente no contexto da Web Semântica e do Linked Data. Surgiu como uma alternativa simplificada para adaptar os tesauros tradicionais. Enquanto os tesauros clássicos são amplamente utilizados na indexação e recuperação da informação, o SKOS oferece uma camada semântica especializada, baseada em RDF, que permite que esses sistemas sejam interpretados por máquinas e vinculados a outros dados na web (MILES et al., 2005).

Em outras palavras, o SKOS funciona como um vocabulário RDF específico para a organização do conhecimento, oferecendo estruturas prontas para uso. Já o RDF genérico exige que todos os elementos sejam definidos do zero. O modelo SKOS fornece um conjunto de classes e propriedades RDF que permite representar conceitos e suas relações (como hierarquias e associações) em forma de grafos conectáveis na web (MILES et al., 2005), por exemplo:

- Conceitos (skos:Concept), como entidades básicas (ex.: Inteligência Artificial).
- Rótulos (skos:prefLabel para termos preferenciais, skos:altLabel para sinônimos).
- Relações semânticas (skos:broader/narrower para hierarquias, skos:related para associações).

Essas propriedades permitem modelar sistemas de organização do conhecimento tradicional na Web Semântica, como:

Tesauros: Utiliza `skos:prefLabel` para indicar o termo preferido de um conceito, `skos:altLabel` para representar sinônimos ou variantes, e `skos:related` para estabelecer relações associativas entre conceitos não hierárquicos, conforme apresentado na Listagem 10.13.

```
    :musica a skos:Concept ;
    skos:prefLabel "Música"@pt ;
    skos:altLabel "Som musical"@pt ;
    skos:related :danca .
```

Listagem 10.13: Exemplo de tesauro em SKOS

Taxonomias: Estrutura conceitos em hierarquias com o uso de `skos:broader` para conceitos mais amplos e `skos:narrower` para conceitos mais específicos, conforme observado na Listagem 10.14.

```
    :animal a skos:Concept ;
    skos:prefLabel "Animal"@pt ;
    skos:narrower :mamifero .
```

Listagem 10.14: Exemplo de taxonomia em SKOS

Glossários: Aplica `skos:definition` para fornecer definições formais dos conceitos, conforme observado na Listagem 10.15.

```
    :internet a skos:Concept ;
    skos:prefLabel "Internet"@pt ;
    skos:definition "Rede global de computadores que
        permite a comunicação e troca de dados."@pt .
```

Listagem 10.15: Exemplo de glossário em SKOS

Esquemas de Classificação: Utiliza `skos:ConceptScheme` para organizar conjuntos de conceitos relacionados, e propriedades como `skos:hasTopConcept` e `skos:inScheme` para associá-los à estrutura geral, conforme observado na Listagem 10.16.

```
web:taxonomiaWeb a skos:ConceptScheme ;
    skos:prefLabel "Taxonomia de Tecnologias Web"@pt ;
    skos:hasTopConcept web:frontend, web:backend, web:protocolos ;
    skos:note "Classificação de tecnologias para desenvolvimento web
        moderno"@pt .

web:frontend a skos:Concept ;
    skos:prefLabel "Frontend"@pt ;
    skos:narrower web:html, web:css, web:javascript ;
    skos:definition "Tecnologias executadas no navegador do usuário"
        @pt .
```

Listagem 10.16: Exemplo de esquema de classificação em SKOS

O SKOS apresenta uma linguagem simples, adequada tanto para especialistas quanto para desenvolvedores, permitindo a criação e o compartilhamento de novos esquemas em um formato legível por máquinas. Sua adoção favorece a interoperabilidade entre sistemas e a publicação de dados em formatos padronizados. Além disso, o SKOS pode ser utilizado isoladamente ou em conjunto com linguagens mais formais, como a OWL (MILES; BECHHOFER, 2009).

Embora o SKOS e a OWL sejam voltados à modelagem de conhecimento, suas finalidades são distintas. O SKOS é mais indicado para estruturas conceituais simples, como relações entre termos em taxonomias e vocabulários controlados. Ele permite, por exemplo, modelar a ideia de que "Instrumento musical"é mais geral do que "Violino", sem recorrer a lógica formal ou propriedades personalizadas. Por esse motivo, o SKOS é ideal para representar domínios de menor complexidade semântica, mantendo a clareza e a interoperabilidade necessárias à Web Semântica.

10.3.7 Shapes Constraint Language (SHACL)

O SHACL (Shapes Constraint Language) é uma linguagem padrão da W3C para descrever e validar gráficos RDF como ontologias, grafos de conhecimento ou *datasets* semânticos em relação a um conjunto de condições, Surgiu com a necessidade de especificar restrições em grafos RDF e mecanismos para detectar violações dessas restrições. Essas condições são fornecidas como *shapes* (formas) e outros elementos representados como grafos RDF, esses grafos validados são chamados de data graphs (grafo de dados). O SHACL funciona como um "fiscal de qualidade", assegurando que os dados em grafos RDF sigam regras específicas (validação da estrutura conforme o modelo esperado). Além disso, essas regras *(shapes)* também podem ser usadas para descrever grafos válidos, com aplicações que vão além da validação, como: construção de interfaces de usuário, geração de código e integração de dados (W3C, 2017). Na Figura 10.12 é possível visualizar um exemplo análogo para compreender a definição.

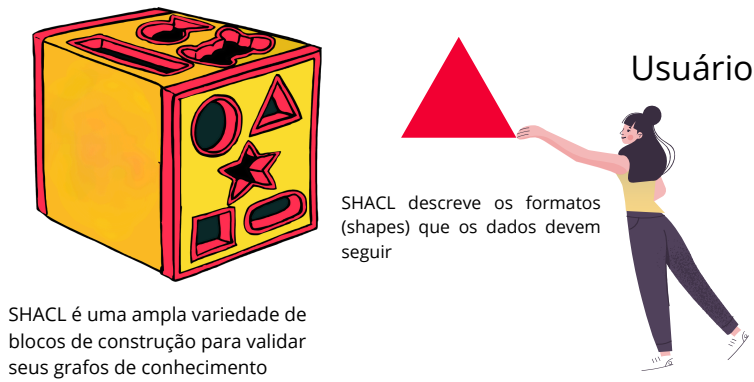


Figura 10.12: Analogia de representação do SHACL

A linguagem é estruturada com base em três conceitos fundamentais (CORMAN et al., 2018; W3C, 2017):

Shapes e Targets

- **Shapes:** Representam as regras que os dados devem seguir. Podem ser *Node Shapes*, aplicáveis a nós (ex.: `sh:NodeShape`), ou *Property Shapes*, aplicáveis a propriedades.
- **Targets:** Definem a quais partes do grafo as formas se aplicam. Por exemplo: `sh:targetClass` (para todos os nós de uma classe), `sh:targetNode` (para nós específicos) e `sh:targetSubjectsOf` (para sujeitos de uma propriedade).

Constraints (Restrições)

- Restrições de propriedade, como `sh:minCount` e `sh:datatype`, controlam a cardinalidade e os tipos de dados.
- Restrições lógicas, como `sh:and`, `sh:or` e `sh:not`.
- Restrições de valor, como `sh:hasValue` e `sh:in`.
- Restrições baseadas em SPARQL, por meio de `sh:sparql`.

Validação e Relatórios

- Um validador SHACL verifica se o grafo de dados está conforme as formas especificadas.
- Em caso de violação, um relatório detalhado é gerado, indicando os pontos de não conformidade e os motivos.

Considere o exemplo RDF da Listagem 10.6, que descreve uma pessoa usando o vocabulário FOAF. Suponha que se deseje garantir que todo indivíduo do tipo `foaf:Person` possua exatamente uma propriedade `foaf:name` (obrigatória, do tipo `xsd:string`) e, opcionalmente, uma propriedade `foaf:age`, do tipo `xsd:integer` e com valor maior ou igual a zero. O seguinte *shape* SHACL formaliza essas regras de validação:

```
1
2  @prefix sh: <http://www.w3.org/ns/shacl#> .
3  @prefix foaf: <http://xmlns.com/foaf/0.1/> .
4  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
5
6  foaf:PersonShape
7      a sh:NodeShape ;
8      sh:targetClass foaf:Person ;
9
10     # Regra para o nome (obrigatório)
11     sh:property [
12         sh:path foaf:name ;
13         sh:minCount 1 ;
14         sh:maxCount 1 ;
15         sh:datatype xsd:string ;
16         sh:message "Erro: Cada pessoa deve ter exatamente
17         um nome (string)." ;
18                 ] ;
19
20         # Regra para idade (opcional, se existir)
21         sh:property [
22             sh:path foaf:age ;
23             sh:datatype xsd:integer ;
24             sh:minInclusive 0 ;
25             sh:message "Erro: Idade deve ser um inteiro
26             positivo." ;
27         ] ;
```

Listagem 10.17: Shape SHACL para validação de dados pessoais em FOAF

A Listagem 10.17 apresenta o *Shape SHACL* para validação de dados pessoais ilustra de maneira concreta o poder desta linguagem na garantia da qualidade de dados semânticos. Como demonstrado no código, a validação opera em três níveis fundamentais (CORMAN et al., 2018; W3C, 2017):

Validação Estrutural: Por meio das propriedades sh:minCount e sh:maxCount, o *shape* impõe restrições de cardinalidade, garantindo que cada indivíduo do tipo foaf:Person possua exatamente um nome. Essa validação previne problemas comuns como dados incompletos ou duplicados.

Validação de Tipagem: O uso explícito de sh:datatype com tipos XML Schema (xsd:string, xsd:integer) assegura a integridade sintática dos valores, evita inconsistências como números representados como strings ou formatos de data inválidos.

Validação Semântica: A restrição sh:minInclusive aplicada à propriedade foaf:age demonstra como SHACL vai além da sintaxe ao incorporar regras de negócio diretamente no processo de validação.

Quando aplicado a grafos RDF, este mecanismo gera relatórios de validação detalhados que desempenham um papel duplo: técnico e documental. Cada violação é acompanhada de mensagens claras (como “Erro: Idade deve ser um inteiro positivo”), que servem tanto para correção imediata quanto para documentação

viva das regras de qualidade. O SHACL é uma ferramenta poderosa para garantir a qualidade dos dados semânticos, sendo amplamente adotado em projetos de dados vinculados e em ontologias. Seu uso facilita a detecção de inconsistências e a manutenção de conjuntos de dados confiáveis.

10.3.8 Protocol and Resource Description Framework Query Language (SPARQL)

O SPARQL Protocol and RDF Query Language é a linguagem padrão definida pelo W3C para consultas em bases de dados que utilizam o modelo RDF. Por meio do SPARQL, é possível realizar consultas complexas, tais como: recuperação de dados, junções entre diferentes conjuntos e aplicação de filtros. O SPARQL utiliza uma sintaxe semelhante à do SQL, porém adaptada ao modelo baseado em grafos do RDF. Essa linguagem é amplamente utilizada para consultar fontes de dados semânticas, como triplas RDF, ontologias, dados de *Linked Data* e bases de conhecimento acessíveis por meio de *endpoints* SPARQL (PÉREZ et al., 2009).

Uma consulta SPARQL é composta por padrões de triplas, operadores lógicos, filtros e padrões opcionais. Existem atualmente quatro tipos principais de consulta (SILVA, 2015):

- **SELECT**: Recupera informações diretamente de um endpoint SPARQL. Os resultados são apresentados em formato tabular, refletindo o estado atual dos dados..
- **CONSTRUCT**: Extrai informações e as reestrutura em um novo grafo RDF, permitindo a transformação dos dados com base em padrões definidos.
- **ASK**: Retorna um valor booleano (verdadeiro ou falso), indicando se os dados consultados satisfazem as condições especificadas.
- **DESCRIBE**: Fornece um grafo RDF com informações relevantes sobre um recurso. O conteúdo retornado depende da implementação do endpoint consultado.

A Listagem 10.18 apresenta um exemplo de consulta SPARQL do tipo *SELECT*, que retorna nomes e endereços de e-mail de indivíduos descritos em um grafo RDF utilizando o vocabulário FOAF (Friend of a Friend).

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?email
WHERE {
    ?person foaf:name ?name ;
            foaf:mbox ?email .
}
```

Listagem 10.18: Exemplo de uma consulta SPARQL do tipo SELECT

Na consulta apresentada (Listagem 10.18), a cláusula `PREFIX` define um prefixo para facilitar o uso do namespace FOAF. A cláusula `SELECT` especifica as variáveis a serem retornadas, enquanto a cláusula `WHERE` estabelece os padrões a que os dados devem corresponder. Nesse caso, a variável `?person` representa um recurso que possui um nome e um endereço de e-mail descritos pelas propriedades `foaf:name` e `foaf:mbox`.

10.4 Web Semântica no cenário atual

A Web Semântica continua sendo uma área relevante de pesquisa e aplicação, com papel fundamental no suporte a novas tecnologias e na criação de aplicações mais ricas e interativas (NECULA, 2020). Ao promover a estruturação e organização dos dados, essa abordagem permite que diferentes sistemas e tecnologias se integrem de maneira mais eficiente, resultando em experiências de usuário mais aprimoradas, práticas e acessíveis (ISOTANI et al., 2009).

Atualmente, seu uso é amplamente observado em aplicações práticas e setores estratégicos, como saúde, educação e inteligência artificial, ao facilitar a criação de sistemas de dados interligados e interoperáveis. Com a consolidação de padrões como RDF, OWL e SPARQL, o desenvolvimento de ferramentas e plataformas tem se tornado mais acessível, impulsionando iniciativas como o *Linked Open Data (LOD)*. Um exemplo notável é o projeto *DBpedia*, que organiza informações estruturadas extraídas da Wikipédia, tornando-as acessíveis e reutilizáveis para diferentes sistemas.

Além disso, a Web Semântica vem sendo progressivamente integrada a tecnologias emergentes, como a Inteligência Artificial (IA) e o *Big Data*. Assistentes virtuais, como Siri e Alexa, bem como sistemas de busca semântica, como o Google, aplicam conceitos semânticos para fornecer respostas mais precisas e contextualizadas. Isso aprimora significativamente os processos de recuperação da informação e otimiza a experiência do usuário (SOUZA; ALVARENGA, 2004; SEGUNDO, 2015; HITZLER, 2021).

No contexto educacional, conforme destacado por Isotani et al. (2009) no artigo *"Estado da Arte em Web Semântica e Web 2.0: Potencialidades e Tendências"*, a Web Semântica tem sido incorporada aos princípios da Web 2.0, que prioriza a interação social e a colaboração. Essa convergência possibilita a criação de ambientes de aprendizagem mais dinâmicos, com sistemas que utilizam ontologias e metadados

para classificar, organizar e recomendar conteúdos com base no perfil de cada aluno. Plataformas educacionais semânticas são capazes de sugerir leituras e atividades personalizadas com base no desempenho e nas preferências de aprendizagem dos usuários, promovendo uma educação mais eficiente e colaborativa.

Diante das constantes inovações tecnológicas, a Web Semântica se consolida como uma ferramenta crítica para enfrentar os desafios relacionados à organização e à integração da informação na era digital. Seu papel vai além da simples interligação de dados, atuando como catalisador para a criação de ambientes mais inteligentes, colaborativos e personalizados, contribuindo para a construção de uma internet mais conectada e centrada no usuário.

Capítulo 11

Teste de Software para Web

Raíssa Ellen de Sousa (UFC)

Cleiton Tavares (PUC-MG)

Euler Horta Marinho (UFOP)

Fischer Ferreira (Unifei)

Este capítulo fornece uma visão geral sobre teste de software. Portanto, são abordados os seguintes tópicos: teste de *software* (Seção 11.1), tipos de testes de *software* (Seção 11.2), automação de testes (Seção 11.3), bem como apresentar as ferramentas de automação de testes *Selenium* (Seção 11.4) e *Cypress* (Seção 11.5).

11.1 Teste de *software*

Antigamente, pouco se aplicavam os conceitos e a execução de testes ao longo do ciclo de desenvolvimento de um *software*. Não havia um profissional especificamente direcionado para a área de qualidade. Essa atividade muitas vezes era vista como uma tarefa secundária, sem grande importância ou prioridade, e realizada pelos próprios desenvolvedores dos sistemas. Ao longo do tempo, com o aumento da concorrência no mercado de tecnologia e do nível de exigência em relação à qualidade, surgiu a necessidade de olhar os testes de *software* com outros olhos. Hoje, existem inúmeras ferramentas, aplicações, cursos e uma vasta literatura sobre o tema. Assim como o mercado de trabalho passou a demandar profissionais específicos para as áreas de qualidade e de testes.

O teste de *software* tem como objetivo fornecer evidências da confiabilidade do *software*, complementando outras atividades, como revisões e técnicas formais e rigorosas de especificação e verificação (MALDONADO; JINO, 1991). Teste de software pode ser definido como um processo ou uma série de processos, realizados com a finalidade de garantir que o *software* atenda às funcionalidades para as quais foi projetado (MYERS et al., 2011). A literatura apresenta algumas definições para teste de software recorrentemente citadas: "Testar *software* significa verificar por meio de uma execução controlada se o seu comportamento corre de acordo com o especificado" (NETO; CLAUDIO, 2007). "Teste de *software* é um elemento crítico da garantia de qualidade de *software* e representa a revisão final da especificação, projeto e geração de código" (PRESSMAN; MAXIM, 2021). Dessa forma, ao realizar os testes em um *software*, analisam-se itens e recursos para detectar possíveis diferenças entre as condições existentes e as necessárias (ISO/IEC/IEEE:24765, 2017).

Para compreender melhor a atividade de teste de *software* e estabelecer uma linguagem comum, são estabelecidos na literatura alguns conceitos importantes: defeito, erro, falha e engano (DELAMARO et al., 2017). As definições utilizadas neste capítulo seguem o documento "*Standard Glossary of Terms used in Software Testing*" (BOARD, 2014)[1] e a terminologia padrão para Engenharia de Software do *Institute of Electrical and Electronics Engeneers* (ISO/IEC/IEEE:24765, 2017).

- Defeito (do inglês, *fault*): pode ser conceituado como um passo ou definição de dados incorretos (DELAMARO et al., 2017). Pode ser encontrado du-

[1] O ISTQB®(*International Software Testing Qualifications Board*) é uma associação sem fins lucrativos, que se tornou líder mundial na certificação de competências em testes de software. ⟨https://www.istqb.org/⟩

rante a execução do *software*, podendo causar falhas em componentes ou sistema (BOARD, 2014).

- Erro (do inglês, *error*): pode ser caracterizado como um estado inconsistente ou inesperado, ocasionado pela manifestação concreta de um defeito durante a execução de um programa (DELAMARO et al., 2017). Ou seja, qualquer inconsistência que pode afetar a funcionalidade do sistema. (ISO/IEC/IEEE:24765, 2017).
- Falha (do inglês, *failure*): é quando o comportamento produzido pelo *software* difere do resultado esperado (DELAMARO et al., 2017).
- Engano (do inglês, *mistake*): pode ser definido como uma ação humana que produz um defeito (DELAMARO et al., 2017), uma compreensão incorreta (ISO/IEC/IEEE:24765, 2017).

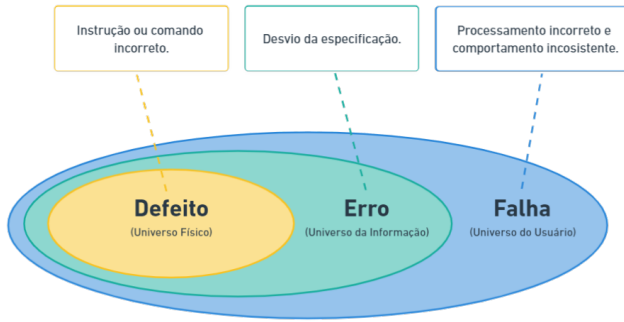


Figura 11.1: Diferença entre os conceitos de defeito, erro e falha

A Figura 11.1 expressa a diferença entre esses conceitos. Os defeitos estão relacionados a aplicação propriamente dita, e podem ser causados por usuários, por exemplo, por meio do mal uso de uma tecnologia. Assim, os mesmos podem refletir no surgimento de erros em um produto, ou seja, a construção de um *software* de forma diferente ao que foi especificado. Por fim, os erros podem ocasionar falhas, que consistem em comportamentos inesperados que podem inviabilizar a utilização de uma aplicação, afetando diretamente o usuário final. O foco deste capítulo será as falhas, já que estas são detectadas por meio do comportamento do sistema. Desvios na especificação (erros) e imperfeições no código (defeitos) não fazem parte do escopo desse capítulo.

Os testes são geralmente aplicados em diferentes níveis e devem ser considerados em paralelo ao desenvolvimento do *software*. Na literatura, são definidos os principais níveis de teste de *software*, que são: teste de unidade, teste de integração, teste de sistema e teste de aceitação. A Tabela 11.1 apresenta uma comparação entre esses

Tabela 11.1: Níveis de testes de software

Critério	Unidade	Integração	Sistema	Aceitação
Finalidade	O funcionamento correto da unidade /módulo	O funcionamento correto das unidades integradas	Todo o sistema funciona bem quando integrado	As expectativas do cliente são atendidas
Foco	Menor parte testável	Interface e interação de módulos	Interação e funciona - mento de todos os módulos como um	Software funcionando de acordo com as especificações fornecidas
Tempo de teste	Uma vez que um novo código é escrito	Uma vez que novos componentes são adicionados	Assim que o software estiver completo	Uma vez que o software é lido operacional- mente
Executado por	Desenvolvedor	Equipe de desenvolvimento	Equipe de teste	A equipe de desenvolvi- mento e os usuários finais
Técnicas de teste	Geralmente Caixa Branca e Caixa Cinza	Caixa Branca e Caixa Preta	Geralmente Caixa Preta e Caixa Cinza	Teste de Caixa Preta
Automação	Automatizável usando JUnit, PHPUnit, TestNG etc	Automatizável usando Soap UI, Rest Client etc	Automatizável usando Webdriver	Automatizável usando o Cucumber

níveis de testes, levando em consideração determinados critérios. Na primeira linha, consideramos a finalidade de cada um dos tipos de testes. Por exemplo, os testes de unidade têm a finalidade de garantir o funcionamento correto de cada unidade.

- Teste de Unidade: também conhecido como testes unitários, enfatiza a unidade individual ou módulo isoladamente. Durante os testes, são verificadas possíveis falhas ocasionadas por problemas de lógica e/ou de implementação em cada módulo, separadamente.
- Teste de Integração: consiste em analisar unidades combinadas da estrutura do *software* e a integração geral do projeto.
- Teste de Sistema: consiste em testar o *software* completo, integrado, com o intuito de verificar sua conformidade com os requisitos e a interação geral dos componentes, garantindo assim o funcionamento unânime de todos os módulos e programas sem erros.
- Teste de Aceitação: enfatiza a validação do *software* em relação aos requisitos do cliente e sua aceitabilidade.

11.2 Tipos de teste de software

Para evitar o surgimento de problemas durante o ciclo de vida de desenvolvimento de um *software*, são empregados diferentes tipos de testes, uma vez que

o objetivo principal ao desenvolver um *software* é garantir a qualidade em todas as suas camadas. Existem muitos tipos de testes, cada um com um propósito específico. São exemplos: testes de funcionalidades, testes de performance, testes de usabilidade, testes de regressão, entre outros (VALENTE, 2020a). Segue uma breve descrição dos principais tipos de teste:

- Testes funcionais: Os requisitos e regras de negócio da aplicação são testados para validar as funcionalidades documentadas do *software* (BARBOSA; TORRES, 2011).
- Teste de performance/desempenho: Pode ser definido como um processo que avalia se os requisitos de performance do sistema são atendidos, considerando um volume de acessos/transações dentro do esperado e avaliando métricas como o tempo de resposta (SOUZA; GASPAROTTO, 2013).
- Testes de usabilidade: São desempenhados pela perspectiva do usuário final, determinando se a aplicação é intuitiva e fácil para o uso do público-alvo. O objetivo desse tipo de teste é verificar a usabilidade da interface do sistema (VALENTE, 2020a).
- Testes de regressão: Consistem na execução de testes que anteriormente já foram executados com sucesso, verificando se as alterações na aplicação introduziram novos defeitos e analisando se os resultados são conforme o esperado (SOMMERVILLE, 2019). Esse tipo de teste é realizado a cada nova alteração ou atualização do sistema (BARBOSA; TORRES, 2011).

11.3 Automação de testes

A execução de testes é uma das atividades que visam contribuir para a melhoria da qualidade do *software* (BARBOSA et al., 2000). É importante em todas as fases do desenvolvimento, desde o levantamento de requisitos, definição de arquitetura, codificação e, especialmente, nos momentos de integração de componentes do sistema e validação das funcionalidades existentes. O uso de testes automatizados de *software* é considerado uma ferramenta essencial no desenvolvimento de aplicações de grande porte, principalmente para reduzir custos e tempo associados à execução exaustiva de testes (DUARTE et al., 2010). A automatização dos testes de *software* é essencial para executá-los de forma otimizada, simples e a qualquer momento (BARTIÉ, 2002).

Desta forma, automatizar testes envolve o uso de ferramentas específicas que permitem lidar com teste de forma: rápida, independente, repetível, autoavaliáveis e no tempo adequado para utilizar os teste. Este é conhecido como princípio *F.I.R.S.T.* (VALENTE, 2020a). Os testes são realizados de forma controlada, comparando os resultados esperados com os resultados reais obtidos. A qualidade do produto final está diretamente relacionada à qualidade do processo utilizado ao

longo de seu ciclo de vida, de acordo com Watts S. Humphrey, criador do CMM [2]. Portanto, a eficiência da automação de testes depende de processos consistentes e bem definidos ao longo de todo o ciclo de desenvolvimento. A automação de testes oferece maior segurança para a realização de mudanças em uma aplicação, como refatoração, manutenção ou desenvolvimento de novas funcionalidades, pois permite uma execução rápida dos testes, reduzindo esforços e possibilitando a reprodução (MOLINARI, 2008).

Com o crescimento da área de testes de *software*, é cada vez mais discutida a utilização de ferramentas que auxiliem o trabalho dos profissionais de teste, como o planejamento de casos de teste, a execução de testes e o reporte de erros e defeitos (PATUCI, 2011). Existem diversas ferramentas disponíveis para automatizar testes de *software*. Neste capítulo, serão apresentadas as ferramentas: *Selenium*[3] e *Cypress*[4].

11.4 SELENIUM

O *Selenium* é um conjunto de ferramentas amplamente utilizado para automatizar testes de forma prática e eficiente (Selenium, 2023). Criado em 2004 por Jason Huggins, foi idealizado para otimizar o processo de testes de um projeto interno na empresa ThougthWork[5]. Desenvolvida como uma biblioteca em *JavaScript* para interagir com a aplicação e reexecutar os testes em navegadores, a ferramenta começou a ser difundida na empresa. Logo, o *Selenium* recebeu destaque por seu desempenho, facilidade de uso e potencial de crescimento como uma estrutura de teste reutilizável para outros aplicativos da *Web*. O *Selenium* causou grande impacto na época, e com o crescimento na área de testes automatizados, essa biblioteca se transformou mais tarde no núcleo das ferramentas *Selenium*.

O *Selenium* permite realizar operações altamente adaptáveis, possibilitando encontrar e manipular elementos da interface, validar resultados, preencher valores e simular com precisão o comportamento de um usuário ao navegar por um sistema web. Um recente estudo sobre testes de *software* indica o *Selenium* como uma das principais ferramentas de testes atuais, seguido por *JUnit* e *Cucumber* (CERIOLI et al., 2020). Já segundo o estudo realizado por (VELOSO et al., 2010), é uma das ferramentas mais adequadas para a realização de testes funcionais automatizados. Essa popularidade pode ser justificada por algumas vantagens importantes da ferramenta, como o código-fonte aberto, a flexibilidade e o suporte a diferentes sistemas operacionais e linguagens de programação. Além disso, permite a interação com distintos navegadores, como o *Chrome*, *Firefox*, *Edge* e *Opera*, de forma automatizada.

[2] O CMM - *Capability Maturity Model*
[3] ⟨https://www.selenium.dev/pt-br/documentation/⟩
[4] ⟨https://docs.cypress.io/guides/overview/why-cypress⟩
[5] ⟨https://www.thoughtworks.com/⟩

Atualmente, o *Selenium* é um dos *frameworks* de testes mais populares. Além disso, muitas outras ferramentas se baseiam no *Selenium*, como mostram os estudos de Fowler (2018). As ferramentas que incorporam o *Selenium* são: *Selenium IDE*[6] (do inglês *Integrated Development Environment*), *Selenium RC*[7] (do inglês *Remote Control*), *Selenium WebDriver*[8] e *Selenium Grid* [9], onde cada uma possui uma abordagem distinta e se concentra em atender a necessidades específicas.

11.4.1 *Selenium IDE*

O *Selenium IDE* é uma ferramenta usada para construir *scripts* de testes automatizados por meio de um *plugin* para navegadores. A ferramenta fornece uma interface amigável e possui uma abordagem *record-and-playback*, ou seja, permite capturar as ações executadas pelo testador ou usuário em um *script*, e posteriormente reproduzir o caso de teste de forma automatizada. A ferramenta está disponível para os navegadores *Google Chrome*, *Mozilla Firefox* e *Microsoft Edge* e não requer nenhuma configuração adicional além de instalar a extensão no navegador.

O processo de criação de testes com o *Selenium IDE* pode ser exemplificado seguindo as seguintes etapas: a) Definir passos: Seguindo o caso de teste previamente especificado, o testador interage diretamente com o sistema. Para cada ação realizada no sistema, a ferramenta adiciona um passo no *script* de testes. b) Definir critérios de validação: O *Selenium IDE* fornece vários mecanismos para verificar os resultados da execução dos testes. Com base nos critérios do caso de teste em questão, o testador seleciona parâmetros e asserções para verificar se os componentes correspondem aos resultados esperados. c) Executar *script*: Por fim deve-se executar o *script* de teste para avaliar se a ferramenta capturou todos os passos do caso de teste e se as verificações correspondem aos resultados esperados. Assim, são replicadas as ações realizadas pelo testador durante a gravação do *script*. Ao final da execução, é avaliado o critério de validação, onde será informado o sucesso ou a falha do caso de teste.

[6] ⟨https://www.selenium.dev/pt-br/documentation/ide/⟩
[7] ⟨https://www.selenium.dev/pt-br/documentation/legacy/selenium_1/⟩
[8] ⟨https://www.selenium.dev/documentation/webdriver/⟩
[9] ⟨https://www.selenium.dev/documentation/grid/⟩

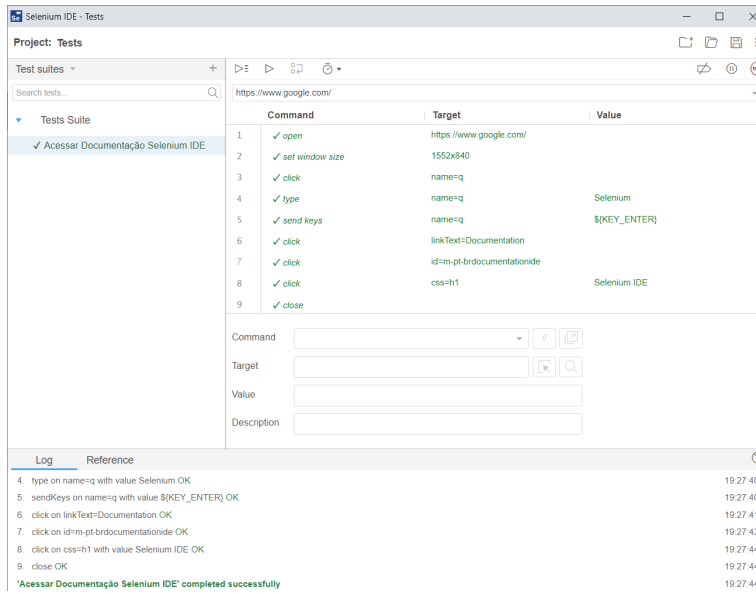


Figura 11.2: Interface do *plugin Selenium* IDE para o Chrome.

A Figura 11.2 apresenta a interface do *Selenium IDE* com um caso de teste definido. Resumidamente, o teste apresentado na Figura 11.2 consiste em abrir o navegador *Chrome*, pesquisar por *Selenium* e acessar a documentação do *Selenium IDE*. Para você reproduzir o teste demonstrado na Figura 11.2 siga os seguintes passos: Com a extensão do *Selenium IDE* instalada no seu navegador de preferência, se faz necessário, selecionar a opção "gravar um novo teste em um novo projeto", informando o nome do projeto a sua escolha e a URL desejada, no exemplo da Figura 11.2 foi utilizado: ⟨https://www.google.com/⟩. A partir desse ponto, as interações manuais podem ser automatizadas, com o *Selenium* gravando as ações e reproduzindo-as quando solicitado. Uma nova guia será aberta com a URL fornecida, para que você possa realizar uma pesquisa: pesquise por "*Selenium*"e acesse o link da documentação do *Selenium*. Ao retornar ao *Selenium IDE*, a gravação será encerrada e será solicitado um nome para o caso de teste, que poderá ser salvo e executado quando desejado. Ao executar o caso de teste, a plataforma exibe os resultados dos testes, destacando os acertos em linhas verdes e as falhas em linhas vermelhas.

Na Listagem 11.1, temos um exemplo do mesmo teste, apresentado na Figura 11.2. Porém, agora está implementado por meio de um *script* em *JavaScript*. Você pode exportar o *script* da interface do *Selenium IDE*. Para realizar essa exportação, é necessário acessar o menu de opções adicionais do caso de teste referenciado, representado por três pontos (ícone verticalmente alinhado). Ao clicar nos três pontos, um menu suspenso será exibido, e você deve procurar a opção de

exportar o caso de teste. Ao selecionar essa opção, o caso de teste será convertido em um *script* em *JavaScript*. A Listagem 11.1 apresenta um exemplo do *script* exportado. Na Linha 10 deste *script*, o *driver* é instanciado para permitir o acesso ao navegador. Nas Linhas 17 a 20, é realizada a pesquisa no Google. Em seguida, das linhas 21 a 74, ocorre o acesso à documentação do *Selenium*.

```
1  // Generated by Selenium IDE
2  const{Builder, By, Key, until} = require('selenium-webdriver')
3  const assert = require('assert')
4
5  describe('Acessar Documentação Selenium IDE', function() {
6    this.timeout(30000)
7    let driver
8    let vars
9    beforeEach(async function() {
10     driver = await new Builder().forBrowser('chrome').build()
11     vars = {}
12   })
13   afterEach(async function() {
14     await driver.quit();
15   })
16   it('Acessar Documentação Selenium IDE', async function() {
17     await driver.get("https://www.google.com/")
18     await driver.findElement(By.name("q")).click()
19     await driver.findElement(By.name("q")).sendKeys("Selenium")
20     await driver.findElement(By.name("q")).sendKeys(Key.ENTER)
21     await driver.findElement(By.linkText("Documentation"))
22     .click()
23     await driver.findElement(By.id("m-pt-brdocumentationide"))
24     .click()
25     await driver.findElement(By.css("h1"))
26     .click()
27     await driver.close()
28   })
29 })
```

Listagem 11.1: Teste exportado do Selenium IDE para JavaScript

Os *scripts* gerados são implementados com comandos *Selenium*, com parâmetros definidos conforme o contexto de cada elemento. Após a criação destes *scripts*, a ferramenta permite exportá-los para as linguagens de programação suportadas, como *Ruby*, *Java*, *Python*, *PHP* e etc. Permitindo ao testador aprimorar os *scripts* de testes por meio da programação na linguagem desejada. Desta forma, a capacidade de exportação de *script* é uma importante característica da ferramenta *Selenium IDE*, pois permite contornar possíveis limitações quanto a recursos mais avançados, como iterações e instruções condicionais. O *Selenium IDE* é considerado por muitas

literaturas como uma ferramenta de prototipagem rápida. Recomenda-se utilizar outras soluções do *Selenium* para criação de testes mais robustos que necessitem de recursos mais avançados.

11.4.2 Selenium RC

O *Selenium RC* é uma solução de testes em navegadores. Essa ferramenta foi desenvolvida com base em uma biblioteca *JavaScript* e possibilita automatizar casos de testes escritos em variadas linguagens de programação (MAHAJAN et al., 2016).

Ainda, o *Selenium RC* apresenta uma arquitetura cliente-servidor, sendo a arquitetura distribuída por dois principais componentes: o *Selenium Server* (Servidor do *Selenium RC*) e as *Client Libraries* (bibliotecas do cliente). O *Selenium Server* é responsável por inicializar e finalizar os navegadores, além de interpretar e executar os comandos *Selenium*. Além disso, atua como um *proxy* HTTP, interceptando e validando mensagens HTTP entre o navegador e o programa de teste. Já as bibliotecas do cliente fornecem uma interface entre o programa de automação de testes e o *Selenium Server* (GARCíA et al., 2020).

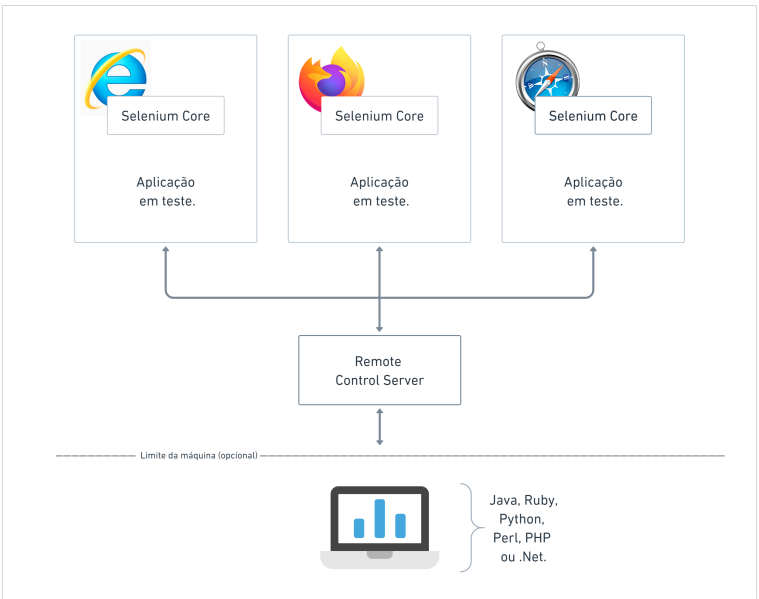


Figura 11.3: Arquitetura do Selenium RC

A Figura 11.3 ilustra a arquitetura do *Selenium RC*. As bibliotecas do cliente enviam instruções do *Selenium* ao *Selenium Server*, que, por sua vez, repassa os comandos ao *Selenium Core*. O *Selenium Core* é uma biblioteca de comandos

em *JavaScript* responsável por executar comandos no navegador, permitindo a automatização das ações na aplicação web testada.

Embora o *Selenium RC* tenha sido um dos principais projetos do *Selenium* por um tempo e tenha feito contribuições significativas no campo da automação de testes em navegadores, a ferramenta apresenta limitações significativas. Tais como: algumas funcionalidades não eram suportadas nas interações automatizadas da web, como *upload* e *download* de arquivos, *pop-ups* e o manuseio de diálogos. Além disso, uma considerável sobrecarga do servidor *Selenium RC* afeta o desempenho final dos testes. Diante disso, em 2008, Simon Stewart da empresa Thinkworks, desenvolveu uma nova ferramenta para automatização de testes com recursos mais poderosos, chamada *Selenium WebDriver* (GARCíA et al., 2020).

11.4.3 Selenium WebDriver

O *Selenium WebDriver* pode ser definido como uma coleção de APIs usadas para automatizar testes em uma aplicação da web. A ferramenta oferece suporte à automação dos principais navegadores do mercado. A comunicação da ferramenta é bidirecional: o *WebDriver* passa os comandos ao navegador por meio do *driver* e recebe informações pela mesma rota (GONZALEZ et al., 2022).

Como mencionado anteriormente, o *Selenium WebDriver* surgiu como solução para as limitações do *Selenium RC*. O *Selenium WebDriver*, também conhecido como *Selenium* 2.0, fornece uma interface de programação mais simples e concisa. Diferentemente do *Selenium RC*, que exige que um servidor específico opere durante a utilização da ferramenta, o *Selenium WebDriver* interage diretamente com o navegador para a realização de testes de uma aplicação Web (WARDHAN; MADAN, 2021).

O *Selenium WebDriver* é uma versão mais avançada do *Selenium RC*, que, além de abordar as desvantagens do *Selenium RC*, também adiciona novos recursos. A ferramenta possui interface baseada em Ajax e é ainda mais rápida do que o *Selenium RC*, pois faz chamadas diretas aos navegadores, utilizando o suporte nativo de cada navegador para automação (TIWARI et al., 2021).

O *Selenium WebDriver* é uma ferramenta muito útil para navegação avançada de usuários. Suporta de forma eficiente as páginas dinâmicas da web, nas quais os elementos de uma página podem variar sem que a própria página seja recarregada (ANGMO; SHARMA, 2014; WARDHAN; MADAN, 2021). Logo, ao utilizar a ferramenta, é possível navegar por páginas mais complexas e realizar testes automatizados.

A ferramenta também suporta a execução em modo *Headless*, destinado a navegadores baseados em Firefox e Chromium. Com esse recurso, é possível executar *scripts* automatizados no modo sem cabeçalho, ou seja, durante a execução, a janela do navegador não ficará visível. Isso permite que o *Selenium WebDriver* seja utilizado para executar testes automatizados em ambientes de linha de comando, além de facilitar sua integração em ambientes de integração contínua."

O *Selenium WebDriver* é considerado o mais rápido entre os componentes do *Selenium*. Um dos fatores que contribuem para isso é a ausência do envolvimento do *Selenium Server*. Os comandos fornecidos no código são traduzidos em métodos de serviço da web, e o *driver* remoto aceita solicitações HTTP e as executa diretamente no navegador. Isso resulta em ganhos relacionados à eficiência e à otimização no tempo de execução dos testes (RAMYA et al., 2017).

11.4.4 Selenium Grid

O *Selenium Grid* é uma ferramenta que permite a execução de testes em paralelo em várias máquinas remotas. Ele suporta diferentes tipos e versões de navegadores e de sistemas operacionais. Também permite executar os *scripts* de testes em várias instâncias do mesmo navegador. Essa paralelização no *Selenium Grid* pode ser um fator importante na otimização do tempo de execução, principalmente quando há uma grande quantidade de casos de teste para serem realizados.

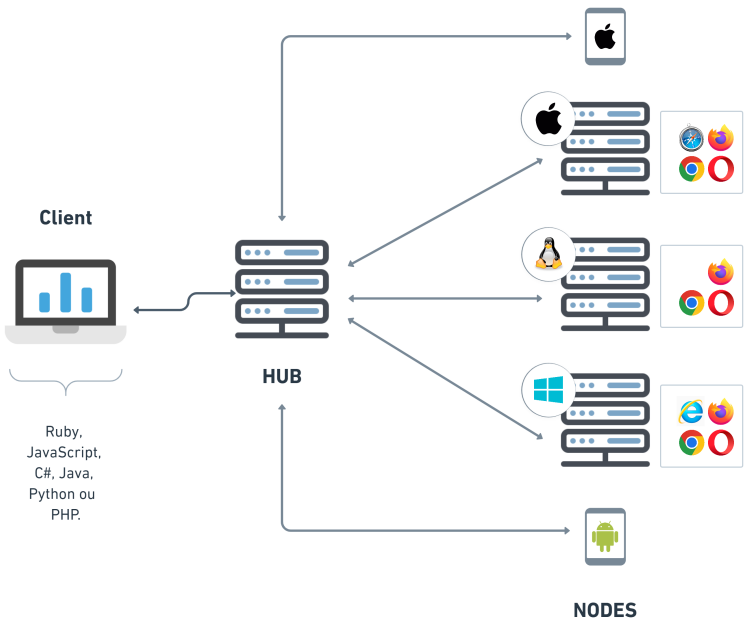


Figura 11.4: Arquitetura do Selenium Grid

A arquitetura do Selenium Grid pode ser exemplificada em dois principais componentes: um Hub ou um servidor e um nó ou um dispositivo remoto. O Selenium Hub é um servidor que rastreia os nós e as solicitações realizadas pelos *scripts* do Selenium. Conforme ilustrado na Figura 11.3, o hub recebe instruções do cliente WebDriver, e a execução paralela do teste é realizada em uma máquina

diferente chamada de nó (GARCíA et al., 2020). É possível configurar diferentes nós em diferentes navegadores e/ou sistemas operacionais.

11.4.5 Exemplo prático com *Selenium*

O *Selenium* oferece, por meio do *WebDriver*, suporte à automação de testes nos principais navegadores do mercado, como Chrome/Chromium, Firefox, Internet Explorer, Edge e Safari. Para esse exemplo prático, iremos considerar a utilização do *Selenium WebDriver* no navegador Google Chrome no Windows 10. O exemplo será implementado em *JavaScript*.

11.4.6 Preparação do ambiente

Para preparar o ambiente para a utilização do *Selenium WebDriver*, será necessário instalar a biblioteca *Selenium* para a linguagem de programação escolhida, neste caso, *JavaScript*. A instalação será realizada pelo terminal, por meio do npm, conforme demonstrado na Listagem 11.2.

```
npm install selenium-webdriver
```

Listagem 11.2: Instalar Selenium WebDriver com npm

Logo após, será necessário instalar os *drivers* do navegador. Na documentação oficial da ferramenta podemos encontrar diferentes formas de realizar esse processo. Para essa explicação inicial, usaremos a localização do *driver* definida no código. Realizamos o download manual do *driver* do navegador escolhido, o Google Chrome. Em seguida, especificamos a localização do *driver* no próprio código, como demonstrado na Listagem 11.3. Desta forma, dispensamos a necessidade de configurar variáveis de ambiente no sistema. Contudo, isso acaba tornando o código menos flexível, mas para este exemplo com fins didáticos, não levaremos isso em consideração.

```
const {Builder} = require('selenium-webdriver');
const chrome = require('selenium-webdriver/chrome');
const service = new chrome.ServiceBuilder(
'/path/to/chromedriver');
const driver = new Builder().forBrowser('chrome').
setChromeService(service).build();
```

Listagem 11.3: Configuração de drivers de navegadores definida no código em *Selenium*

11.4.7 Explicação do Exemplo

Para ilustrar o uso do *Selenium*, iremos utilizar um site de demonstração de automação de testes de interface do usuário, o SauceDemo[10]. Uma página web que simula um *e-commerce*: nas Figuras 11.5 e 11.6 podemos observar as telas de login e de exibição de produtos, respectivamente. A solução fornece algumas opções de usuários para a realização dos testes: o *standard_user*, *locked_out_user*, *problem_user* e *performance_glitch_user*. Usaremos as credenciais de acesso do *standard_user*, que é um usuário comum e funcional para simular compras no site.

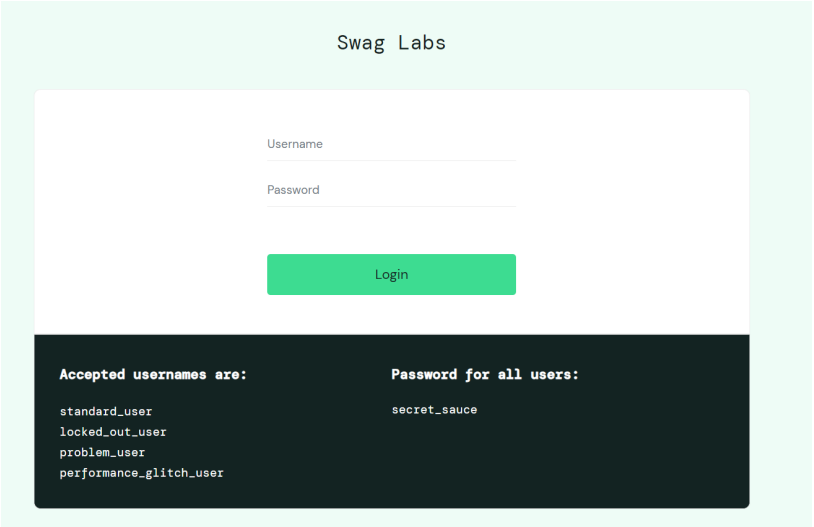


Figura 11.5: Interface da tela de login da aplicação SauceDemo

[10] ⟨https://www.saucedemo.com/⟩

Tabela 11.2: Especificação do caso de teste para o exemplo prático

Item de teste	Realização de um login padrão.
Especificações de entrada	**Usuário:** *standard_user* **Senha:** *secret_sauce* Clicar no botão "Login"
Procedimentos	1. Acesse a página web https://www.saucedemo.com/ 2. Insira o dado de entrada "Usuário" no campo Nome de usuário. 3. Insira o dado de entrada "Senha" no campo Senha. 4. Localizar e clicar no botão de "Login".
Especificações de saída	Após a realização dos procedimentos sentidos, a tela da aplicação deve ser alterada para a exibição inicial da aplicação (contendo os produtos do e-commerce).

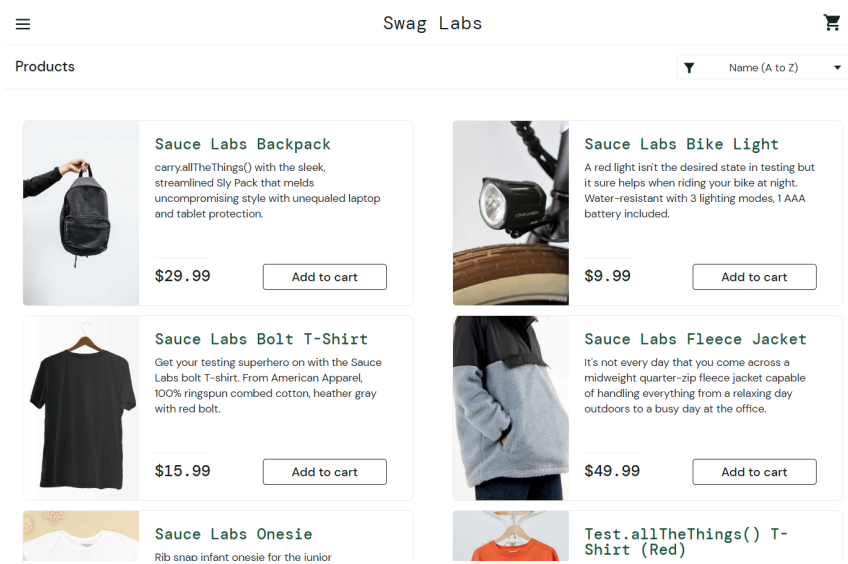


Figura 11.6: Interface da tela de produtos da aplicação SauceDemo

O exemplo prático será desenvolvido a partir de um fluxo básico de um login padrão. Portanto, o caso de teste consiste em acessar a página de login da aplicação (Figura 11.5), inserir os dados do usuário *standard_user* e verificar o sucesso do login ao acessar a tela inicial com os produtos (Figura 11.6). A Tabela 11.2 apresenta a especificação completa do caso de teste.

11.4.8 Implementação do Exemplo

Após configurar o ambiente, como mencionado anteriormente na Seção 11.4.6, precisamos criar um novo arquivo em *JavaScript*. Neste arquivo, serão adicionados os comandos para a realização do teste automatizado. Antes de organizar a estrutura do teste, vamos discutir os principais comandos utilizados neste exemplo.

Primeiramente, é necessário iniciar uma sessão para abrir o navegador escolhido, neste caso, o Google Chrome. Para isto, usamos o comando apresentado na Listagem 11.4. Logo após, iremos navegar até a página web da aplicação *SauceDemo*. Utilizamos o *driver.get*() para acessar a URL da aplicação. conforme demonstrado na Listagem 11.5.

```
1 driver = await new Builder().forBrowser('chrome').build();
```

Listagem 11.4: Instanciar o navegador Google Chrome no Selenium WebDriver

```
1 await driver.get("https://www.saucedemo.com/")
```

Listagem 11.5: Abrir página Web no Selenium WebDriver

Para adicionar os dados de acesso do usuário, precisamos localizar, na página, os campos de entrada correspondentes. Utilizamos a função *driver.findElement*() para encontrar elementos no DOM [11]. Os parâmetros usados podem ser *ids*, *classes*, características de estilo e outros métodos para identificar os elementos da página web com os quais desejamos trabalhar. Assim, para os campos **Username** e **Password**, e para o próprio botão de Login, usaremos os comandos apresentados na Listagem 11.6.

```
1 let userBox=await driver.findElement(By.id("user-name"));
2 let passBox=await driver.findElement(By.id("password"));
3 let loginButton=await driver.findElement(By.id("login-button"));
```

Listagem 11.6: Localizar elementos na página Web com *Selenium*

Agora que já encontramos e definimos os elementos principais, teremos que interagir para a realização dos procedimentos do caso de teste. Existem alguns comandos usados para realizar essa interação, contudo, usaremos principalmente o *.click*() e *.sendKeys*() para essa automatização. Usamos o comando *.click*() para clicar nos elementos de uma página web e o *.sendKeys*() para inserir dados fornecidos em elementos editáveis. Com os elementos já definidos nas variáveis *userBox* e *passBox*, é necessário clicar e informar os dados do usuário nos campos correspondentes. Em seguida, devemos clicar no botão definido na variável *loginButton*. Esse processo pode ser observado na Listagem 11.7.

[11] DOM: *Document Object Model*, é uma interface de programação que os navegadores utilizam para representar páginas na web.

```
await userBox.click();
await userBox.sendKeys("standard_user");
await passBox.click();
await passBox.sendKeys("secret_sauce");
await loginButton.click();
```

Listagem 11.7: Inserindo dados em campos de entradas em *Selenium*

Por fim, para realizar a verificação das especificações de saída, temos que obter informações da página web. Neste exemplo utilizaremos o *.getText*() e *.getCurrentUrl*(). O comando *.getText*() obtém da tela um texto visível de um elemento. Já o comando *.getCurrentUrl*(), obtém uma *string* que representa a URL atual que o navegador está visualizando. Portanto, para verificar a mudança de tela, observaremos a URL atual e a presença do título "Products"na tela (Listagem 11.8).

```
let productsTitle = await driver. findElement((By.className(
"title"))).getText();
assert.equal("Products", productsTitle);
let urlPage = await driver.getCurrentUrl();
assert. equal("https://www.saucedemo.com/inventory.html",
urlPage);
```

Listagem 11.8: Verificação de *asserts* no *Selenium*

Na Listagem 11.10 é apresentado o *script* de teste automatizado em `Selenium WebDriver`, completo e reorganizado. O resultado após a execução do caso de teste é ilustrado na Figura 11.7.

```
require("chromedriver");
const {By, Builder, Browser} = require('selenium-webdriver');
const {suite} = require('selenium-webdriver/testing');
const assert = require("assert");
suite(function (env) {
  describe("Login", function() {
    let driver;
    before(async function() {
      driver = await new Builder().forBrowser('chrome').build();
    });
    after(async () => await driver.quit());
    it('Login Padrão.', async function() {
      // 1. Acessar a página Web https://www.saucedemo.com/
      await driver.get("https://www.saucedemo.com/");

      let userBox=await driver.findElement(By.id("user-name"));
      let passBox=await driver.findElement(By.id("password"));
      let loginButton = await driver.findElement(By.id(
      "login-button"));

      // 2. Inserir o dado de entrada Usuário no campo Username.
      await userBox.click();
      await userBox.sendKeys("standard_user");

      // 3. Inserir o dado de entrada Senha no campo Password.
      await passBox.click();
      await passBox.sendKeys("secret_sauce");

      // 4. Localizar e clicar no botão de Login.
      await loginButton.click();

      // Resultados: A tela da aplicação deverá ser alterada
      // para a exibição inicial da aplicação
      // (contendo os produtos do e-commerce).
      let productsTitle = await driver.findElement((
      By.className("title"))).getText();
      assert.equal("Products", productsTitle);

      let urlPage = await driver.getCurrentUrl();
      assert.equal("https://www.saucedemo.com/inventory.html",
      urlPage);
    })
  })
},{ browsers: [Browser.CHROME, Browser.FIREFOX]});
```

Listagem 11.9: Código completo do exemplo no *Selenium*

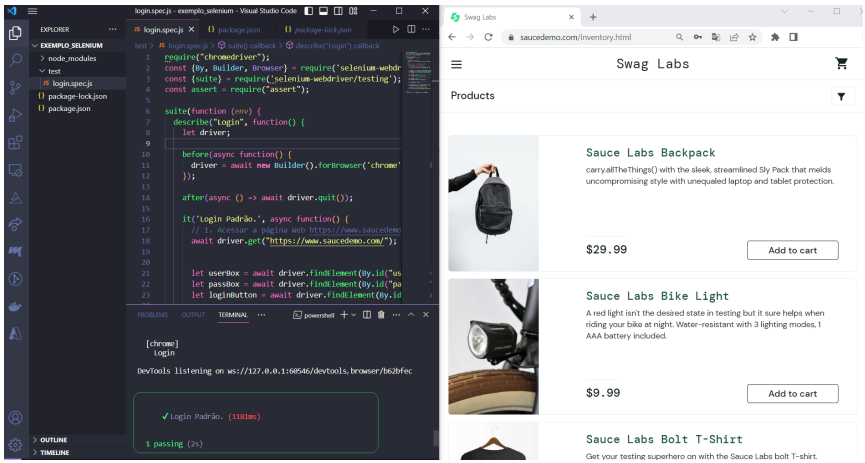


Figura 11.7: Resultado do caso de teste com Selenium

11.5 CYPRESS

O *Cypress*[12] é um *framework* de testes automatizados de aplicações web *end-to-end*[13], desenvolvido por Brian Mann. É uma ferramenta relativamente nova no cenário de testes automatizados quando comparada ao Selenium. Segundo Machado (2017) a primeira versão foi liberada em 2015, mas o lançamento oficial do Cypress foi realizado somente em 2017. De acordo com (TAKY, 2021), a ferramenta foi projetada para auxiliar o trabalho de desenvolvedores e de engenheiros de garantia de qualidade.

O *Cypress* é construído em Node.js e é empacotado como um módulo npm. Utiliza JavaScript para o desenvolvimento dos *scripts* de testes e é um projeto de código aberto (*open source*). Além disso, a ferramenta herda muitos dos métodos *jQuery* para a identificação de componentes de interface do usuário e simplifica a navegação e a manipulação da árvore do DOM HTML, bem como a manipulação de eventos, animação CSS e Ajax. O *Cypress* funciona como um aplicativo autônomo e pode ser instalado nos sistemas operacionais macOS, Unix/Linux e Windows, tanto por meio de aplicativos quanto de ferramentas de linha de comando (MWAURA, 2021).

Diferentemente da maioria das ferramentas de testes para aplicações web, que operam interagindo com o navegador e executam comandos remotos via rede, o *Cypress* opera diretamente no navegador. O navegador executa o código de teste e permite que o *Cypress* ouça e modifique o comportamento do navegador em tempo de execução, manipulando o DOM e alterando as solicitações e respostas da rede

[12] ⟨https://docs.cypress.io/guides/overview/why-cypress⟩
[13] ⟨https://circleci.com/blog/what-is-end-to-end-testing/⟩

em tempo real (KHETARPAL, 2021). A Figura 11.8 exemplifica a arquitetura de funcionamento da ferramenta.

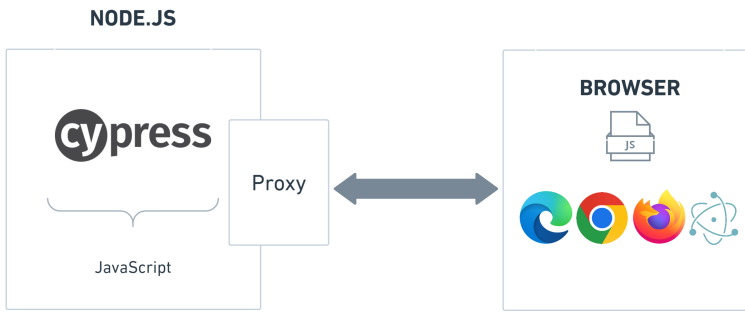


Figura 11.8: Arquitetura do Cypress

Os navegadores que o Cypress suporta são: Chrome, Electron (padrão), Chromium, Mozilla Firefox (suporte beta) e o Microsoft Edge (baseado em Chromium). Segundo a documentação oficial da ferramenta, também podem ser adicionados alguns *plugins* via npm, o que estende a funcionalidade e acrescenta novos comandos e possibilidades de personalização. Além disso, existem alguns outros recursos disponíveis em versões pagas, como, por exemplo, integrações com ferramentas de gerenciamento de projetos e algumas outras funcionalidades.

O *Cypress* também permite a paralelização dos testes, distribuindo automaticamente os testes para otimizar o tempo de execução. Apresenta uma interface amigável com um *dashboard* onde é possível verificar as execuções dos testes. Com a ferramenta é possível realizar diferentes tipos de testes, como, por exemplo, testes unitários, funcionais, de integração e de ponta a ponta.

11.5.1 Exemplo prático com Cypress

O *Cypress* é uma ferramenta desktop compatível com alguns sistemas operacionais, como por exemplo: macOS, Linux Ubuntu, Fedora, Debian e Windows. Para esse exemplo prático, iremos considerar a utilização do *Cypress* no navegador Google Chrome no Windows 10.

11.5.2 Preparação do ambiente

Podemos instalar o *Cypress* com o npm. Além de facilitar a preparação do ambiente, também possibilita a manutenção e atualização da ferramenta. Caso o testador ou desenvolvedor opte por não utilizar o NodeJs, ou npm, a ferramenta pode ser instalada baixando-a por meio do download direto. Na documentação

oficial da ferramenta, podemos baixar o *Cypress* por meio de um link direto. Desta forma, a versão mais recente disponível da ferramenta é baixada. O site detecta a plataforma que realizou a solicitação de download automaticamente, então não é necessário escolher a versão segundo seu ambiente. Após finalizar o download, descompactar e instalar a ferramenta, o *Cypress* estará pronto para uso. Neste exemplo, utilizaremos o npm para preparar o ambiente, como demonstrado na Listagem 11.10.

```
1 npm install cypress --save-dev
```

Listagem 11.10: Comando de instalação do *Cypress*

Após concluir a instalação, além dos arquivos e pastas do *Cypress* , é gerado automaticamente um arquivo chamado package-lock.json. Este arquivo controla as possíveis alterações que o npm aplica à árvore node_modules ou ao package.json. Descrevendo a árvore exata que foi gerada, e retornando árvores idênticas mesmo após instalações subsequentes. Com o *Cypress* instalado, para abrir a ferramenta a partir da raiz do projeto, é necessário apenas executar, no terminal, o comando 11.11.

```
1 npx cypress open
```

Listagem 11.11: Comando de execução do *Cypress*

11.5.3 Explicação do Exemplo

Para ilustrar a utilização do *Cypress*, iremos fazer o mesmo exemplo realizado com o Selenium na Seção 11.4.5. No site de demonstração, SauceDemo, utilizaremos as credenciais do perfil *standard_user*. Portanto, serão realizados o acesso à página da aplicação, o preenchimento dos dados do usuário e a verificação do sucesso do login. As especificações do caso de teste permanecem as mesmas da Tabela 11.2.

11.5.4 Implementação do Exemplo

Após instalar o framework com o npm, como mencionado anteriormente, precisamos criar o projeto e o *script* de testes em JavaScript. Para isto, executamos o *Cypress* utilizando o comando da Listagem 11.11 em uma interface de linha de comando. Logo após, a ferramenta exibe algumas opções de configuração para a elaboração dos *scripts* de testes (Figura 11.9). Inicialmente, será definido o tipo de teste a ser realizado; neste exemplo, usaremos a opção E2E Testing. O teste E2E, ou *end-to-end*, é usado para testar uma aplicação desde o navegador web até o *back-end*, além de integrá-la a outros serviços. Segundo a documentação oficial da ferramenta, neste tipo de teste, o *Cypress* reproduz fielmente o comportamento de usuários ao interagirem diretamente com a aplicação. Utilizaremos, portanto, esta opção, pois o objetivo deste exemplo é simular um login de usuário.

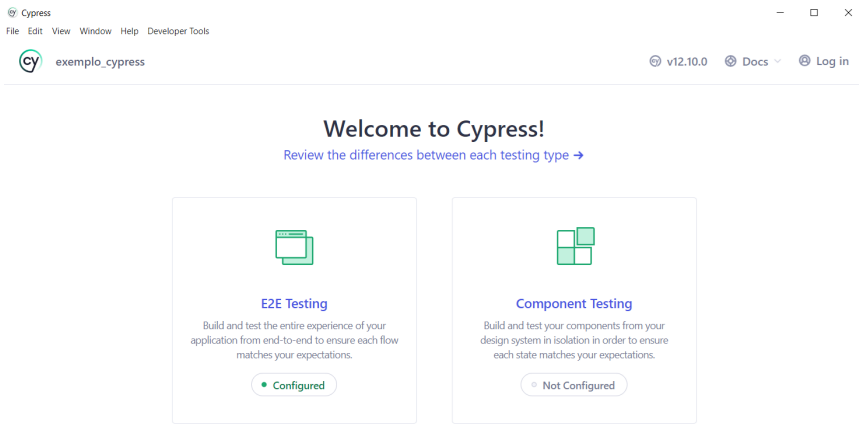


Figura 11.9: Interface da tela inicial do Cypress

Posteriormente, a ferramenta elenca um conjunto de arquivos para a configuração prévia do teste do tipo escolhido; nesta etapa, basta prosseguir (Figura 11.10). Em seguida, são apresentadas as opções de navegadores. Neste teste, utilizaremos o Google Chrome. Após selecionar o navegador, será necessário criar uma nova especificação vazia. Neste arquivo, iremos configurar os passos do teste. Assim, temos uma estrutura inicial de pastas e arquivos necessários para o desenvolvimento e execução dos testes. A Figura 11.11 ilustra a estrutura inicialmente gerada pelo *framework*.

Assim, com a estrutura do projeto inicialmente gerada pelo Cypress, iremos adicionar os comandos para o desenvolvimento do *script* de teste no arquivo *spec.cy.js*. Antes de organizar a estrutura do teste, vamos discutir os principais comandos utilizados neste exemplo.

Para abrir a página web da aplicação em teste, utilizamos o comando *cy.visit()* usado para acessar URLs no navegador, como demonstrado na Listagem 11.12. É importante notar, que diferente de outras ferramentas, não foi necessário realizar nenhuma configuração para a ferramenta fazer uso do navegador.

```
1 cy.visit("https://www.saucedemo.com/")
```

Listagem 11.12: Abrir página Web usando *Cypress*

Os comandos *.get()* e *.type()*, são utilizados para localizar e inserir os dados de acesso nos campos de entrada de **Username** e **Password**. Na Listagem 11.13 podemos observar como encontramos e atribuímos os dados nos campos que desejamos. Com o comando *.get()*, podemos especificar características dos elementos da página web que desejamos trabalhar; estes parâmetros podem ser *ids*, *classes*,

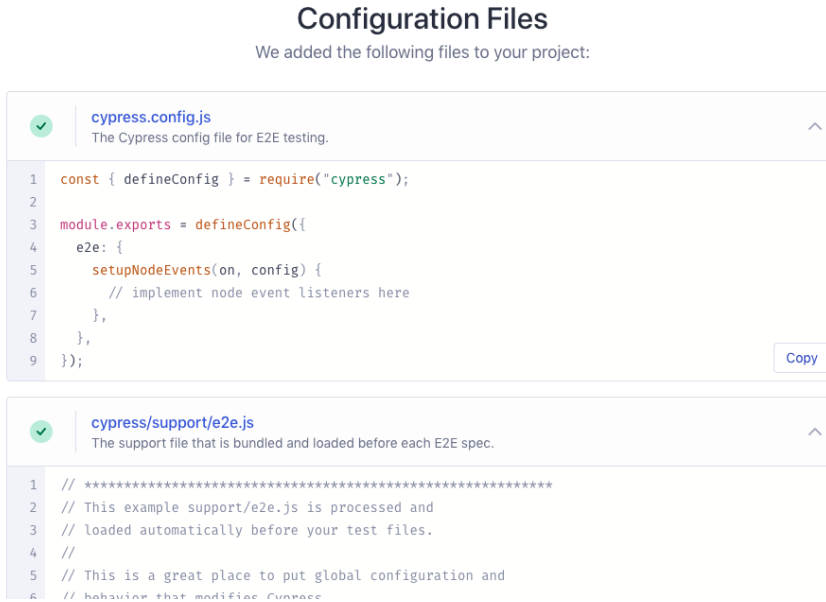


Figura 11.10: Interface da tela de configuração do Cypress

características de estilo e outros métodos de identificação de elementos DOM. Por meio do comando *.type*() podemos inserir os dados de texto a serem digitados no elemento.

```
cy.get('#user-name').type("standard_user")
cy.get('#password').type("secret_sauce")
```

Listagem 11.13: Localizar e preencher os campos de entrada no *Cypress*

Em seguida, para localizar o botão de Login na tela, utilizamos o mesmo comando *.get*() com argumentos referentes ao botão. Por fim, utilizamos o *.click*() para dar o *input* e clicar no botão para solicitar a realização do login na página.

```
cy.get('#login-button').click()
```

Listagem 11.14: Clicar em elemento DOM usando *Cypress*

Para a validação das especificações de saída, utilizamos o comando *.should*() para criar as asserções. Assim, verificamos a mudança de tela comparando se a URL atual *.url*().*should*() corresponde ao endereço da página que deveria aparecer. Realizamos isto, atribuindo *'eq'* (de *equals*) e a *string* contendo a URL como parâmetro. Também é verificada a presença do elemento com o título **Products**



Figura 11.11: Estrutura de arquivos do projeto em Cypress

na tela atual da página, usando o atributo de visibilidade *.should('be.visible')*. Na Listagem 11.15 podemos verificar esses passos do teste.

```
1 cy.url().should('eq','https://www.saucedemo.com/inventory.html')
2 cy.get('span:contains(Products)').should('be.visible')
```

Listagem 11.15: Verificação dos *asserts* em *Cypress*

Assim, teremos o *script* de teste automatizado em Cypress completo e reorganizado, como pode ser observado na Listagem 11.16. O resultado após a execução do *script* pode ser visto na Figura 11.10.

```
/// <reference types="cypress" />
describe("Login", () => {
  it("Login Padrão.", function () {
    // 1. Acessar a página Web https://www.saucedemo.com/
    cy.visit("https://www.saucedemo.com/")

    // 2. Inserir o dado de entrada Usuário no campo Username.
    cy.get('#user-name').type("standard_user")

    // 3. Inserir o dado de entrada Senha no campo Password.
    cy.get('#password').type("secret_sauce")

    // 4. Localizar e clicar no botão de Login.
    cy.get('#login-button').click()

    // Resultados: A tela da aplicação deverá ser alterada
    //para a exibição inicial da aplicação (contendo os produtos
    //do e-commerce).
    cy.url().should('eq', 'https://www.saucedemo.com/
    inventory.html')
    cy.get('span:contains(Products)').should('be.visible')

  });
});
```

Listagem 11.16: Código completo do exemplo em *Cypress*

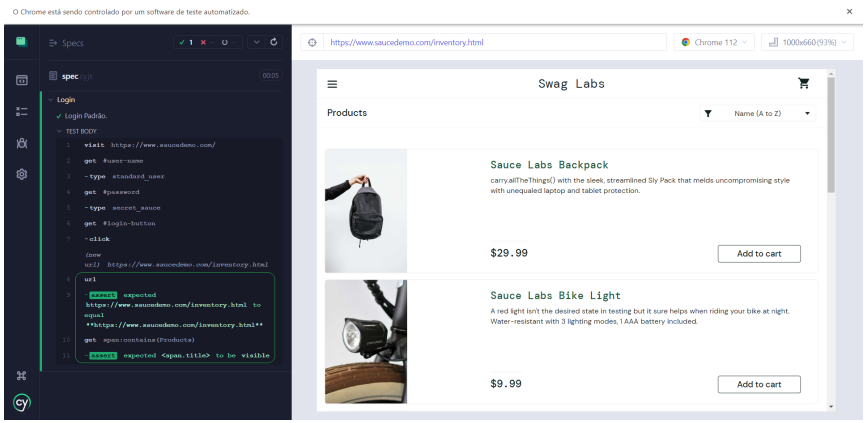


Figura 11.12: Resultado do caso de teste com Cypress

11.6 Exemplo prático de uso do *Selenium* e *Cypress*

Para demonstração prática de uma construção de suíte de teste utilizando *Selenium* e *Cypress* foi selecionada uma aplicação web para a realização dos testes automatizados em ambas as ferramentas. Logo, a mesma assume o papel de sistema em teste, do inglês SUT[14]. Para selecionar o SUT foram realizadas algumas pesquisas de aplicações disponíveis para estudo de testes, levando em consideração as seguintes condições: a) deve ser uma aplicação web sob licença de código aberto; b) deve simular um sistema real comumente usado, para facilitar a compreensão das funcionalidades; c) deve ser uma aplicação de configuração e acesso simplificado para facilitar a automatização e replicação dos experimentos. Com base nesses requisitos, o SUT selecionado para aplicação prática dos testes automatizados foi o *ParaBank*.

O *ParaBank* é uma aplicação web que simula um site de banco online, desenvolvida pela empresa *Parasoft*[15] para demonstração de suas soluções de *software*. O *ParaBank* é um projeto de código aberto, implementado em Java, no lado do servidor; entre suas principais funcionalidades, podemos elencar o gerenciamento de clientes, transações, pagamentos e outros serviços bancários. Na Figura 11.13 podemos observar a tela inicial do sistema, que dispõe de menus laterais para navegação, bem como de campos para realizar o login e o cadastro de clientes.

Assim, com a definição e escolha do SUT, foi realizado um levantamento dos aspectos funcionais da aplicação. O levantamento de requisitos é uma etapa significativa, em que se utilizam todas as informações disponíveis do sistema para gerar os requisitos e, com isso, identificar as principais funções daquele sistema (WAZLAWICK, 2011). A aplicação *Parabank* possui muitas funcionalidades e possibilidades para simular um sistema bancário, entretanto, para o escopo deste trabalho, foram consideradas apenas algumas dessas funcionalidades para a realização dos testes.

11.6.1 Modelagem dos casos de teste

Antes de executar qualquer tipo de teste em um *software*, é necessário analisar e elencar os possíveis passos que deverão ser executados, assim como, os resultados que deverão ser obtidos de acordo com os requisitos da aplicação em teste. A documentação destas informações consiste na criação de um caso de teste. Os casos de teste são especificações das entradas e das saídas esperadas do sistema, além de uma declaração do que está sendo testado (SOMMERVILLE, 2019).

Para a criação dos casos de teste, foi proposta uma abordagem de especificação baseada em casos de uso. Um caso de uso é a descrição de uma sequência de

[14] *System under test*, refere-se a um sistema que está sendo testado ou validado.

[15] *Parasoft* é uma empresa de tecnologia da informação, que ajuda as organizações a fornecer continuamente *software* de qualidade com seu conjunto integrado e comprovado de mercado de ferramentas automatizadas de teste de *software* (⟨https://www.parasoft.com/⟩).



Figura 11.13: Tela inicial da aplicação web *Parabank*

ações que um usuário pode realizar para interagir com o sistema, visando alcançar um determinado resultado esperado (LEFFINGWELL, 2010). Ainda, de acordo com o RUP, *Rational Unified Process (RUP)* é uma abordagem de engenharia de *software* para delegar atividades e responsabilidades em uma organização de desenvolvimento. Seu objetivo principal é permitir a criação de *software* de alta qualidade que atenda aos requisitos do usuário final.Um caso de uso é uma sequência de ações executadas para gerar um resultado observável em um sistema. Assim, os casos de teste derivados de casos de uso podem garantir a conformidade da aplicação em relação aos seus requisitos funcionais (KOSINDRDECHA; DAENGDEJ, 2010).

O SUT escolhido neste estudo apresenta uma quantidade significativa de funcionalidades; portanto, pode haver um grande número de possíveis casos de uso. No entanto, o escopo deste trabalho se limita a apenas algumas dessas funcionalidades. A Figura 11.14 ilustra as etapas adotadas para definir a modelagem dos casos de teste.

Em trabalhos anteriores Heumann (2001), é proposto um processo de três etapas para gerar casos de teste a partir de casos de uso: (a) para cada caso de uso,

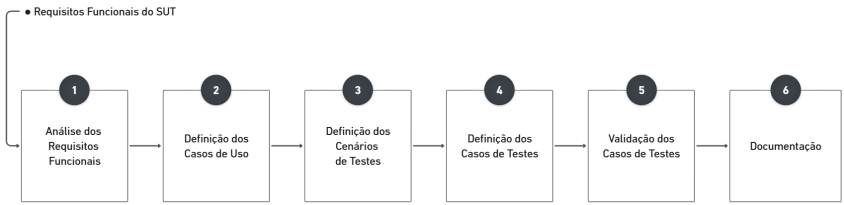


Figura 11.14: Modelagem dos casos de teste

é definido um conjunto de cenários; (b) para cada cenário, é identificado pelo menos um caso de teste; e (c) para cada caso de teste, são determinados os valores dos dados com os quais testar. Logo, as etapas realizadas neste estudo e ilustradas na Figura 11.14 as etapas escolhidas para criação do conjunto de testes:

i. Análise dos Requisitos Funcionais: Inicialmente, foi realizada uma análise dos requisitos definidos no SUT, obtidos na Seção 11.6, para definir a abrangência e a priorização das funcionalidades e dos requisitos deste capítulo.

ii. Definição dos Casos de Uso: Foram identificados, de acordo com os requisitos, os casos de uso que definem as ações e as interações com o SUT para atender às funcionalidades elencadas nos testes. Para cada caso de uso foram definidos os dados de título, descrição, precondições e pós-condições.

iii. Definição dos Cenários: Para cada caso de uso foi definido um conjunto de cenários, identificados como diferentes combinações de fluxos básicos e alternativos que podem ser executados nas funcionalidades do SUT.

iv. Definição dos Casos de Teste: Com os cenários obtidos, definiram-se os casos de teste. Cada caso de teste representa uma especificação dos fluxos a serem percorridos; para estes, foram determinados um identificador, um título, um conjunto de entradas, as condições de execução (ou passos) e os resultados esperados.

v. Validação dos Casos de Teste: Para avaliar a viabilidade de implementar os casos de teste criados, cada um foi executado manualmente no SUT. Realizando, portanto, os fluxos definidos detalhadamente, considerando os dados de entrada, os passos e os resultados esperados.

vi. Documentação: Por fim, foram documentados todos os casos de uso, cenários e casos de teste definidos após a realização das etapas anteriores.

11.6.2 Implementação e execução dos testes automatizados

Durante esta etapa foram desenvolvidos *scripts* de testes utilizando as duas ferramentas em estudo, *Selenium* e *Cypress*. Para dar equidade ao estudo empírico, para os testes em ambas as ferramentas, foram utilizados o mesmo ambiente de produção, aplicação web, requisitos e casos de teste, além da linguagem *JavaScript* com a *IDE VSCode*[16]. Como resultado, foi desenvolvido suítes de testes automatizados para as duas ferramentas, devidamente documentados e armazenados no GitHub[17] (Apêndice C).

Após a finalização da etapa anterior (Seção 11.6.2), foram executadas as suítes de testes automatizados em ambas as ferramentas em estudo. Como resultado, foi gerado um relatório de execução que exibe o *status* de sucesso ou de falha para cada caso de teste. Assim, além de gerar o relatório, a execução automatizada dos testes em cada uma das ferramentas forneceu os dados necessários para embasar o estudo empírico. As métricas de avaliação utilizadas são detalhadas na próxima etapa.

11.6.3 Características das ferramentas alvo do estudo

Adotando uma metodologia baseada na análise documental das ferramentas em estudo, consideraram-se algumas características importantes para as ferramentas de automação de testes. Assim, são elencados as características e critérios escolhidos para o exemplo prático de uso das ferramentas:

i. Sistemas Operacionais suportados: Conhecer em qual SO uma ferramenta de automação de testes web pode ser utilizada é muito importante. Sabendo que atualmente há uma grande variedade de sistemas operacionais e distribuições, é crucial que estas ferramentas possam executar-se em diferentes ambientes. A confiabilidade de um sistema depende do contexto em que é usado (SOMMERVILLE, 2019); portanto, é necessário avaliar o sistema em contextos o mais diversos possível. Logo, neste estudo, são considerados os suportes do *Selenium* e *Cypress* para os principais sistemas operacionais da atualidade: Microsoft Windows, Apple macOS, Linux, Android e iOS (WGU, 2021).

ii. Linguagens de Programação suportadas: Uma linguagem de programação é um método padronizado com um conjunto de regras sintáticas e semânticas, usadas para expressar instruções de um programa a um computador (GOTARDO, 2015). A escolha de qual linguagem utilizar pode estar vinculada a diferentes variáveis, como, por exemplo, a curva de aprendizagem, o tempo de experiência ou até mesmo a preferência. Neste contexto, para o uso e a escrita de *scripts* de testes, é importante avaliar o suporte das linguagens disponíveis em uma ferramenta de automação de testes. Assim, quanto maior a variedade de linguagens de programação a que uma ferramenta dá suporte, mais fácil

[16] *Visual Studio Code* é um editor de código redefinido e otimizado para criar e depurar aplicativos modernos da web e da nuvem (⟨https://code.visualstudio.com/⟩)

[17] ⟨https://github.com/fischerJF/manual-web-dev⟩

pode se tornar o processo de aprendizado e utilização dessa ferramenta, uma vez que os testadores podem utilizar a linguagem com a qual possuem mais familiaridade. Neste trabalho, considerou-se, para efeitos de comparação, o suporte a algumas das linguagens mais usadas: JavaScript, Python, Java, PHP e Ruby (iMasters, 2023).

iii. Navegadores suportados: Aplicações web executam diretamente no navegador; é crucial avaliar as possibilidades em diversos contextos (SOMMERVILLE, 2019). Logo, para uma ferramenta de automação de testes, é importante oferecer a possibilidade de executar testes em diferentes navegadores. Assim, neste estudo consideramos o suporte das ferramentas aos diferentes navegadores do mercado: Google Chrome, Mozilla Firefox, Microsoft Edge, Opera e Safari (SANTOS, 2023).

iv. Instalação e configuração: Algumas ferramentas podem exigir dependências de outras tecnologias para funcionar adequadamente. Configurações complexas e mal documentadas podem ser um fator limitante para algumas ferramentas. Neste contexto, foram avaliadas as dependências das ferramentas em relação a outros recursos ou tecnologias, como, por exemplo, o uso de *drivers* ou bibliotecas.

v. Recursos disponíveis: Foram avaliados também alguns recursos disponíveis ao utilizar as ferramentas, como por exemplo, a possibilidade de usar múltiplas instâncias do navegador durante os testes, execução de testes em paralelo, execução remota, *Time Travel*[18] e espera automática[18]. Também foram avaliados aspectos da geração de relatórios e do suporte ao CI/CD[19].

11.6.4 Criação da suíte de teste de diferenças entre o *Selenium* e *Cypress*

Os resultados do estudo das características das ferramentas *Selenium* e *Cypress* são apresentados nos tópicos a seguir. São consideradas informações comprovadas e obtidas por meio de documentações oficiais e de estudos bibliográficos de cada ferramenta.

11.6.5 Sistemas Operacionais suportados

Em relação ao suporte à utilização e à execução em diferentes sistemas operacionais, ambas as ferramentas apresentam resultados satisfatórios, cobrindo a

[18] *Time Travel*, em português "viagem no tempo", aplicado ao contexto de testes possibilita salvar e retornar aos estados anteriores da aplicação durante a execução dos testes.

[18] A espera automática está relacionada ao carregamento de elementos em uma aplicação web, como um *timeout* configurado pela própria ferramenta em tempo de execução.

[19] CI/CD, abreviação de *Continuous Integration/Continuous Delivery*, trata-se de uma prática de desenvolvimento de *software* que visa tornar a integração de código mais eficiente por meio de *builds* e testes automatizados.

Tabela 11.3: Suporte das ferramentas Selenium e Cypress aos principais SOs

Ferramenta de Testes	Sistemas Operacionais				
	Microsoft Windows	Apple MacOS	Linux	Android	iOS
Selenium	✓	✓	✓	✓	✓
Cypress	✓	✓	✓	✗	✗

Tabela 11.4: Suporte das ferramentas Selenium e Cypress a diferentes linguagens de programação

Ferramenta de Testes	Linguagem de Programação				
	JavaScript	Python	Java	PHP	Ruby
Selenium	✓	✓	✓	✓	✓
Cypress	✓	✗	✗	✗	✗

maioria dos sistemas operacionais mais populares, como mostra a Tabela 11.3. Neste aspecto, as ferramentas demonstram diferença apenas em relação aos sistemas operacionais para aplicações móveis, Android e iOS.

Segundo a sua documentação oficial, o *Cypress* é uma ferramenta de teste criada para a web moderna (CYPRESS, 2023). Portanto, não oferece suporte à execução de testes em dispositivos móveis. Para realizar este tipo de teste, é preciso que o aplicativo possa ser executado em um navegador de computador pessoal, e não em um dispositivo móvel. Já o *Selenium*, oferece suporte para realizar testes em aparelhos móveis (Selenium, 2023). Para isso, a ferramenta utiliza *frameworks* específicos para automatizar os testes, como, por exemplo, o Appium[20]. O Appium deriva suas raízes do *Selenium* e utiliza a mesma API do WebDriver para controlar navegadores móveis no Android e no iOS. Diante disso, o *Selenium* oferece maior suporte a diferentes SOs, incluindo projetos para dispositivos móveis.

11.6.6 Linguagens de Programação suportadas

A Tabela 11.4 apresenta uma comparação do suporte a diferentes linguagens de programação para a escrita de testes nas ferramentas *Selenium* e *Cypress*. Neste aspecto, as ferramentas em estudo apresentam uma grande diferença. O *Selenium* oferece mais liberdade na escolha da linguagem, com suporte a uma ampla variedade de linguagens de programação para a criação de testes automatizados. Já o *Cypress* se limita ao uso do JavaScript, que é uma linguagem amplamente utilizada em aplicações web. O *Cypress* foi criado para lidar especialmente com estruturas *JavaScript*, prometendo maior controle na automação ao manipular tudo o que está sendo executado no navegador (CYPRESS, 2023).

[20] O Appium é um *framework* projetado para facilitar a automação da interface do usuário de muitas plataformas, incluindo dispositivos móveis (⟨https://appium.io/⟩).

Tabela 11.5: Suporte das ferramentas Selenium e Cypress a diferentes navegadores

Ferramentas de Testes	Navegadores				
	Google Chrome	**Mozilla Firefox**	**Microsoft Edge**	**Opera**	**Safari**
Selenium	✓	✓	✓	✓	✓
Cypress	✓	✓	✓	✗	✗

11.6.7 Navegadores suportados

A Tabela 11.5 apresenta um comparativo do suporte aos diferentes navegadores das ferramentas *Selenium* e *Cypress*. É verificado, portanto, que o *Selenium* pode ser utilizado em uma maior variedade de navegadores em comparação ao *Cypress*. Isto pode ser justificado principalmente pelo favorecimento que a arquitetura do *Selenium* proporciona, uma vez que sua execução em diferentes navegadores pode ser realizada apenas com o uso dos *drivers* correspondentes a cada navegador. O *Cypress*, além dos navegadores listados na Tabela 11.5, também oferece suporte a outros navegadores, como o Electron, que já vem embutido na ferramenta e não precisa ser instalado separadamente.

11.6.8 Instalação e configuração

O grau de complexidade de instalação e configuração pode ser relativo; portanto, as ferramentas em estudo foram avaliadas quanto à dependência de outras tecnologias para compor o projeto de automação de testes. A Tabela 11.6 apresenta os resultados obtidos a partir da documentação de ambas as ferramentas. O *Cypress* pode ser instalado sem necessidade de nenhuma dependência ou tecnologia adicional, por meio do *download* direto da versão mais recente disponível, bastando descompactar e instalar. Também pode ser instalado com o *npm* ou o *yarn*[21] com um único comando. Já para instalar e utilizar o *Selenium*, são necessárias outras tecnologias para a configuração. O *Selenium Webdriver* requer o *download* de *drivers* de navegadores e a configuração prévia do ambiente de teste, assim como o *Selenium Grid*. Entretanto, para o *Selenium IDE*, basta baixar a extensão e instalar. Logo, neste contexto, há maior complexidade na instalação e na configuração do *Selenium* em relação ao *Cypress*. Os dados da coluna de configuração prévia foram baseados nas dependências e no envolvimento de outros conceitos nos tutoriais de instalação da documentação das ferramentas.

11.6.9 Recursos disponíveis

A Tabela 11.7 apresenta os resultados da comparação entre as ferramentas *Selenium* e *Cypress* quanto a alguns recursos disponíveis. As ferramentas compartilham alguns recursos, como a geração de relatórios de testes, a execução de testes

[21] O Yarn é um dos principais gerenciadores de pacotes JavaScript (⟨https://yarnpkg.com/⟩).

Tabela 11.6: Instalação e configuração das ferramentas Selenium e Cypress

Ferramenta de Testes		**Dependência de bibliotecas, drivers ou outras tecnologias**	**Configuração prévia complexa do ambiente**
Selenium	**Grid**	SIM	SIM
	IDE	NÃO	NÃO
	Webdriver	SIM	SIM
Cypress		NÃO	NÃO

Tabela 11.7: Recursos disponíveis das ferramentas Selenium e Cypress

Ferramentas de Testes	**Interface da Execução dos Testes**	**Espera Automática**	**CI/ CD**	**Execução Remota**	**Teste Paralelo**	**Múltiplas instâncias do navegador**	**Time Travel**	**Relatório de testes**
Selenium	✗	✗	✓	✓	✓	✓	✗	✓
Cypress	✓	✓	✓	✗	✓	✗	✓	✓

em paralelo e o uso em ambientes de integração e entrega contínua (CI/CD). O *Cypress* apresenta um diferencial em relação à disponibilidade de uma interface para acompanhar a execução dos testes, como também a possibilidade de espera automática e de salvar e retornar aos estados anteriores da aplicação durante a execução dos testes (conceito de *Time Travel*). Já o *Selenium*, permite a execução remota de testes e a utilização de múltiplas instâncias de um navegador e múltiplas abas.

Existem inúmeros outros recursos ofertados pelas ferramentas *Selenium* e *Cypress*, bem como formas de configurar alguns recursos que não são disponibilizados por padrão. Deste modo, a comparação entre as ferramentas torna-se mais subjetiva, considerando que cada projeto de testes tem contextos e objetivos distintos. Portanto, cabe ao testador realizar o levantamento do que será necessário para a automação dos testes e optar pela ferramenta que ofereça os recursos desejados. No Apêndice A, apresenta-se uma tabela que resume os resultados teóricos discutidos nesta seção, bem como outros aspectos e características das ferramentas em estudo.

11.6.10 Breve comparação entre *Selenium* e *Cypress*

Nesta seção são apresentadas algumas comparações entre as ferramentas, *Selenium* e *Cypress*, baseadas na implementação de testes automatizados em uma aplicação web selecionada como *System Under Test* (*SUT*). Seguindo as etapas previamente mencionadas na Seção 11.6. Logo, foi realizado todo o processo de instalação e configuração das ferramentas, o desenvolvimento dos *scripts*, a execução dos testes e a observação dos resultados e dos relatórios.

11.6.11 Sistema em teste (SUT)

O sistema *Parabank* foi selecionado como SUT neste trabalho para a modelagem e execução dos testes automatizados. O *Parabank* é uma aplicação web gratuita e de código aberto; o seu projeto está disponível para uso tanto em um domínio[22] na web quanto em um repositório remoto[23]. Como mencionado anteriormente, a aplicação oferece uma série de funcionalidades de um sistema bancário online, desde o cadastro de clientes até a realização de transações e pagamentos. Contudo, para delimitar o escopo e abrangência dos testes implementados foram selecionadas algumas telas e menus do sistema, são eles: login de usuários, encontrar informações de login, cadastrar usuários (menus 1, 2 e 3, respectivamente, na Figura 11.15), criar uma conta, visualizar painel geral das contas, atualizar dados e sair do sistema (menus 4, 5, 6 e 7, respectivamente, na Figura 11.16).



Figura 11.15: Menus da tela inicial da aplicação Web Parabank

Logo, com o SUT definido, determinamos os principais requisitos funcionais abordados e cobertos na implementação dos testes automatizados. Na Tabela 11.8 são elencados e descritos os requisitos que limitam o escopo dos testes realizados neste estudo. Os requisitos foram baseados nas funcionalidades selecionadas para a implementação dos testes.

Com os requisitos de funcionamento bem definidos, foram elaborados os casos de uso que determinam o comportamento básico da aplicação para atender a estes

[22]Domínio web do Parabank:⟨https://parabank.parasoft.com/parabank/index.htm⟩
[23]Repositório Github do Parabank:⟨https://github.com/parasoft/parabank⟩

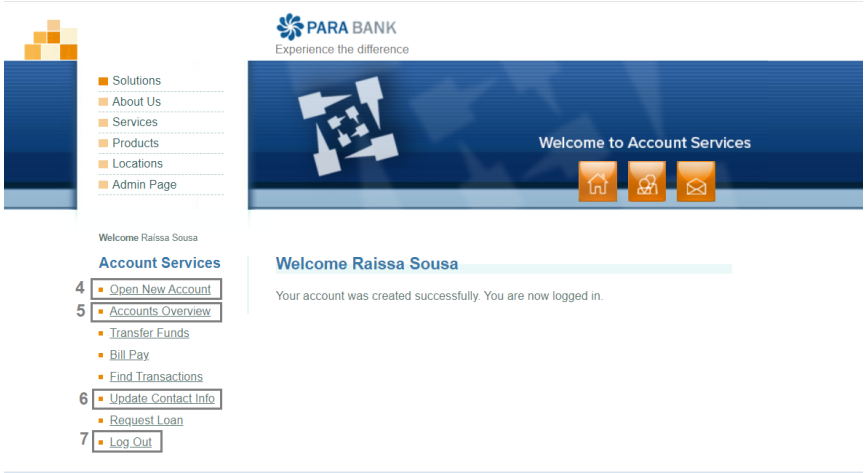


Figura 11.16: Menus da tela principal da aplicação Web Parabank

Tabela 11.8: Requisitos funcionais do SUT

Identificador	Descrição
RF01	Permitir o cadastro de novos clientes.
RF02	Assegurar que todos os dados do cadastro de novos clientes são válidos e completos.
RF03	Permitir a validação do acesso de clientes cadastrados.
RF04	Identificar problemas com a validação do acesso de clientes cadastrados.
RF05	Permitir recuperar dados de acesso de clientes cadastrados.
RF06	Garantir a segurança dos dados dos clientes ao tentar recuperar informações de acesso.
RF07	Permitir atualizar os dados cadastrais de clientes.
RF08	Permitir a criação de contas bancárias para clientes cadastrados.
RF09	Permitir ao cliente acessar as informações de movimentação em sua(s) conta(s).
RF10	Permitir ao cliente realizar movimentações em sua(s) conta(s).

requisitos; estes são elencados na Tabela 11.9. Em seguida, para cada caso de uso, definiram-se cenários de teste. A Tabela 11.10 exibe todos os casos de teste que foram documentados previamente para orientar a implementação e execução dos testes automatizados no SUT.

Para cada caso de teste foram definidos os objetivos, os dados necessários, os passos e resultados esperados de acordo com a utilização do SUT. Por exemplo, o caso de teste **TC01 - Cadastro de usuário com sucesso** que realiza o cadastro de um cliente na aplicação Parabank é exibido na Tabela 11.11. Após a criação e especificação de todos os casos de teste de acordo com o exemplificado na Tabela 11.11, os testes foram implementados e executados de forma automatizada utilizando as ferramentas *Selenium* e *Cypress*. No Apêndice B deste trabalho é apresentada uma tabela com todos os casos de testes realizados neste estudo, detalhando as especificação de cada um deles.

Tabela 11.9: Casos de Uso do SUT

Identificador	Título	Objetivo
UC01	Cadastro de Usuários	Cadastro de clientes na aplicação ao fornecer dados válidos e completos.
UC02	Login de Usuários	Autenticação de clientes cadastrados na aplicação.
UC03	Encontrar informações de login	Recuperação dos dados de acesso de clientes cadastrados ao fornecer dados válidos.
UC04	Atualizar dados	Atualização dos dados de clientes cadastrados.
UC05	Criar uma conta	Criar contas bancárias para clientes cadastrados.
UC06	Visualizar painel geral das contas	Monitorar dados e informações de movimentações da(s) conta(s) de clientes cadastrados.
UC07	Sair do sistema	Encerrar a sessão de clientes cadastrados.

Tabela 11.10: Casos de Testes do SUT

Casos de Uso	Requisitos Funcionais	Casos de Testes
UC01	RF01 e RF02	TC01 - Cadastro de usuário com sucesso
		TC02 - Cadastro de usuário com todos os campos vazios
		TC03 - Cadastro de usuário com um username já cadastrado
		TC04 - Cadastro de usuário com campos não obrigatórios vazios
		TC05 - Cadastro de usuário com senhas diferentes
		TC06 - Cadastro de usuário com campos obrigatórios vazios
UC02	RF03 e RF04	TC07 - Login com sucesso
		TC08 - Login com todos os campos vazios
		TC09 - Login com usuário não cadastrado
		TC10 - Login com senha errada
		TC11 - Login com campo de usuário vazio
		TC12 - Login com campo de senha vazio
UC03	RF05 e RF06	TC13 - Encontrar informações de login com sucesso
		TC14 - Encontrar informações de login com todos os campos vazios
		TC15 - Encontrar informações de login com dados inconsistentes
		TC16 - Encontrar informações de login com usuário não cadastrado
		TC17 - Encontrar informações de login com campos obrigatórios vazios
UC04	RF07	TC18 - Atualizar dados com sucesso
		TC19 - Atualizar dados com todos os campos vazios
		TC20 - Atualizar dados com campos obrigatórios vazios
		TC21 - Atualizar dados com o campo telefone (não obrigatório) vazio
UC05	RF08	TC22 - Criar uma conta com sucesso
		TC23 - Visualizar detalhes de uma nova conta
UC06	RF09	TC24 - Visualizar painel geral das contas com sucesso
		TC25 - Exibir lista de contas com sucesso
		TC26 - Visualizar detalhes de uma conta com sucesso
		TC27 - Verificar dados da conta
		TC28 - Visualizar atividades da conta
		TC29 - Visualizar lista de transações da conta
UC07	RF10	TC30 - Realizar logout

Tabela 11.11: Especificação do caso de teste TC01

TC	Descrição	Dados do Teste	Passos	Resultado Esperado
TC01	Verifica se o usuário consegue realizar o cadastro com sucesso na aplicação após fornecer todos os dados válidos.	- Acesso à aplicação ParaBank. - Dados para cadastro de clientes: Nome, Endereço (Rua, Estado, Zipe Code), Telefone e SSN.	1. Abrir a aplicação ParaBank.	1. Exibir página inicial do ParaBank.
			2. Verificar o menu Register.	2. O menu Register deve ser exibido.
			3. Clicar em Register.	3. A aplicação exibe a tela de cadastro de clientes.
			4. Verificar a exibição da tela de cadastro com os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm.	4. Os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm devem ser exibidos.
			5. Inserir dados válidos nos campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm.	5. Os dados devem ser inseridos nos campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm.
			6. Clicar no botão Register.	6. O login deve ser feito e a aplicação exibe na tela a mensagem: Your account was created successfully. You are now logged in.

Tabela 11.12: Especificações do ambiente e máquina de desenvolvimento e execução dos testes

Máquina	**Modelo**	Dell Inc. G3 3500
	Processador	Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz (12 CPUs), ~2.6GHz
	Memória	16,0 GB
	Placa de vídeo	NVIDIA GeForce GTX 1660 Ti
	Sistema operacional	Windows 10 Pro 64-bit
Outros programas	**Navegador**	Google Chrome, Versão 114.0.5735.134 ~64 bits
	IDE	Visual Studio Code 1.79.2

11.6.12 Instalação e configuração

Para instalar e configurar cada uma das ferramentas em estudo, foi necessário seguir alguns passos no ambiente de desenvolvimento utilizado para a implementação e execução dos testes. Para tornar a comparação prática deste estudo mais precisa, foram realizados esforços para que todos os processos fossem o mais semelhantes possível entre ambas as ferramentas. Logo, foram utilizados os mesmos requisitos iniciais para a instalação: o Node.js e o editor de código *Visual Studio Code*. Bem como as mesmas especificações de ambiente e de máquina de desenvolvimento. Os detalhes são apresentados na Tabela 11.12.

Como o *Cypress* não tem requisitos adicionais para uma instalação padrão, não foi necessário o uso de bibliotecas, mecanismos de teste ou *drivers*. Com um único comando de instalação, a ferramenta já estava pronta para executar e gerar relatórios dos testes. Já com o *Selenium*, foi necessário utilizar outras tecnologias para a execução dos testes. Mesmo realizando uma configuração mais simples com o uso da estrutura de testes Mocha[24] para instalar os módulos necessários, foi preciso

[24]Mocha é uma estrutura de teste JavaScript executada em Node.js (⟨https://mochajs.org/⟩).

recorrer a outras tecnologias, como, por exemplo, o *driver* do navegador. Destarte, podemos verificar que o *Cypress* apresentou, durante este estudo empírico, uma configuração e instalação mais simples e mais rápidas.

11.6.13 Desenvolvimento dos *scripts*

Ambas as ferramentas, *Selenium* e *Cypress*, apresentam seus próprios comandos e sintaxes para criar e executar testes automatizados. De acordo com as funcionalidades e os casos de uso definidos anteriormente, foi desenvolvido um *script* para cada caso. Assim, com o uso de ferramentas, foram criados sete *scripts* de testes. Em cada um destes arquivos foram implementados os casos de teste correspondentes aos cenários de cada caso de uso.

As métricas avaliadas no processo de desenvolvimento dos *scripts* deste trabalho baseiam-se na análise do código, e não na experiência de uso. Assim, como especificado anteriormente, as métricas de *software* analisadas são: a) *Source Lines of Code* (SLOC); b) *Comment Lines of Code* (CLOC); c)*Physical Lines of Code* (PLOC); d) *Blank Lines of Code* (BLANK)); e) contagem de caracteres totais do código. Para calcular as linhas de código dos *scripts* em ambas as ferramentas, foi utilizada a extensão *VS Code Counter* para o VSCode, que oferece suporte à contagem de linhas de código-fonte, em branco, comentadas e totais[22], que realiza a contagem das linhas de código-fonte, em branco, comentadas e totais. Os resultados são apresentados na Tabela 11.13.

A contagem de linhas de código é uma métrica simples e confiável, porém, determinar como realizar comparações válidas pode ser um problema (LAIRD; BRENNAN, 2006). Diante disso, considerou-se uma análise com base na quantidade de linhas de código necessárias para que as ferramentas executassem o mesmo conjunto de casos de teste. É importante evidenciar que todos os testes foram implementados com o mesmo número de passos e *assertions*[23].

A Tabela 11.13 mostra que, para ambas as ferramentas, os *scripts* apresentam valores semelhantes de CLOC e BLANK, indicando que as linhas de comentário e em branco foram equiparadas para facilitar a análise do código. As métricas CLOC e BLANK geralmente são usadas para analisar a quantidade de documentação no código e os espaçamentos; são sensíveis à formatação logicamente irrelevante e a convenções de estilo. Portanto, são mais subjetivas e não necessariamente evidenciam uma conclusão acerca do código. Os valores das métricas SLOC e PLOC apresentam um contraste maior. A métrica PLOC evidencia a quantidade bruta de linhas de um código, contendo linhas em branco e comentadas. Já a métrica SLOC apresenta um comparativo interessante sobre os *scripts*. Os resultados mostram que, para todos os *scripts* dos UCs, o *Selenium* apresenta o maior número de linhas de código. Destarte, o estudo evidencia que, para realizar o mesmo conjunto de

[22] ⟨https://marketplace.visualstudio.com/items?itemName=uctakeoff.vscode-counter⟩

[23] *Assertion* é um conceito-chave em testes, as asserções são usadas para declarar o comportamento esperado ou o resultado de um teste

Tabela 11.13: Métricas do desenvolvimento dos *scripts* de testes em Selenium e Cypress

SLOC								
Ferramenta de Teste	**UC01**	**UC02**	**UC03**	**UC04**	**UC05**	**UC06**	**UC07**	**Total**
Selenium	148	78	106	118	79	179	31	739
Cypress	137	65	96	89	52	143	21	603
CLOC								
Ferramenta de Teste	**UC01**	**UC02**	**UC03**	**UC04**	**UC05**	**UC06**	**UC07**	**Total**
Selenium	7	8	6	9	7	12	3	52
Cypress	7	7	6	5	6	11	3	45
BLANK								
Ferramenta de Teste	**UC01**	**UC02**	**UC03**	**UC04**	**UC05**	**UC06**	**UC07**	**Total**
Selenium	3	1	2	8	6	14	2	36
Cypress	4	2	2	6	6	13	2	35
PLOC								
Ferramenta de Teste	**UC01**	**UC02**	**UC03**	**UC04**	**UC05**	**UC06**	**UC07**	**Total**
Selenium	158	87	114	135	92	205	36	827
Cypress	148	74	104	100	64	167	26	683
Quantidade de caracteres								
Ferramenta de Teste	**UC01**	**UC02**	**UC03**	**UC04**	**UC05**	**UC06**	**UC07**	**Total**
Selenium	12103	5377	8680	9125	5733	13070	1742	55830
Cypress	6938	3327	7879	5226	2791	8029	1048	35238

testes elaborado neste trabalho, o *Selenium* requer mais linhas de instrução do que o *Cypress*.

Na Tabela 11.13 também são exibidos os resultados das quantidades de caracteres em cada *script* de teste. Para obter esses dados, foram desconsideradas as linhas em branco e comentadas. Analisando os valores individuais e totais, é possível perceber que o *Selenium* possui comandos mais verbosos[27] para realizar os mesmos testes, em comparação ao *Cypress*. Em Toomim et al. (2004), destaca-se que redundância e verbosidade podem dificultar a compreensão de um código e obscurecer informações significativas. Por outro lado, comandos pouco descritivos também podem causar essa dificuldade. No estudo empírico realizado, foi observado que com o *Selenium* era necessário utilizar instruções mais longas do que no *Cypress* para realizar a mesma ação, por exemplo, nas Listagens 11.17 e 11.18 são exibidos os comando para verificar a exibição da mensagem *"Welcome username"* na página.

```
1 assert.equal("Welcome "+ USERNAME,
2 await driver.findElement((By.className("title")))
3 .getText())
```

Listagem 11.17: Comando no Selenium para verificar a exibição de um elemento

[27] Verbosidade: qualidade do que é verboso, que fala muito ou usa palavras em excesso para se expressar.

Tabela 11.14: Tempo de execução dos testes em Selenium e Cypress

UC	Tempo de Execução Médio (s)	
	Selenium	Cypress
UC01	20	25
UC02	12	13
UC03	19	18
UC04	24	22
UC05	17	15
UC06	27	28
UC07	10	5
Total	129	126

```
1 cy.get(".title").contains("Welcome "+ USERNAME)
```

Listagem 11.18: Comando no Cypress para verificar a exibição de um elemento

11.6.14 Execução dos testes

O tempo total de execução de testes automatizados é uma métrica bem relativa; contudo, pode auxiliar no acompanhamento do progresso geral da execução dos testes. Como os valores podem variar de acordo com o equipamento em que é executado, foi utilizada a mesma máquina e ambiente para execução dos *scripts* de teste, as especificações estão descritas na Seção 11.6.12. Existem muitos fatores que podem influenciar no resultado da medição do tempo, como por exemplo a interferência de processos executados na máquina ou até a latência da rede, tentando minimizar a ocorrência destes desvios a execução dos testes em cada uma das ferramentas foi realizada cem vezes. A Tabela 11.14 mostra o tempo médio de execução para cada *script* de teste utilizando o *Selenium* e o *Cypress*.

Logo, descobriu-se que o tempo total de execução para este conjunto de testes não apresentou uma diferença significativa entre o *Selenium* e o *Cypress*. Com base na Tabela 11.14, as diferenças encontradas foram de apenas $3s$ na média no tempo total de execução. Todavia, como a aplicação em teste não está em um ambiente totalmente controlado, o tempo de execução deste estudo pode não refletir de fato o desempenho real das ferramentas.

11.6.15 Relatórios de testes

Ao fim de cada ciclo de teste é escrito um relatório, que consiste na descrição de como foi a execução do teste, apontando os resultados e quais os possíveis problemas que ocorreram (MENDEZ, 2011). Tanto o *Selenium* quanto o *Cypress* exibem, ao

final da execução dos testes, um relatório no próprio terminal da IDE do VSCode. Esses resultados, gerados por ambas as ferramentas, também fornecem informações úteis para o rastreio de falhas no código. As Figuras 11.17 e 11.18 mostram, respectivamente, exemplos desses relatórios de testes gerados com o *Selenium* e o *Cypress*.

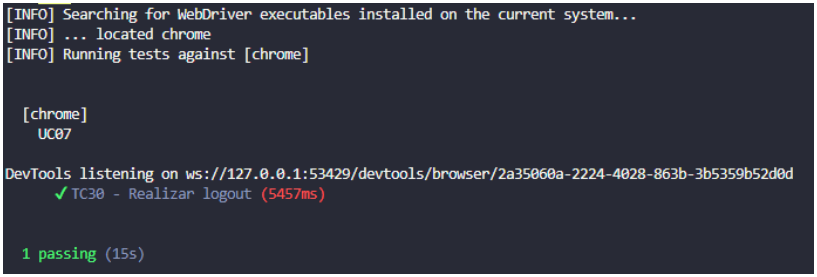


Figura 11.17: Relatório de testes via terminal utilizando o Selenium

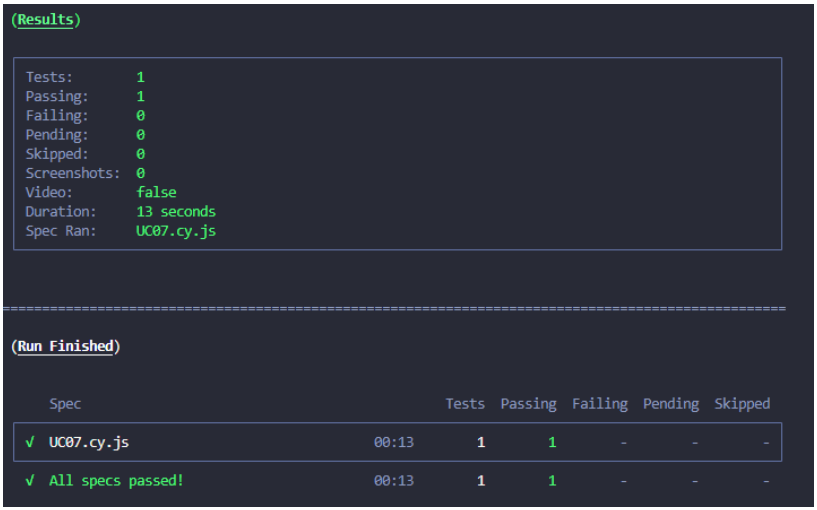


Figura 11.18: Relatório de testes via terminal utilizando o Cypress

Ambas as ferramentas também oferecem suporte aos modelos de relatórios mais visuais, como, por exemplo, o *Mochawesome*[24] que consiste em um modelo de relatório personalizado para uso em estrutura de teste Javascript. Logo, avaliando da perspectiva da geração de relatórios as ferramentas em estudo se mostram bastante eficientes, uma vez que, permitem obter os resultados, rastrear falhas e personalizar a visualização e estilos do relatório.

[24] ⟨https://www.npmjs.com/package/mochawesome⟩

11.7 Lições aprendidas

Esta seção discute algumas características das ferramentas *Selenium* e *Cypress*, com base nas lições aprendidas durante a realização deste estudo. Com base nos resultados, foram elencados alguns tópicos para apresentar e descrever as lições. É importante mencionar que, em cada tópico, são abordadas as perspectivas dos pontos fortes e fracos de cada ferramenta, considerando os resultados deste estudo. Logo, isso não significa necessariamente que uma ferramenta seja melhor que outra.

I. **Sintaxe para desenvolvimento dos *scripts*:** As ferramentas em estudo apresentam características diferentes quanto a sintaxe adotada. Enquanto o *Cypress* utiliza instruções de código curtas e objetivas, no *Selenium* os comandos são mais extensos. De fato, existem estudos que associam a verbosidade à menor legibilidade de um código de *software*, destacando que pode obscurecer informações significativas no código, tornando-o difícil de entender. Por outro lado, comandos curtos e sem objetividade também podem prejudicar a compreensão de um *script*. Nessa perspectiva, embora o *Selenium* detenha *scripts* mais verbosos do que o *Cypress*, foi verificado, em ambos, que os códigos produzidos são legíveis e objetivos. Por exemplo, os comandos `cy.contains(id).should('be.visible')` e `driver.findElement(By.id())`.isDisplayed(), usados respectivamente no *Selenium* e *Cypress*, deixam clara a ação executada de procurar um elemento pelo *id* e verificar se é exibido na página.

II. **Limitações ao testar aplicações Web dinâmicas:** Durante o estudo, foram observadas algumas limitações ao utilizar ambas as ferramentas para automatizar os testes no SUT escolhido, uma aplicação web dinâmica. Com o *Selenium*, notou-se maior dificuldade no manuseio de elementos dinâmicos, como, por exemplo, menus *dropdown*. Além disso, a ferramenta apresentou limitações na limpeza dos elementos de *input*. Por outro lado, ao utilizar o *Cypress*, observou-se uma lentidão ao trabalhar com formulários, o que pode ser atribuído aos conceitos de espera automática da ferramenta. Como também, existe a dificuldade em lidar com a manipulação de várias guias e a solução para isto no *Cypress* pode ser uma tarefa complicada, contudo não foi preciso para os testes com o SUT deste trabalho.

III. **Configurações de espera para garantir o sucesso:** Durante a execução dos testes, o *Cypress* aguarda automaticamente o carregamento do DOM. Característica muito interessante para a aplicação de testes otimizados, sem a necessidade de implementar ou configurar esperas explícitas ou implícitas. Já com o *Selenium*, às vezes é necessário fazer este tipo de declaração de espera para carregar os elementos e garantir que o teste não falhe, o que muitas vezes aumenta o tempo de desenvolvimento.

IV. **Replicabilidade em ambientes diferentes:** O *Selenium* oferece maior flexibilidade na escolha de ambientes, com suporte a linguagens, sistemas

operacionais e navegadores variados. Já o *Cypress* se limita às definições de linguagem e do ambiente de execução. Contudo, quando se trata de replicação de testes em ambientes diferentes, o *Cypress* apresenta uma vantagem significativa. Ao tentar reproduzir a execução dos testes no SUT em diferentes navegadores, constatou-se que o uso dos *scripts* do *Selenium* exigiu grande esforço, incluindo diversas modificações e configurações. Em contrapartida, com o *Cypress*, foi possível executar em todos os navegadores suportados e listados na interface da ferramenta com apenas um clique, além de permitir a definição do parâmetro *–browser* diretamente na linha de comando.

Capítulo 12

Visão geral sobre DevOps

Ana Cristina Pousa Machado (UFOP)

Márcio Bruno Loiola Gomes (UFC)

Marco Antônio Alves de Almeida (UFOP)

Igor Muzetti Pereira (UFOP)

Fischer Ferreira (UNIFEI)

12.1 O que é DevOps?

Nos últimos anos, o termo DevOps tem ganhado destaque no cenário tecnológico. Diferentemente de metodologias consolidadas, como o Ágil, não há um manifesto oficial nem um órgão regulador que defina com precisão o que é DevOps. Essa ausência de uma definição única faz com que autores e empresas interpretem o conceito de maneiras distintas, tratando-o como uma cultura, um conjunto de práticas, uma filosofia de trabalho ou até mesmo um novo paradigma (DYCK et al., 2015; JABBARI et al., 2016).

De forma geral, o DevOps pode ser entendido como um esforço colaborativo e multidisciplinar voltado à automação da entrega contínua de software, buscando garantir correção, confiabilidade e agilidade (LEITE et al., 2019). O próprio nome revela sua essência: unir "Dev" (development, ou desenvolvimento) e "Ops" (operations, ou operações). A proposta central é integrar áreas tradicionalmente separadas para otimizar a entrega de software, promovendo colaboração constante entre equipes multifuncionais por meio de processos automatizados de desenvolvimento, implantação e monitoramento (DEBOIS, 2008; VALENTE, 2020b; EBERT et al., 2016). Essa integração elimina barreiras, aumenta a eficiência do ciclo de entrega e acelera o retorno de feedbacks, elemento crucial para enfrentar os desafios da era digital.

O movimento surgiu em 2009, quando John Allspaw (operações) e Paul Hammond (desenvolvimento) apresentaram a palestra *"10+ Deploys Per Day: Dev and Ops Cooperation at Flickr"* na *O'Reilly Velocity Conference*. Na ocasião, demonstraram como a aproximação entre operações e desenvolvimento, aliada a uma cultura colaborativa e boas ferramentas, tornava os projetos mais ágeis e escaláveis. Inspirado por essa apresentação, Patrick Debois organizou, no mesmo ano, o primeiro DevOpsDays, evento que consolidou o termo e deu início a um movimento global na área de tecnologia (KIM et al., 2018; DEBOIS, 2009).

Mais do que um conjunto de práticas ou ferramentas, o DevOps representa uma mudança cultural profunda na forma como as organizações desenvolvem, entregam e mantêm software. Ao promover a integração contínua entre desenvolvimento e operações, ele estabelece um ambiente colaborativo capaz de responder rapidamente às demandas do mercado e dos usuários. Compreender sua origem e princípios é o primeiro passo para explorar as metodologias, práticas e tecnologias que dão suporte a esse modelo.

12.2 DevOps no ciclo de desenvolvimento de software

No desenvolvimento de software contemporâneo, especialmente no contexto web, as metodologias ágeis tornaram-se amplamente adotadas, criando um ambiente naturalmente receptivo à cultura DevOps. Essa abordagem pode ser vista como uma evolução do Ágil, expandindo seus princípios para além da programação e abrangendo

todo o ciclo de entrega de software (FREEMAN, 2021). Esse ciclo envolve etapas que exigem colaboração constante entre as equipes, como planejamento, desenvolvimento, testes, empacotamento, lançamento e implantação. Após a disponibilização do sistema, a equipe de operações assume a configuração e o monitoramento contínuo, assegurando a estabilidade e o desempenho da aplicação.

Mais do que uma definição formal, o DevOps busca integrar desenvolvimento e operações para tornar o ciclo de entrega mais ágil, confiável e eficiente. Essa integração se traduz em etapas interconectadas, apoiadas por práticas e ferramentas específicas. Entretanto, um dos maiores desafios desse fluxo ocorre justamente na fase final, durante o lançamento e a implantação do sistema (VALENTE, 2020b). Nesse momento, as diferenças técnicas e culturais entre os times tendem a se acentuar, o que Martin Fowler descreve como *the last mile*, fazendo analogia à etapa mais desgastante de uma maratona (FOWLER, 2014).

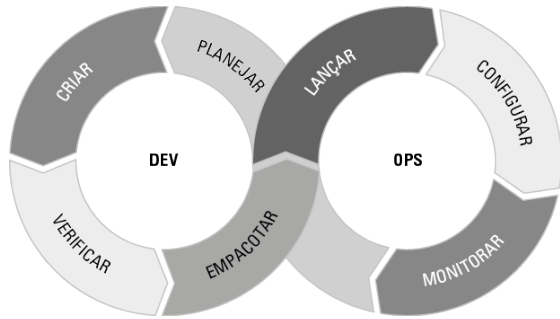


Figura 12.1: A cadeia de processos de DevOps

Fonte: (FREEMAN, 2021)

A Figura 12.1 ilustra como a etapa de lançamento atua como ponto de convergência entre desenvolvimento e operações no ciclo ágil. Tradicionalmente, o *deploy*, isto é, a disponibilização de um sistema aos usuários finais, era realizado por uma equipe específica, em horários definidos e com grandes pacotes de alterações acumuladas ao longo de meses. Esse modelo frequentemente gerava instabilidades e atrasos, pois a preparação de ambientes de teste e produção dependia de processos manuais em servidores físicos internos (VALENTE, 2020b).

Com o avanço de tecnologias como virtualização, sistemas de versionamento, conteinerização e computação em nuvem, a alocação de infraestrutura tornou-se automatizada e acessível aos próprios desenvolvedores (KIM et al., 2018). Somadas à cultura DevOps, essas inovações viabilizam ciclos com *deploys* contínuos e automáticos por meio de *pipelines* de Integração Contínua e Entrega Contínua (CI/CD). Esse modelo oferece maior flexibilidade, rastreabilidade e controle sobre as aplicações, permitindo que novas versões sejam testadas, validadas e implantadas de forma mais eficiente e segura (FOWLER, 2016).

Tabela 12.1: Visão geral de ferramentas DevOps

Ferramenta	Categoria
Jira, Trello	Planejamento
Git, GitHub	Versionamento
Maven, Gradle	Build
Jenkins, GitHub Actions, GitLab CI	CI / CD
Docker	Containers
Puppet	Automação de Deploy
Nagios, Prometheus	Monitoramento

Adaptado de (LEITE et al., 2019)

Para explorar todo o potencial do DevOps, é fundamental adotar ferramentas que intensifiquem a automação e a integração. Essas "ferramentas DevOps" fortalecem a colaboração entre equipes, simplificam a entrega contínua e aumentam a confiabilidade dos projetos (LEITE et al., 2019). Entre elas, destacam-se as soluções de CI/CD, como Jenkins, GitHub Actions e GitLab CI/CD, que automatizam processos de compilação, versionamento, testes e implantação. A Tabela 12.1 apresenta alguns exemplos de ferramentas usadas em diferentes etapas do ciclo DevOps.

Em síntese, o ciclo DevOps é composto por processos interdependentes, apoiados por práticas e ferramentas que asseguram entregas rápidas e confiáveis. Entre essas práticas, a Integração Contínua e a Entrega Contínua desempenham papel central, pois automatizam desde a compilação até a implantação. As próximas seções aprofundarão esses conceitos e apresentarão, de forma prática, a construção de *pipelines* utilizando GitHub *Actions*, demonstrando como aplicar a metodologia no dia a dia do desenvolvimento.

12.3 Ferramentas para o DevOps

O sucesso na adoção da cultura DevOps depende não apenas de princípios e práticas, mas também do uso de ferramentas adequadas em cada etapa do ciclo de desenvolvimento. Essas ferramentas permitem automatizar tarefas, integrar equipes e garantir que o fluxo de trabalho seja ágil, confiável e rastreável.

Nesta seção, serão apresentadas as principais categorias de ferramentas que dão suporte ao DevOps, desde o planejamento inicial do projeto até a implantação e o monitoramento contínuo. O percurso começa com recursos voltados à organização e acompanhamento do trabalho, passa por soluções de versionamento, testes e conteinerização, e culmina nos *pipelines* de Integração Contínua e Entrega Contínua (CI/CD), que consolidam a automação e a entrega contínua de software.

12.3.1 Planejamento - Trello

No contexto das metodologias ágeis, como Scrum e Kanban, o planejamento visual e a gestão de tarefas desempenham um papel central para garantir entregas rápidas e adaptáveis. O Scrum organiza o trabalho em ciclos curtos e incrementais (*sprints*), enquanto o Kanban foca na visualização contínua do fluxo de trabalho. A combinação dessas abordagens, conhecida como *Scrumban*, une a previsibilidade das entregas à flexibilidade de adaptação ao longo do processo (LADAS, 2009; SUBIH et al., 2019).

O Trello é uma ferramenta web que adota a lógica do Kanban, permitindo representar projetos como *boards* (quadros) compostos por listas que organizam o trabalho em etapas e por cartões (*cards*) que representam tarefas específicas (JOHNSON, 2017). Essa estrutura favorece a transparência, a colaboração e o alinhamento entre equipes, valores essenciais à cultura DevOps.

Cada **quadro** é a base de organização do Trello e está vinculado a um **workspace** (área de trabalho), que reúne membros e projetos de uma empresa ou equipe. As **listas** representam fases do fluxo de trabalho, podendo ser personalizadas conforme as necessidades do projeto. Uma configuração bastante utilizada inclui colunas como *Backlog* (tarefas a priorizar), *To Do* (tarefas a iniciar), *Doing* (tarefas em andamento) e *Done* (tarefas concluídas). A Figura 12.2 ilustra um quadro com listas e cartões simulando o fluxo de um projeto real, facilitando o acompanhamento do progresso e a identificação de gargalos.

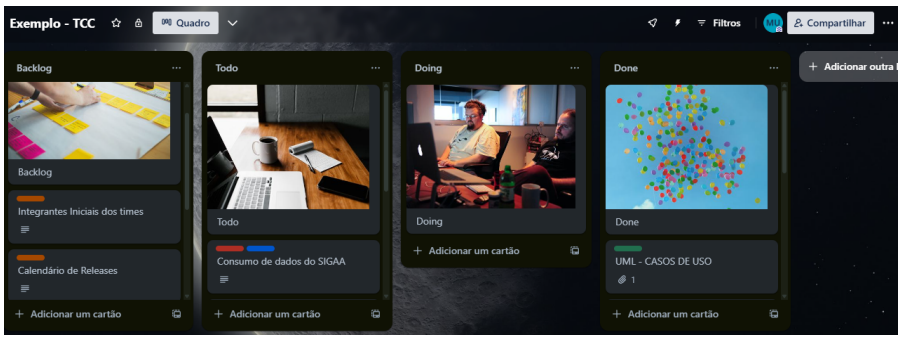


Figura 12.2: Visão geral de um quadro no Trello

Os **cartões** são as unidades fundamentais do Trello e podem representar histórias de usuário, funcionalidades, correções de erros ou qualquer outra tarefa rastreável. Cada cartão pode conter descrições detalhadas, listas de verificação, anexos, comentários, membros responsáveis, datas de entrega e etiquetas para classificação. A Figura 12.3 mostra um exemplo de cartão com campos principais e etiquetas, enquanto a Figura 12.4 apresenta elementos adicionais, como *checklists* e registros de conversas da equipe.

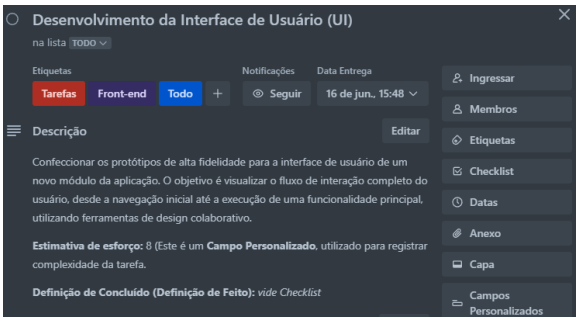


Figura 12.3: Exemplo de cartão no Trello

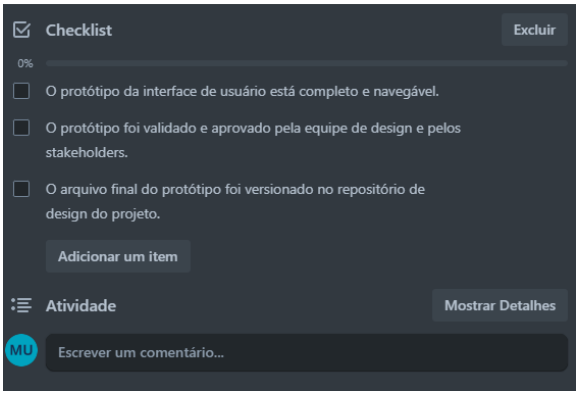


Figura 12.4: Outros elementos de um cartão no Trello

Além das funções básicas, o Trello oferece recursos avançados, como o *board menu*, que centraliza o controle de permissões, integrações e histórico de atividades. Os *Power-Ups*, plugins que expandem funcionalidades, permitem integração com ferramentas como Slack, GitHub, Google Drive e Gmail, além de possibilitar automações que transformam quadros em aplicações dinâmicas. A Figura 12.5 apresenta a interface desse menu de controle.

A flexibilidade do Trello permite ainda representar as etapas de um pipeline de CI/CD, como *PR/Revisão*, *Build e Teste*, *Pronto para Deploy*, *Deploy Realizado* e *Concluído* (MASUD et al., 2022). Na Figura 12.6, cada cartão corresponde a uma tarefa rastreável do início ao fim, com progresso registrado por comentários e no histórico do quadro. Essa visualização centraliza informações, mantém o status visível a toda a equipe e ajuda a eliminar barreiras entre desenvolvimento e operações.

Com as integrações adequadas, os cartões podem ser atualizados automaticamente para refletir o andamento real das atividades. Por exemplo, um cartão pode

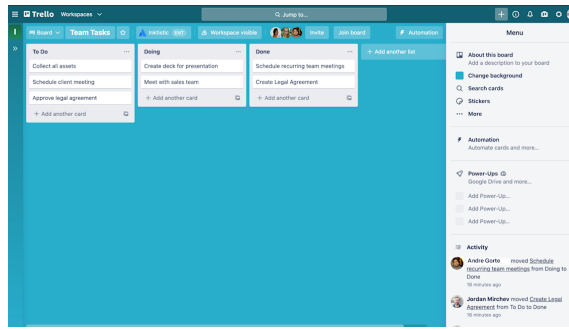


Figura 12.5: Visão geral do *Board Menu*

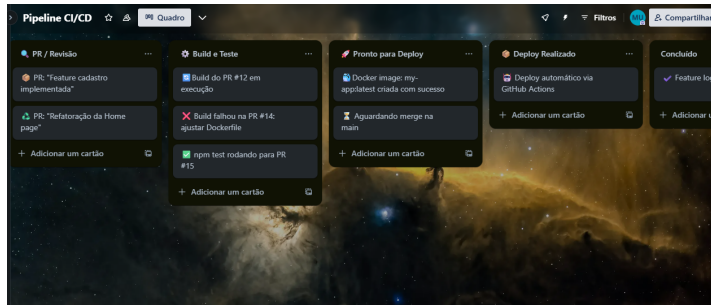


Figura 12.6: Quadro para etapas do Pipeline CI/CD

ser movido para *Deploy Realizado* em caso de sucesso ou para uma lista de retrabalho em caso de falha. Ao unir comunicação, transparência e automação, o Trello reforça princípios fundamentais do DevOps e se consolida como uma ferramenta estratégica para equipes que buscam eficiência nos processos de Integração e Entrega Contínua.

12.3.2 Versionamento – Git e GitHub

O versionamento de código é um pilar essencial para projetos que adotam a cultura DevOps, pois assegura que o código da aplicação esteja sempre acessível, sincronizado e com histórico rastreável para todos os membros da equipe. Ele permite que vários desenvolvedores trabalhem simultaneamente, mantenham um registro organizado das alterações e revertam para versões anteriores em caso de falhas, garantindo integridade, transparência e colaboração no desenvolvimento (KONNORATE et al., 2019).

Para gerenciar esse processo, utilizam-se os Sistemas de Controle de Versão (SCV), que registram todas as modificações no código em um repositório. Eles se dividem em duas categorias:

- **Centralizados (CVCS)** – todos os arquivos e histórico ficam em um servidor central, exemplo: Subversion (SVN) (O'REILLY, 2009);
- **Distribuídos (DVCS)** – cada desenvolvedor mantém uma cópia completa do repositório, podendo trabalhar offline e sincronizar alterações posteriormente, por exemplo: Git (ZOLKIFLI et al., 2018).

Atualmente, os DVCS dominam o mercado, sendo o Git a solução mais utilizada. O Git permite criar *repositórios* locais independentes, registrar alterações com *commits* descritivos, utilizar *branches* para desenvolvimento em paralelo e rastrear modificações por meio de identificadores únicos (*hashes*), características que se integram perfeitamente às práticas do DevOps (The Git Development Team, 2024).

Além do Git, plataformas como GitHub, GitLab, Bitbucket, Azure DevOps e AWS CodeCommit oferecem hospedagem de repositórios Git na nuvem, permitindo colaboração remota, controle de acesso e integração com pipelines de CI/CD (UDDIN et al., 2023). No GitHub, cada repositório contém arquivos, histórico de commits, ramificações e recursos adicionais como *Issues* para registro de tarefas e *Actions* para automação.

Inicialização e clonagem de repositórios: Para iniciar um novo repositório Git local, utiliza-se o comando `git init`, que cria a pasta oculta `.git` com todo o controle de versionamento. A Figura 12.7 mostra esse processo.

```
PS C:\Programação\tcc\exemplo-git> git init
Initialized empty Git repository in C:/Programação/tcc/exemplo-git/.git/
PS C:\Programação\tcc\exemplo-git> git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        index.js

nothing added to commit but untracked files present (use "git add" to track)
```

Figura 12.7: Inicialização de um repositório Git local

Para copiar um repositório já existente, normalmente hospedado no GitHub, utiliza-se o comando `git clone` seguido do link do repositório. A Figura 12.8 apresenta a interface de um repositório no GitHub, e a Figura 12.9 mostra a execução do comando de clonagem.

Após a clonagem, o Git mantém um vínculo com o repositório remoto, normalmente identificado como *origin*. Esse vínculo pode ser verificado pelo comando `git remote -v`, mostrado na Figura 12.10.

Branches e fluxos de trabalho: No Git, *branches* permitem desenvolver funcionalidades ou correções de forma isolada. A *main* é a ramificação principal, onde se mantém o código estável. Outras ramificações comuns são *feature*, *hotfix* e *release* (DRIESSEN, 2010).

O **Git Flow** organiza o trabalho com branches intermediárias (*develop* e *release*) antes de integrar à *main*, enquanto o **GitHub Flow** simplifica criando

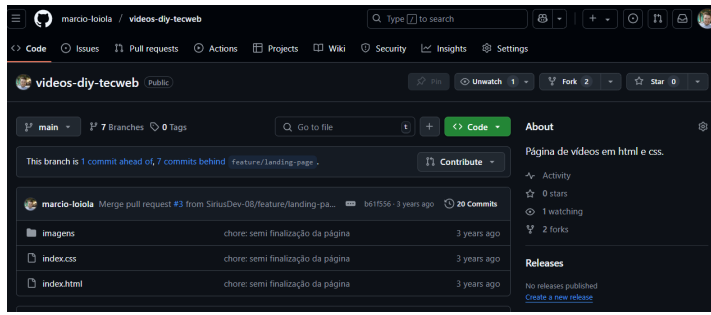


Figura 12.8: Interface de um repositório no GitHub

```
PS C:\Programação\tcc\exemplos> git clone https://github.com/marcio-loiola/videos-diy-tecweb.git
Cloning into 'videos-diy-tecweb'...
remote: Enumerating objects: 194, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 194 (delta 0), reused 0 (delta 0), pack-reused 177 (from 1)
Receiving objects: 100% (194/194), 5.40 MiB | 3.94 MiB/s, done.
Resolving deltas: 100% (62/62), done.
PS C:\Programação\tcc\exemplos> ls

    Directory: C:\Programação\tcc\exemplos

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----          16/04/2025    12:05                videos-diy-tecweb
```

Figura 12.9: Execução do comando `git clone`

ramificações temporárias que são mescladas continuamente na *main* via *pull requests*, favorecendo CI/CD (GITHUB, 2025).

Commits e envio de alterações: Cada *commit* registra um ponto no histórico do projeto, com identificador único e mensagem descritiva. A Figura 12.11 mostra um repositório antes das alterações, enquanto a Figura 12.12 apresenta a preparação e registro das mudanças.

Após o commit, utiliza-se `git push` para enviar alterações ao repositório remoto, como mostra a Figura 12.13. Em fluxos DevOps, alterações na *main* devem passar por revisão para evitar instabilidade.

Pull requests e integração: O *pull request* (PR) é um recurso do GitHub que permite propor alterações, discutir seu conteúdo e aprová-las antes de integrar à *main*. A Figura 12.14 mostra a tela inicial de PRs e a Figura 12.15 um exemplo real do projeto Next.js com múltiplos PRs abertos.

Ao criar um PR (Figura 12.16), o desenvolvedor descreve as mudanças e seleciona revisores. O GitHub verifica automaticamente conflitos e integrações, exibindo as diferenças no código (Figura 12.17).

Após aprovação, o PR é mesclado à *main* (Figura 12.18), gerando um commit de integração (Figura 12.19). Plataformas como GitHub Actions podem executar

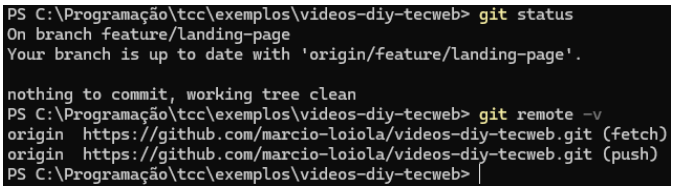


Figura 12.10: Verificação do repositório remoto associado

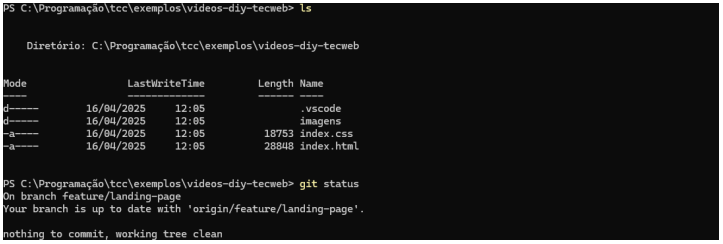


Figura 12.11: *Status* do repositório antes das modificações

pipelines de teste automaticamente antes da mesclagem, garantindo qualidade e estabilidade.

Integração com DevOps: No DevOps, o versionamento com Git e GitHub é mais do que um controle de histórico: é parte fundamental da automação e da integração contínua. O uso de PRs com pipelines automatizados garante que apenas código validado seja integrado à *main*, acelerando as entregas e aumentando a confiabilidade do software.

12.3.3 *Containers* – Docker

No desenvolvimento moderno, equipes multidisciplinares trabalham de forma colaborativa, mas diferenças entre ambientes, como versões de dependências, sistemas operacionais e configurações, podem gerar problemas de compatibilidade, resumidos na frase "na minha máquina funciona".

Uma solução amplamente adotada é o uso de contêineres, que oferecem isolamento, portabilidade e leveza. Diferentemente das máquinas virtuais, que virtualizam todo o hardware, os contêineres compartilham o núcleo do sistema operacional do *host*, encapsulando a aplicação e todas as suas dependências. Isso garante execução consistente em qualquer ambiente, reduzindo o consumo de recursos e o tempo de inicialização.

Docker e seus benefícios: O Docker é a plataforma mais popular para a criação e execução de aplicações em contêineres, com código aberto e uma comunidade ativa (Docker, Inc., 2024). Ele permite construir ambientes isolados, empacotar aplicações, como imagens, e implantá-las localmente ou na nuvem. É

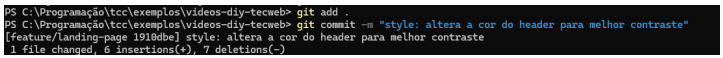


Figura 12.12: Preparação das modificações e criação do *commit*

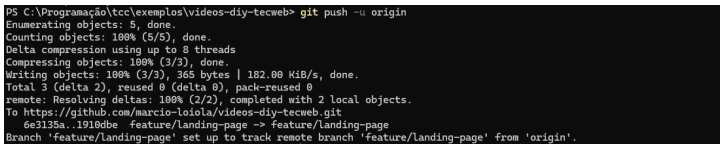


Figura 12.13: Envio de alterações para o repositório remoto

multiplataforma e disponível para Windows, macOS e Linux. No Windows, a instalação é feita pelo *Docker Desktop Installer*, que utiliza o WSL 2 para executar contêineres Linux.

A interface gráfica do Docker Desktop (Figura 12.20) mostra imagens e contêineres ativos, enquanto comandos como `docker run hello-world` (Figura 12.21) e `docker --version` confirmam a instalação.

Imagens e Dockerfile: No Docker, imagens são pacotes com tudo o que a aplicação precisa para rodar, enquanto contêineres são instâncias dessas imagens. O `Dockerfile` define, em texto simples, as instruções para criar uma imagem, incluindo a base (FROM), cópia de arquivos (COPY), execução de comandos (RUN, CMD), variáveis de ambiente (ENV) e portas expostas (EXPOSE). Essas imagens podem ser obtidas ou publicadas em repositórios chamados *registries*, sendo o Docker Hub o principal deles.

No Docker Hub (Figura 12.23), é possível buscar imagens prontas. Por exemplo, a imagem oficial do Python possui diferentes *tags*, incluindo a versão `3.9-alpine`, que é mais leve e ideal para pipelines CI/CD.

Exemplo prático com Flask: Como exemplo, considere uma aplicação Python/Flask (`app.py`) com um Dockerfile baseado na imagem `python:3.9-alpine`. O arquivo define diretório de trabalho, copia o código, instala dependências e expõe a porta 5000.

A Listagem 12.1 apresenta um exemplo de construção da imagem.

Tabela 12.2: Comparativo de tamanhos de imagens comuns e versões Alpine

Tag da Imagem	**Tamanho (Comum)**	**Tamanho (Alpine)**
python:3.9.13	867.66 MB	46.71 MB
node:18.8.0	939.71 MB	164.38 MB
openjdk:17-jdk-bullseye	527.15 MB	105.15 MB

Adaptado Dados do Docker Hub (Docker, Inc., 2025).

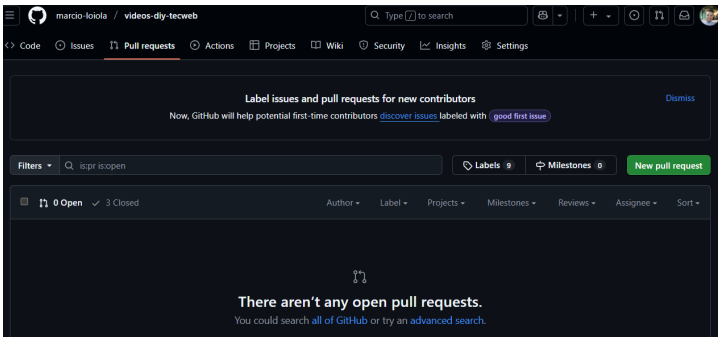


Figura 12.14: Tela inicial de pull requests

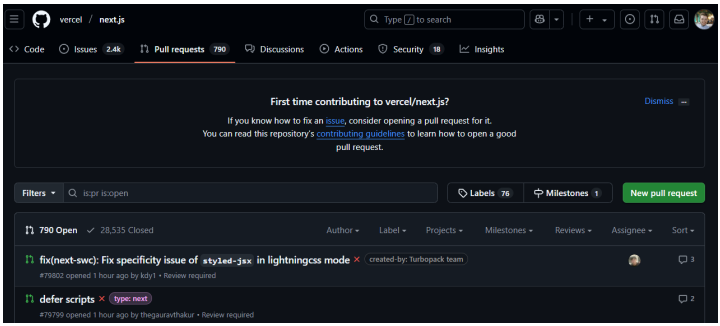


Figura 12.15: Pull requests no repositório Next.js

```
1 docker build -t minha-app-flask .
```

Listagem 12.1: Imagem Docker

A Listagem12.2 apresenta a execução do contêiner.

```
1 docker run -p 5000:5000 minha-app-flask
```

Listagem 12.2: Execução do contêiner

Comandos como `docker ps`, `docker stop` e `docker rm` gerenciam contêineres, enquanto `docker images` e `docker rmi` gerenciam imagens.

Docker no DevOps e na nuvem: O Docker integra-se naturalmente às pipelines de CI/CD, garantindo que desenvolvimento, teste e produção usem o mesmo ambiente. Essa consistência aumenta a confiabilidade e a velocidade das entregas. Na nuvem, provedores como AWS, Azure e GCP oferecem serviços para executar e orquestrar contêineres em escala (ECS/Fargate, AKS, GKE), permitindo

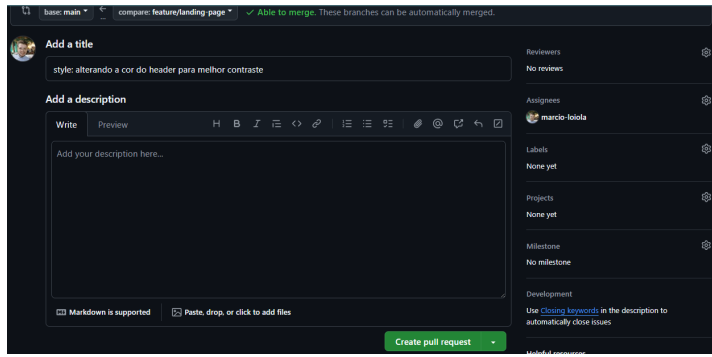


Figura 12.16: Criação de um pull request



Figura 12.17: Visualização das modificações propostas

implantações automatizadas a partir de imagens testadas, fechando o ciclo da entrega contínua.

Em síntese, o uso de contêineres com Docker permite padronizar ambientes, eliminar conflitos de configuração e acelerar entregas, sendo peça-chave para a consistência e escalabilidade de projetos modernos. Na próxima seção, veremos como essa base técnica se conecta às práticas automatizadas de integração e entrega contínua (CI/CD), completando o ciclo DevOps.

12.4 CI/CD – Integração e Entrega Contínua

O CI/CD (Integração Contínua e Entrega Contínua) é um processo essencial no ciclo DevOps, responsável por automatizar o desenvolvimento, os testes e a entrega de software. Em contextos mais maduros, CI/CD pode significar também *Continuous Deployment*, onde todo o processo, inclusive a liberação para produção, é totalmente automatizado (FREEMAN, 2021; HAT, 2025).

Neste capítulo, adota-se a definição tradicional: CI como Integração Contínua, que automatiza a fusão de alterações ao repositório principal, e CD como En-

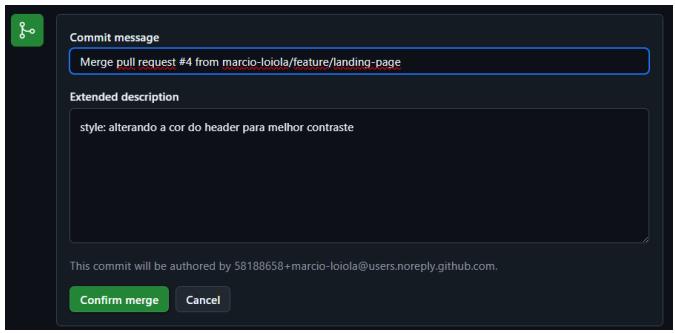


Figura 12.18: *Merge* do pull request



Figura 12.19: Pull request finalizado e integrado à *main*

trega Contínua, em que versões validadas do software são disponibilizadas para implantação, embora a liberação final requeira ação humana (FOWLER, 2010).

A prática de integração contínua surgiu nas metodologias ágeis, como o XP, com o objetivo de detectar e corrigir erros mais rapidamente (BECK, 1999). Ela exige que os desenvolvedores compartilhem frequentemente suas alterações em um repositório comum, o que promove padronização, feedback rápido e maior colaboração (FOWLER, 2023). A cada modificação, o código é automaticamente compilado e testado, garantindo consistência e acelerando entregas.

Já a entrega contínua assegura que o software esteja sempre pronto para produção, ainda que a publicação dependa da aprovação da equipe (KIM et al., 2018). Entre os principais benefícios estão a redução de riscos, possibilidade de deploy com um clique e maior previsibilidade nas atualizações. No modelo mais

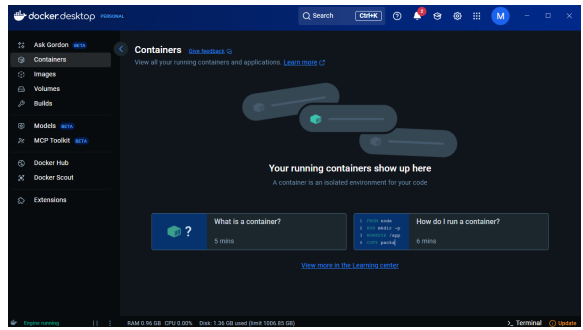


Figura 12.20: Página inicial do Docker Desktop

```
PS C:\Programação> docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
e6590344b1a5: Pull complete
Digest: sha256:ec153840d1e635ac434fab5e377081f17e0e15afab27beb3f726c3265039cfff
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

PS C:\Programação> docker --version
Docker version 28.3.0, build 38b7060
```

Figura 12.21: Execução de teste com `docker run hello-world`

avançado, o deploy contínuo, a publicação é feita automaticamente após a validação, o que exige alto grau de maturidade do processo (HUMBLE, 2010).

Em síntese, CI/CD reduz o tempo entre o desenvolvimento e a entrega de valor ao usuário, permitindo lançamentos frequentes, com maior qualidade e confiabilidade. A seguir, exploramos o GitHub Actions, uma ferramenta integrada que permite construir pipelines CI/CD de forma simples e eficaz.

O GitHub *Actions* é uma plataforma integrada ao GitHub que permite automatizar fluxos de trabalho diretamente nos repositórios, sendo ideal para CI/CD. Sua estrutura baseia-se em arquivos YAML localizados no diretório `.github/workflows`, onde são definidos os gatilhos (eventos), tarefas e condições para execução (GitHub Docs, 2024).

Esses fluxos são iniciados por eventos como `push`, `pull request`, `release` ou horários programados. Dentro dos workflows, os *jobs* representam grupos de etapas

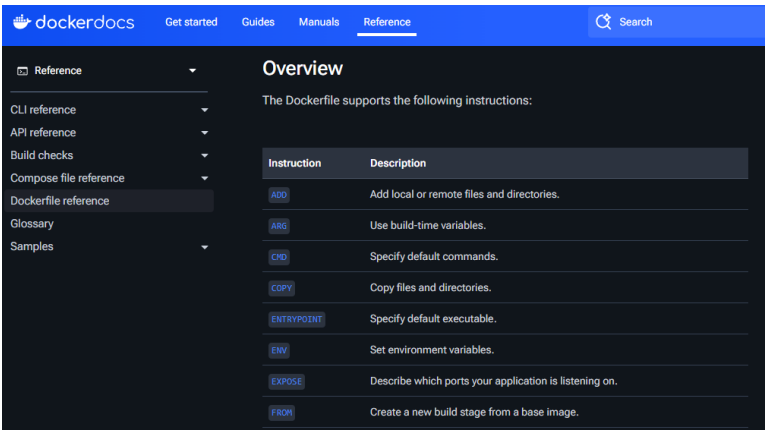


Figura 12.22: Documentação oficial da sintaxe de um Dockerfile

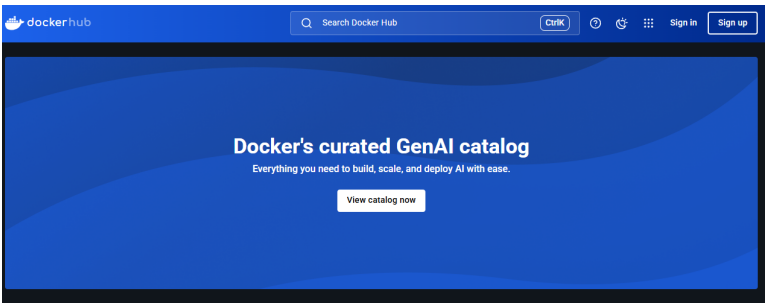


Figura 12.23: Página inicial do Docker Hub

que podem ser executados em paralelo ou sequência, em ambientes chamados *runners* (máquinas Linux, Windows ou macOS, fornecidas pelo GitHub ou configuradas pela organização).

Cada job pode conter várias *actions*, que são scripts reutilizáveis. O GitHub oferece um marketplace com milhares de actions prontas para tarefas como testes, deploy, notificações e análise de código, além de permitir a criação de actions personalizadas, tornando o processo escalável.

Entre os principais diferenciais do GitHub Actions estão sua integração nativa, facilidade de configuração e recursos avançados como execução paralela, builds matriciais, gestão de segredos e suporte a múltiplos ambientes. Além de pipelines CI/CD, a ferramenta permite automatizar tarefas administrativas do repositório, como etiquetar *issues*, gerenciar releases ou interagir com a comunidade.

Assim, o GitHub Actions destaca-se como uma solução moderna, versátil e acessível para automatizar o ciclo de vida do software, reforçando a cultura DevOps e promovendo entregas ágeis, seguras e colaborativas.

12.5 Roteiro sobre configuração e uso de um pipeline CI/CD

Esta seção apresenta um roteiro prático para configurar e utilizar um pipeline de **Integração Contínua e *Delivery* Contínuo (CI/CD)**. O objetivo é simular práticas reais de *DevOps* no contexto de desenvolvimento web, utilizando o GitHub *Actions*.

O GitHub *Actions* permite automatizar fluxos de trabalho (*workflows*) como testes, *builds* e *deploys*. Neste tutorial, aplicaremos esses princípios em um cenário semelhante ao de um ambiente de produção.

Contexto: Você faz parte da equipe de desenvolvimento web de uma ONG dedicada ao cuidado animal, responsável pelo sistema **Adote Fácil**. A equipe é distribuída geograficamente e composta por profissionais especializados em diferentes áreas do projeto web, cada um responsável por módulos distintos do sistema. Diante disso, você identificou que a adoção de metodologias de integração e *delivery* contínuo (CI/CD) neste projeto trará diversas melhorias, como:

- Automatizar tarefas manuais, como a análise e o `merge` de `Pull Requests`, liberando os colaboradores para outras atividades de maior valor;
- Elevar a qualidade e o padrão do código produzido;
- Estimular a criação e atualização constante da documentação técnica;
- Facilitar a coleta rápida e eficiente de *feedbacks* dos usuários;
- E outras vantagens relevantes para o ciclo de desenvolvimento.

Com base nesses benefícios, a equipe elaborou um guia detalhado para a instalação de um ambiente voltado à implementação dessas práticas de CI/CD, visando otimizar o fluxo de trabalho e garantir entregas mais ágeis e seguras.

12.5.1 Configurando o GitHub Actions

Passo 1

Crie um token de acesso pessoal no GitHub e faça um *fork* do projeto **adote-facil**.

I. Para criar um *token*, clique no ícone do seu perfil, localizado no canto superior direito da tela, e selecione **Settings** (Configurações) no menu, conforme demonstrado pela Figura 12.39.

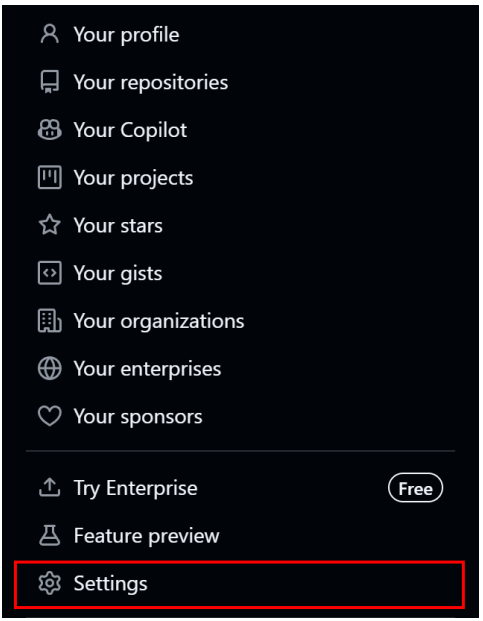


Figura 12.24: Configurações do usuário.

Role a tela para baixo e clique em **Developer settings**, no canto inferior esquerdo (Figura 12.39).

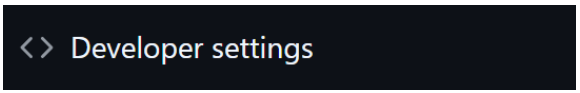


Figura 12.25: Configurações do desenvolvedor.

No menu que aparecer, clique em **Personal access tokens** (Figura 12.39) e depois em **Tokens classic** (Figura 12.39).

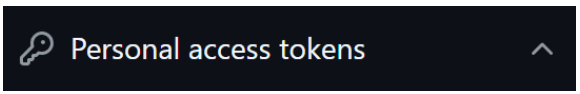


Figura 12.26: Tokens de acesso pessoal.

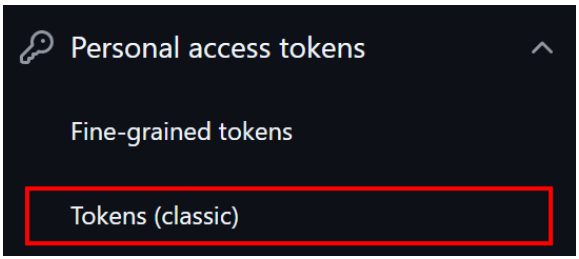
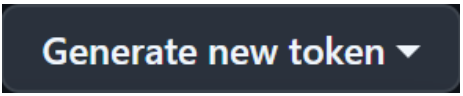
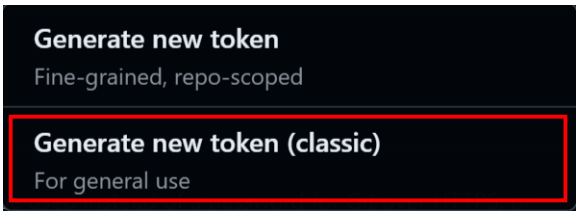


Figura 12.27: Tokens (clássico).

No canto superior direito, clique no botão **Generate new token**(Figura 12.28a) e escolha **Generate new token (classic)** (Figura 12.28b).



(a) Gerar novo token.



(b) Gerar novo token (classic).

Figura 12.28: Gerar novo token

Dê um nome para ele e marque as opções **repo** e **workflow** para gerar o Token. Gere o token mínimo (7 dias) apenas para este experimento, conforme apresentado na Figura 12.29.

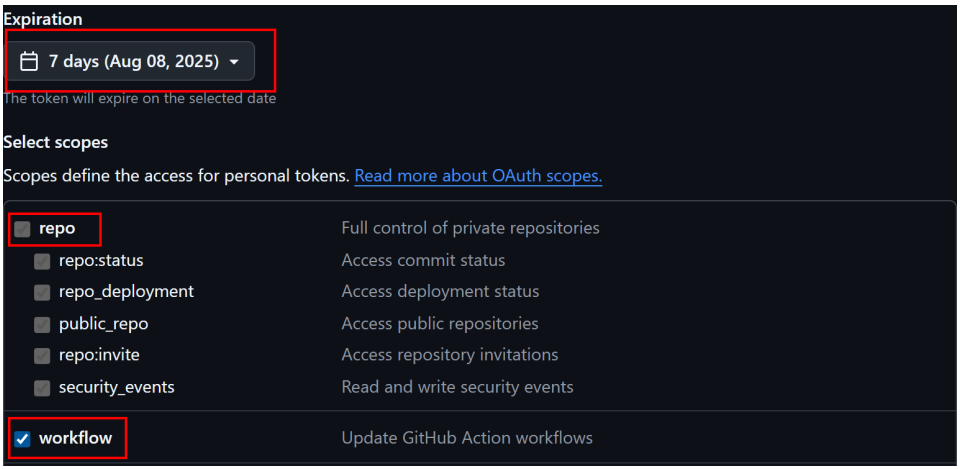


Figura 12.29: Escopos do Token.

Por fim, role a página até o final e clique em **Generate token** (Figura 12.30).

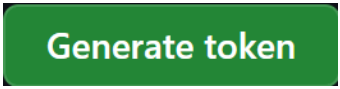


Figura 12.30: Gerando token.

Como este é apenas um experimento, você pode copiar e salvar o *token* em um bloco de notas. Ele pode ser necessário quando o GitHub solicitar sua senha em operações protegidas, como clonar, fazer `push` ou `pull`.

II. Acesse o repositório adote-facil e clique no botão *Fork*, localizado no canto superior direito da página (Figura 12.31).

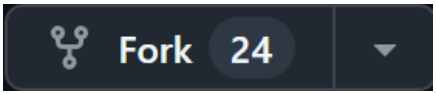


Figura 12.31: Fork.

Em seguida, na tela que será aberta, clique no botão **Create fork** no canto inferior direito, conforme apresentado na Figura12.32.

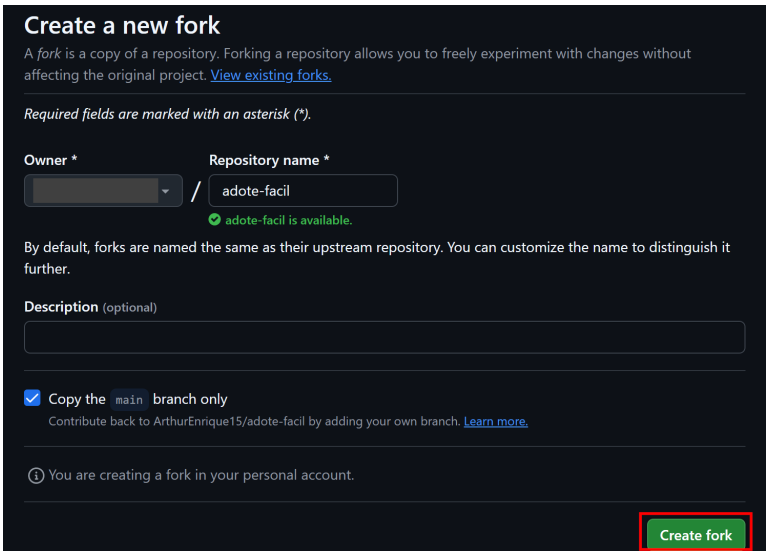


Figura 12.32: Criando um novo fork.

Isso irá criar uma cópia do projeto na sua conta, permitindo que você trabalhe nele livremente.

Passo 2

Clone o seu próprio repositório, criado a partir do `fork`, para a sua máquina local. Para isso, use o comando abaixo, substituindo `<USER>` pelo **seu nome de usuário do GitHub**. A Listagem12.3 apresenta o código para clonagem do repositório `adote-facil`.

```
1 git clone https://github.com/<USER>/adote-facil.git
```

Listagem 12.3: Exemplo de clonagem do repositório adote-facil

Em seguida, dentro do diretório do repositório que você clonou, crie o arquivo `.github/workflows/experimento-ci-cd.yml` com o conteúdo fornecido abaixo.

Para isso, primeiro crie os diretórios `.github` e, dentro dele, o diretório `workflows`, e então salve o arquivo `experimento-ci-cd.yml` nessa última pasta.

A seguir, mostramos como criar os diretórios e o arquivo tanto no *Windows* quanto em distribuições *Linux*. O conteúdo a ser inserido no arquivo `experimento-ci-cd.yml` está abaixo.

Linux: você pode usar os comandos `mkdir` para criar os diretórios, `touch` para criar o arquivo, e outros comandos como `cd`, `ls` e `mv` para navegar e manipular os arquivos, a Listagem 12.4 apresenta os comandos no Linux para criação dos

diretórios e arquivos. Alternativamente, pode usar a interface gráfica (GUI) do seu sistema para criar as pastas e o arquivo, como pode ser observado na Figura 12.33.

```
# Entre no diretório do repositório clonado
cd nome-do-seu-repositorio

# Crie o diretório .github e o subdiretório workflows
# (com -p para criar pais se não existirem)
mkdir -p .github/workflows

# Navegue até o diretório workflows
cd .github/workflows

# Crie o arquivo experimento-ci-cd.yml vazio
touch experimento-ci-cd.yml

# Agora, você pode abrir esse arquivo no editor de texto
# de sua preferência, por exemplo:
nano experimento-ci-cd.yml
# ou
vim experimento-ci-cd.yml

# Cole o conteúdo do pipeline dentro desse
# arquivo e salve
```

Listagem 12.4: Comandos para criar os diretórios e arquivo no Linux

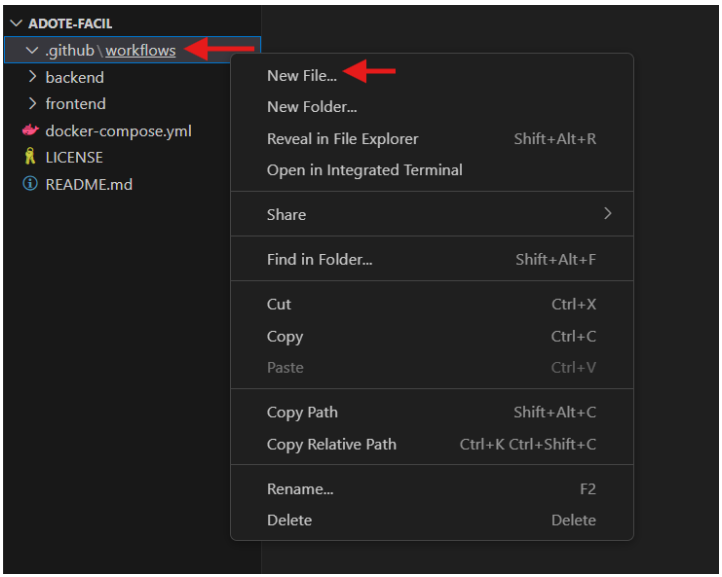


Figura 12.33: Criando arquivo experimento-ci-cd.yml.

Windows: Acesse o diretório do repositório que foi clonado após o *fork* e crie, dentro dele, a pasta `.github` e, em seguida, a subpasta `workflows`.

No VSCode ou na sua IDE de preferência, clique com o botão direito sobre o diretório *workflow* e selecione a opção *New File*. Então, crie o arquivo experimento-ci-cd.yml.

As Listagens12.5, 12.6, 12.7, 12.8 apresentam as definições das tarefas (jobs) que serão executadas.

```yaml
# Nome do workflow
name: experimento-ci-cd

# Evento que aciona o workflow: toda vez que for
# criado um Pull Request para a branch main
on:
  pull_request:
    branches:
      - main


 # Primeiro job: Executar testes unitários
  unit-test:
    # Define o sistema operacional usado no runner
    runs-on: ubuntu-latest
    (Ubuntu na versão mais recente)
    steps:  # Faz o download do repositório no runner
      - name: Checkout do código
        uses: actions/checkout@v4

      - name: Instalar dependências do backend
        run: |
          cd backend #Acessa a pasta do backend
          npm install #Instala as dependências do Node.js

      - name: Executar testes unitários com Jest
        run: |
          cd backend  #Acessa novamente a pasta do backend
          npm test -- --coverage  #Executa os testes e gera relatório
                                  #de cobertura
```

Listagem 12.5: Primeiro job

```
  # Segundo job: Build (construção) das imagens Docker
  build:
  # Esse job só será executado após o job 'unit-test' ser concluído
  # com sucesso
    needs: unit-test
    runs-on: ubuntu-latest
    steps:
      - name: Checkout do código
        uses: actions/checkout@v4

      - name: Configurar Docker Buildx
      # Habilita a ferramenta Buildx do Docker para builds
      # mais avançados
        uses: docker/setup-buildx-action@v2

      - name: Configurar Docker QEMU
      # Permite builds multiplataforma usando emulação (útil em CI)
        uses: docker/setup-qemu-action@v2

      - name: Build das imagens Docker
      # Executa o build das imagens definidas no docker-compose.yml
        run: docker compose build
```

Listagem 12.6: Segundo job

```
# Terceiro job: Subir os containers temporariamente
# para testes básicos de integração
up-containers:
  needs: build  # Esse job depende do job 'build'
  runs-on: ubuntu-latest

  # Definição de variáveis de ambiente necessárias
  # para o backend e banco
  env:
    POSTGRES_DB: adote_facil
    POSTGRES_HOST: adote-facil-postgres
    # Usuário do banco, vindo dos segredos do repositório
    POSTGRES_USER: ${{ secrets.POSTGRES_USER }}
    # Senha do banco
    POSTGRES_PASSWORD: ${{ secrets.POSTGRES_PASSWORD }}
    POSTGRES_PORT: 5432
    POSTGRES_CONTAINER_PORT: 6500

  steps:
    - name: Checkout do código
      uses: actions/checkout@v4

    - name: Criar arquivo .env
    # Define o diretório de trabalho para esse passo
      working-directory: ./backend
      run: |
       # Gera o arquivo .env com as variáveis definidas acima
       echo "POSTGRES_DB=${{ env.POSTGRES_DB }}" > .env
       echo "POSTGRES_HOST=${{ env.POSTGRES_HOST }}" >> .env
       echo "POSTGRES_USER=${{ env.POSTGRES_USER }}" >> .env
       echo "POSTGRES_PASSWORD=${{ env.POSTGRES_PASSWORD }}" >> .env
       echo "POSTGRES_PORT=${{ env.POSTGRES_PORT }}" >> .env
       echo "POSTGRES_CONTAINER_PORT=${{
       env.POSTGRES_CONTAINER_PORT }}" >> .env
      - name: Subir containers com Docker Compose
        working-directory: ./backend
        run: |
          # Sobe os containers em segundo plano
          docker compose up -d
          # Aguarda alguns segundos para garantir que os serviços subam
          sleep 10
          # Encerra os containers após o teste
          docker compose down
```

Listagem 12.7: Terceiro job

```
  # Quarto job: Geração e entrega do artefato do projeto
  delivery:
    needs: build  # Esse job também depende do job 'build'
    runs-on: ubuntu-latest

    steps:
      - name: Checkout do código
        uses: actions/checkout@v4

      - name: Gerar arquivo ZIP do projeto completo
        run: zip -r adote-facil-projeto.zip .
        -x '*.git*' '*.github*' 'node_modules/*'
        # Compacta todos os arquivos do repositório, exceto pastas
        # desnecessárias como .git, .github e node_modules

      - name: Upload do artefato
        uses: actions/upload-artifact@v4
        with:
          # Nome do artefato que aparecerá na aba de artefatos
          # da execução
          name: adote-facil-projeto
          # Caminho do arquivo que será enviado
          path: adote-facil-projeto.zip
```

Listagem 12.8: Quarto job

Este arquivo configura o **GitHub Actions** para ser executado sempre que um evento do tipo `pull_request` for direcionado para a *branch* principal (`main`) do repositório. O *workflow* está dividido em três *jobs*, que representam as etapas da automação:

- **test**: executa os testes do sistema;
- **build**: realiza o processo de compilação do projeto;
- **delivery**: simula uma entrega (*deploy*) da aplicação.

Passo 3

Acesse o diretório do repositório clonado (use o comando `cd` no terminal para entrar na pasta `.../adote-facil/`) e, em seguida, adicione, registre e envie as alterações para o GitHub com os seguintes comandos. A Listagem 12.9 apresenta os comandos para registrar e enviar as alterações para o GitHub.

```
git add --all
git commit -m "Configurando o GitHub Actions"
git push origin main
```

Listagem 12.9: Comandos para registrar e enviar as alterações para o GitHub

12.5.2 Configurando o GitHub Secrets

Em muitos projetos, é comum utilizar variáveis sensíveis - como senhas, *tokens* e chaves de acesso - que não devem ser incluídas diretamente no código por questões de segurança. Para lidar com isso, o **GitHub Actions** oferece o recurso chamado *Secrets*, que permite armazenar variáveis de ambiente de forma criptografada e segura.

A seguir, vamos configurar os *Secrets* para armazenar as informações de conexão com o banco de dados:

Passo 1

Acesse o **seu repositório adote-facil** no GitHub, clique na aba *Settings* (Configurações) na parte superior do repositório (Figura 12.34).

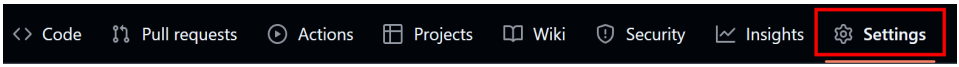


Figura 12.34: Configurações do repositório.

Em seguida, no menu lateral, vá em *Secrets and variables*, depois em *Actions* (Figura 12.35).

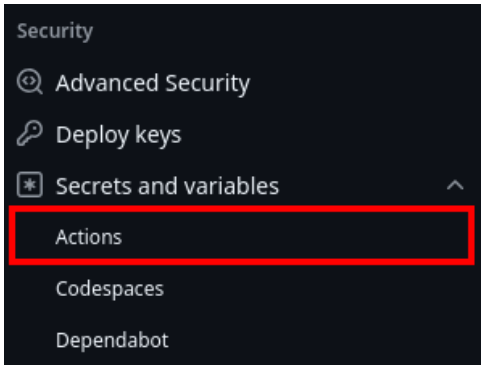


Figura 12.35: Actions.

Por último, clique no botão New repository secret para adicionar um novo segredo.

Passo 2

Crie dois segredos (secrets) com os seguintes valores (Figuras 12.36, e 12.37).

- **POSTGRES_USER**
  - **Name:** `POSTGRES_USER`
  - **Secret:** `Postgres`
- **POSTGRES_PASSWORD**
  - **Name:** `POSTGRES_PASSWORD`
  - **Secret:** `postgres`

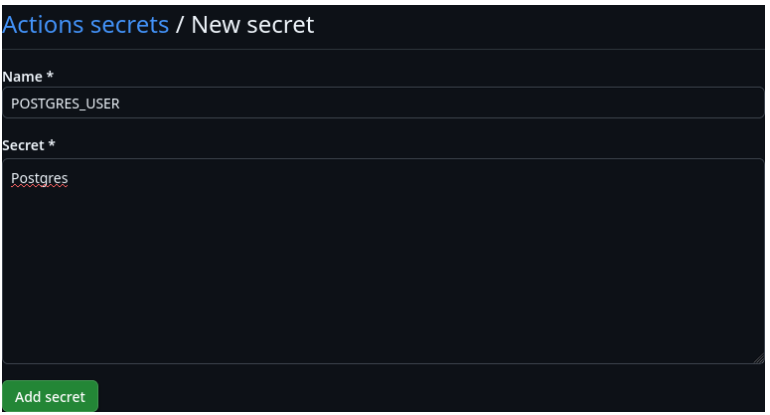


Figura 12.36: Configurando postgres user.

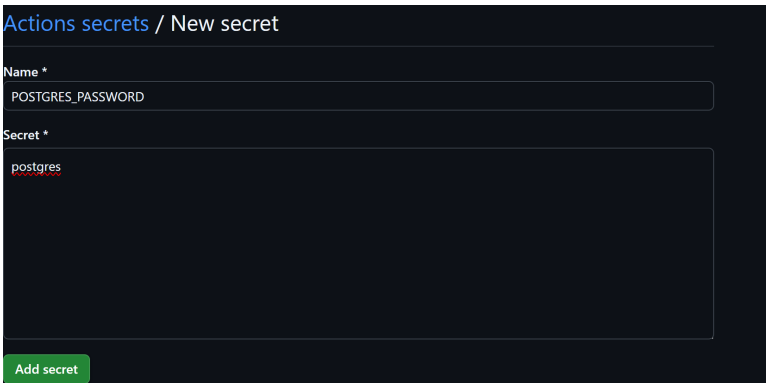


Figura 12.37: Configurando postgres password.

Clique em **Add secret** para adicionar os novos segredos.

Esses valores simulam um cenário de acesso ao banco de dados. Eles serão utilizados automaticamente no workflow `.github/workflows/experimento-ci-cd.yml`.

12.5.3 Criando um Pull Request (PR) com bug

Este caso simula um cenário em que o código possui um erro (bug), o que faz com que o pipeline de CI/CD falhe. Como resultado, o GitHub bloqueia automaticamente o merge na branch main, impedindo que código com falhas seja integrado à versão principal do projeto.

Passo 1

Vamos simular que o teste da função responsável por atualizar o e-mail do usuário está falhando — ou seja, está retornando um valor diferente do esperado.

No seu editor de código, localize o arquivo:
`/backend/src/services/user/update-user-spec.ts`
(conforme mostrado na Figura 12.38) e faça as seguintes alterações:

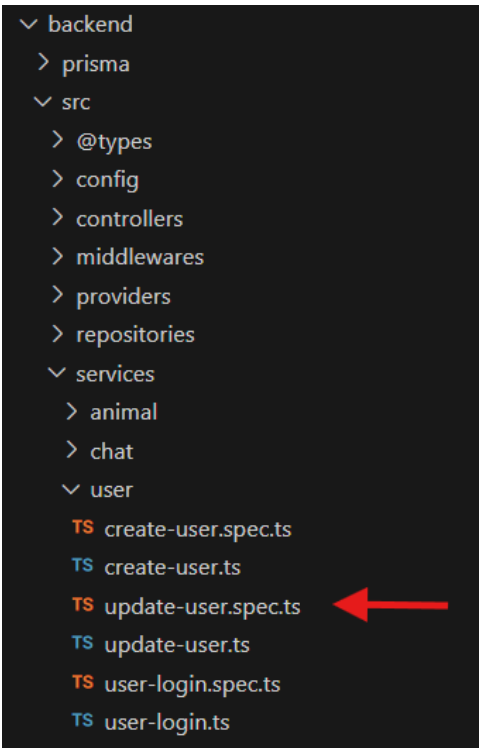


Figura 12.38: Arquivo update-user-spec.ts.

- Comente a linha original do teste (a linha 92 do arquivo), que contém o comando expect(...).
- Logo abaixo, insira o seguinte comando, conforme demonstrado na Figura 12.10 de teste que utiliza um e-mail incorreto.

```
expect(result).toEqual(Success.create({
...updatedUser, email:'email-errado@mail.com' }))
```

Listagem 12.10: Configuração do email

```
test('should return updated user on success', async () => {
  const result = await sut.execute(defaultParams)
  //expect(result).toEqual(Success.create(updatedUser))
  expect(result).toEqual(Success.create({ ...updateUser, email: 'email-errado@mail.com' })}
})
```

Figura 12.39: Mudança no código.

Passo 2

Após fazer as modificações no código, crie uma nova branch, faça o commit das alterações e envie para o repositório remoto com os seguintes comandos, conforme demonstrado na Listagem 12.11.

```
git checkout -b bug
git add --all
git commit -m "Alterando o arquivo update-user-spec.ts"
git push origin bug
```

Listagem 12.11: Comandos para fazer o commit e publicar no repositório

Passo 3

Em seguida, crie um Pull Request (PR) com suas modificações. Para isso, acesse no navegador a seguinte URL, substituindo `<USER>` pelo seu nome de usuário no GitHub: ⟨https://github.com/⟨USER⟩/adote-facil/compare/main...bug.⟩

Nessa página, revise as alterações feitas. Após a verificação, clique no botão Create pull request. Na janela que abrir, adicione uma breve descrição do PR e confirme a criação clicando novamente em *Create pull request.*, conforme demostrado na Figura 12.40.

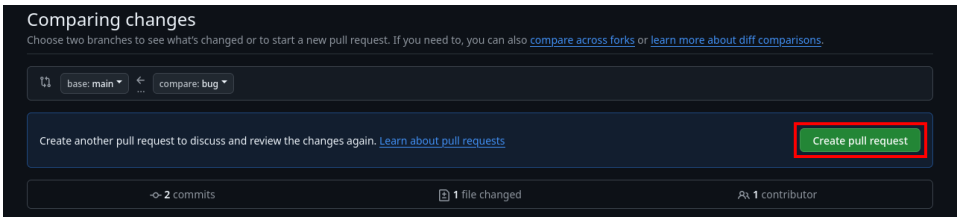


Figura 12.40: Criar Pull Request.

Assim que o PR for criado, o pipeline configurado no arquivo experimento-ci-cd.yml será automaticamente acionado pelo GitHub Actions. Como inserimos um bug, os testes irão falhar, essa falha será exibida na página de execução do pipeline. Você pode acompanhar o status do processo acessando a aba Actions do seu repositório.

Em resumo, o servidor de CI/CD detectou automaticamente um problema no código enviado, impedindo sua integração ao branch principal do projeto, conforme demonstrado na Figura 12.41.

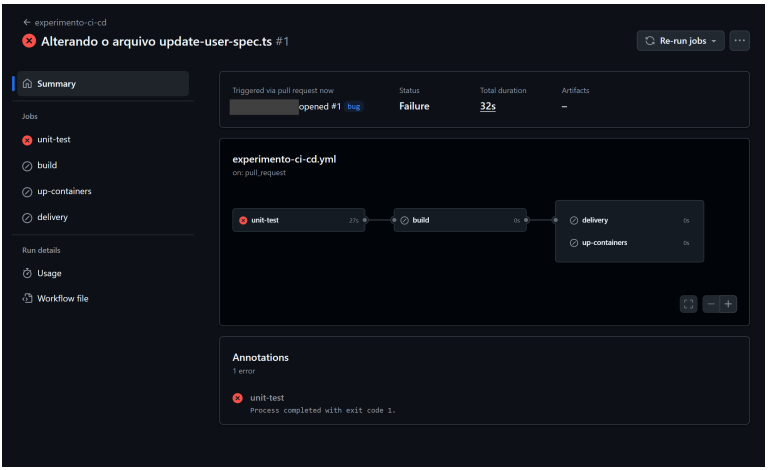


Figura 12.41: Visualização da falha no pipeline.

12.5.4 Criando um Pull Request (PR) com a correção.

Vamos restaurar o arquivo ao seu estado original. Para isso, descomente a linha 92 e remova a linha 93. Assim, ao criar um novo PR, os testes serão executados com sucesso, sem apresentar falhas.

Após fazer essas alterações, salve o arquivo e crie uma nova branch para corrigir o bug. Em seguida, execute os comandos conforme demonstrado na Figura 12.12 para adicionar, commitar e enviar as mudanças ao repositório remoto.

```
git checkout -b fixture
git add --all
git commit -m "Consertando a função Test"
git push origin fixture
```

Listagem 12.12: Comandos para fazer o commit e publicar no repositório com novas alterações

Caso solicitado, informe seu nome de usuário e o token que criamos na `Tarefa 1` para autenticação.

Em seguida, crie novamente um novo Pull Request (PR) com a correção. Para isso, acesse no navegador a seguinte URL, substituindo pelo seu nome de usuário no GitHub: ⟨https://github.com/⟨USER⟩/adote-facil/compare/main...fixture⟩, conforme demonstrado na Figura 12.42.

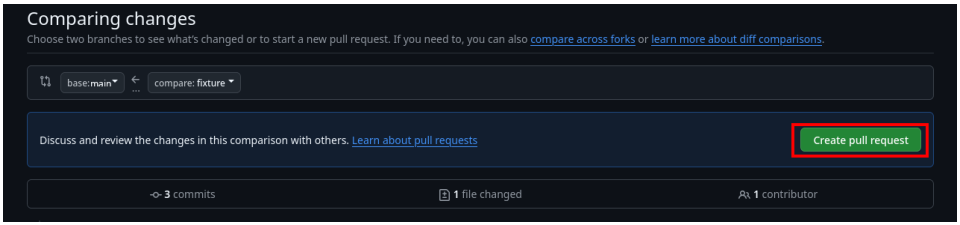


Figura 12.42: Criar Pull Request com correção novamente

Nessa página, revise as alterações realizadas. Em seguida, clique no botão *Create pull request*, no canto superior direito da tela. Na janela que abrir, insira uma breve descrição do PR e confirme a criação clicando novamente no botão Create pull request, no canto inferior direito.

Você pode acompanhar o andamento do pipeline acessando a aba *Actions* do repositório e selecionando o nome do PR em execução, conforme demonstrado na Figura 12.43.

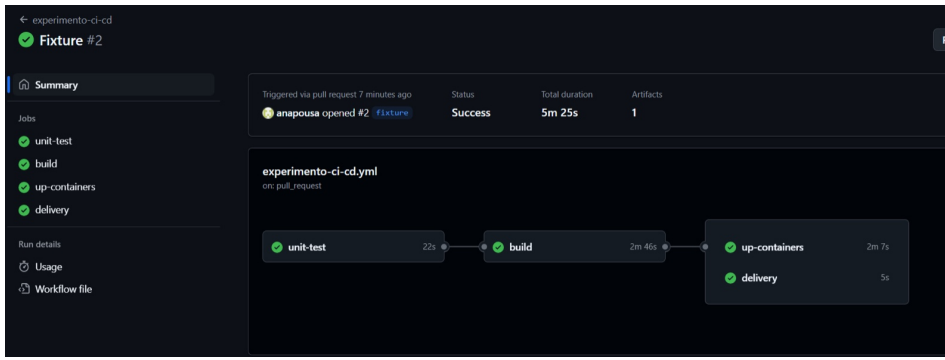


Figura 12.43: Pipeline aceito.

Após a criação do PR, o GitHub Actions iniciará automaticamente o pipeline, que executará os testes, realizará o build e gerará um arquivo *.zip* do projeto. Ao finalizar, o arquivo *.zip* estará disponível para download.

Capítulo 13

Arquitetura de Software

William Bruno Sales de Paula Lima (UFC)

Gustavo Vale (UNILAVRAS)

Fischer Ferreira (UNIFEI)

Este capítulo apresenta as arquiteturas de software mais utilizadas no desenvolvimento de sistemas web. Ele está organizado da seguinte forma: a Seção 13.1 apresenta os principais conceitos da arquitetura de software. Nas Seções 13.2 - 13.6, exploram-se o contexto de utilização, as vantagens, as desvantagens e exemplos práticos das arquiteturas: monolítica, cliente-servidor, microsserviços, orientada a mensagens e publicador-consumidor, respectivamente. Na Seção 13.7 apresenta exemplos práticos dessas arquiteturas.

13.1 Introdução

A arquitetura de software pode ser definida como um conjunto de boas práticas de organização dos componentes de um sistema, que, por vezes, é confundida com o design de software. Em geral, a palavra arquitetura é utilizada para se referir a componentes ou estruturas em um alto nível de organização, enquanto que a palavra *design* é utilizada para se referir a uma visão micro de uma estrutura, observando os detalhes dos elementos (MARTIN, 2019). Desta forma, a arquitetura trata da estrutura macro do sistema, enquanto o design se ocupa dos detalhes dos componentes.

A arquitetura de software exerce um papel central na definição da estrutura e organização de um sistema, garantindo que ela atenda tanto aos requisitos funcionais quanto aos não-funcionais, como desempenho, segurança e escalabilidade. Além disso, a arquitetura representa as principais decisões estruturais do software, definindo como os componentes se relacionam e como o sistema deve evoluir ao longo do tempo. Uma arquitetura bem planejada permite que mudanças nos requisitos sejam atendidas de forma mais eficiente, evitando retrabalho e reduzindo o custo de manutenção do software, além de facilitar a escalabilidade para atender a demandas futuras (SOMMERVILLE, 2015).

A utilização de arquiteturas de software tem como objetivo reduzir o esforço humano na manutenção do sistema de software (MARTIN, 2017). À medida que desenvolvedores constroem sistemas de maneira rápida e sem preocupação com a organização dos elementos deste sistema, eles estão fadados a necessitar de maiores esforços na manutenção do software do que em desenvolver novas funcionalidades deste software.

Com o surgimento da Programação Orientada a Objetos (POO), tornou-se mais simples pensar o software a partir de uma visão mais ampla, uma vez que as características desse paradigma favorecem análises em nível mais alto dos componentes do sistema (MARTIN, 2017). Entre as principais vantagens da POO, destaca-se o polimorfismo, entendido como a capacidade de uma classe herdar métodos de uma classe base e implementá-los mantendo a mesma assinatura, porém com lógicas distintas. Esse conceito permite ao arquiteto estruturar o software segundo uma arquitetura de *plugins* (MARTIN, 2017), na qual se define um componente abstrato comum. A partir dele, diversos componentes concretos

podem ser desenvolvidos, preservando a abstração das implementações em relação ao usuário final.

A escolha das tecnologias a serem utilizadas no desenvolvimento da aplicação, levando vários fatores em consideração, como o paradigma da linguagem utilizada, também é considerada uma etapa importante na construção de uma boa arquitetura. Dessa forma, são igualmente importantes para a construção de um software bem organizado as atividades relacionadas à manutenção de uma boa estrutura do código e à concepção da arquitetura inicial do sistema (MARTIN, 2017).

A arquitetura de um sistema pode ser baseada em um padrão arquitetural (SOMMERVILLE, 2015). Um padrão arquitetural consiste em uma forma de organizar os componentes de um sistema, baseada em padrões de diversas arquiteturas de outros sistemas que possuem propósito e requisitos similares. Nas próximas sessões, serão discutidos alguns padrões arquiteturais conforme mencionado no início deste capítulo.

13.2 Arquitetura Monolítica

Na arquitetura monolítica, todo o sistema é desenvolvido como uma única unidade indivisível de código. Variáveis, classes, funções e demais implementações estão concentradas em um único código-fonte. Como consequência, os diferentes elementos e funcionalidades do sistema permanecem fortemente acoplados entre si. Além disso, toda a aplicação é executada como um único processo, caracterizando o modelo monolítico (FOWLER; LEWIS, 2014; VALENTE, 2020c).

Os sistemas monolíticos tendem a ser convenientes nas fases iniciais de um projeto. Isso ocorre porque reduzem a carga cognitiva da equipe de desenvolvimento durante a construção e a implementação de novas funcionalidades, uma vez que todos os elementos do sistema estão concentrados em uma única unidade. No entanto, diante do rápido crescimento dos sistemas modernos, a capacidade de escalar e de manter esse tipo de sistema torna-se questionável. As principais vantagens de construir um sistema com arquitetura monolítica são: facilidade de desenvolvimento e implantação, facilidade para configurar testes e depurar o código. Como todos os elementos do sistema se encontram em uma única unidade, torna-se fácil para os desenvolvedores construírem novas funcionalidades, testá-las e implementá-las em ambiente produtivo, sendo este o ambiente que hospeda o produto utilizado pelo usuário final. Além disso, torna-se fácil para os desenvolvedores depurarem o código-fonte, visto que todos os elementos estão centralizados em um único código.

No entanto, esta arquitetura também apresenta desvantagens, dentre elas: complexidade no desenvolvimento de um grande sistema monolítico, falta de escalabilidade e flexibilidade no desenvolvimento e complexidade na implementação. Quando um sistema monolítico torna-se muito grande, os desenvolvedores enfrentam dificuldades em desenvolver novas funcionalidades dado o alto nível de complexidade e acoplamento do sistema (FOWLER; LEWIS, 2014).

Além disso, não existe uma maneira de escalar a quantidade de servidores alocados para o processamento de uma funcionalidade específica, pois todo o sistema é monolítico e compartilha os recursos de um servidor ou de um grupo de servidores (FOWLER; LEWIS, 2014). Portanto, se uma determinada funcionalidade do sistema necessita de mais recursos em determinado momento, não há uma maneira fácil de realizar essa operação, pois todos os recursos do grupo de servidores são compartilhados.

Em outras palavras, como todos os elementos do sistema se encontram agrupados em uma única unidade nesta arquitetura, todo o sistema precisa ser implementado em ambiente de produção a cada vez que uma nova funcionalidade é construída (FOWLER; LEWIS, 2014). Assim, em sistemas muito complexos, a implementação de nova versão no ambiente em produção pode ser muito demorada e custosa, considerando que muito capital humano tem de ser investido para manter e acompanhar a implementação do sistema.

Assim, não há flexibilidade na escolha das tecnologias a serem utilizadas para desenvolver diferentes funcionalidades. Ou seja, mesmo que times diferentes sejam responsáveis por desenvolver e manter funcionalidades distintas do sistema, estes times têm de adotar o mesmo conjunto de tecnologias a serem utilizadas por todo o sistema.

Portanto, sistemas monolíticos podem ser fáceis de construir e implementar quando têm poucas funcionalidades. No entanto, à medida que a sua complexidade aumenta, o capital humano necessário para construir, testar e implementar novas funcionalidades também aumenta, devido ao alto acoplamento do sistema. Quando um sistema monolítico se torna muito complexo de manter, é o momento ideal para considerar uma nova arquitetura.

13.3 Cliente-Servidor

Nesta arquitetura, organiza-se os elementos separados em cliente e servidor, sendo o servidor responsável por prover dados e serviços, como data centers ou serviços de computação em nuvem, e o cliente aquile que consome tais dados e serviços, via Internet ou rede local e representa estes dados obtidos na interface do usuário (SOMMERVILLE, 2015). É importante ressaltar que o servidor não necessariamente precisa ser um banco de dados, podendo ser um elo entre os clientes consumidores e os bancos de dados que armazenam as informações.

O propósito da arquitetura cliente-servidor é separar a responsabilidade pelo processamento e pelo armazenamento de dados da visualização desses dados pelos usuários em um sistema web (SOMMERVILLE, 2015). Enquanto o servidor realiza o armazenamento e a gestão dos dados e garante o envio, de forma consistente e escalável, das informações pela rede, o cliente pode se preocupar em consumir todas essas informações por meio de requisições, utilizando métodos HTTP, HTTPS, FTP ou SMTP, e aplicá-las da forma como foi projetado para fazê-lo. Assim, aplicando esse tipo de arquitetura a sistemas web, consegue-se reduzir o acoplamento entre

os elementos que armazenam e os que processam e garantem a visualização das informações ao usuário (SOMMERVILLE, 2015).

Na arquitetura cliente-servidor, o cliente envia uma requisição de dados a um determinado servidor, que é trafegada pela rede. Uma vez aceita a requisição pelo servidor, ele retorna ao cliente uma resposta à sua chamada, podendo esta resposta conter dados, imagens ou até mesmo uma mensagem de acesso negado.

A Figura 13.1, adaptada de (INGALLS, 2021b), apresenta um exemplo de arquitetura cliente-servidor. Como pode ser observado, o cliente faz requisições da API que está localizada no servidor. Sendo que o servidor tem acesso ao banco de dados e ao conjunto de arquivos necessários. Feitas as devidas validações da requisição pelo servidor, serão retornados ao cliente os dados requisitados juntamente a um código HTTP de sucesso na requisição. Caso o pedido seja negado pelo servidor, este retorna ao cliente um código HTTP de erro, geralmente 404.

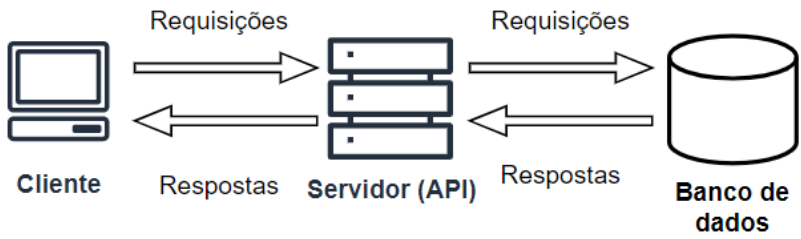


Figura 13.1: Exemplo de arquitetura cliente-servidor

As principais vantagens de construir um sistema web com uma arquitetura cliente-servidor são: desacoplamento de elementos do software e gerenciamento de cargas de requisições aos servidores. O desacoplamento de elementos do software permite que desenvolvedores construam funcionalidades para clientes ou servidores de maneira separada, reduzindo o tempo, o esforço e o capital necessários para resolver erros devido ao alto acoplamento. Além disso, o gerenciamento de cargas de trabalho nos servidores garante que mais servidores sejam provisionados devido à alta demanda de requisições dos clientes, melhorando a escalabilidade e permitindo a criação de sistemas mais robustos (SOMMERVILLE, 2015).

No entanto, esse tipo de arquitetura apresenta algumas desvantagens, sendo uma das principais a exposição dos servidores conectados à Internet a ameaças, como vírus e ataques de negação de serviço distribuída (*Distributed Denial of Service* – DDoS). Nesse tipo de ataque, múltiplas máquinas comprometidas enviam um grande volume de requisições simultâneas ao servidor, com o objetivo de sobrecarregá-lo e torná-lo indisponível para usuários legítimos. Como consequência, um ataque bem-sucedido pode comprometer o funcionamento de todos os serviços dos clientes, uma vez que estes dependem das respostas do servidor. Dessa forma, mesmo que envolva custos elevados, torna-se essencial que o servidor seja desenvolvido de acordo

com as melhores práticas de segurança da informação, a fim de reduzir riscos de ataques e de indisponibilidades futuras (SOMMERVILLE, 2015).

13.4 Arquitetura de Microsserviços

A arquitetura de microsserviços consiste em separar os elementos do software em módulos ou serviços distintos, autossuficientes e independentes entre si, chamados de microsserviços. Assim, todos os módulos do sistema são desenvolvidos e decompostos em espaços de memória distintos, de modo que a execução de um módulo não afeta a de outro (FOWLER; LEWIS, 2014; VALENTE, 2020c). Desta forma, não corre-se o risco de um microsserviço acessar ou modificar uma variável ou objeto de outro serviço, minimizando problemas de concorrência de acesso a informações que podem ocorrer em arquiteturas monolíticas.

A separação dos elementos do software em microsserviços deve ser baseada na funcionalidade de negócios de cada elemento no sistema, de modo que elementos responsáveis por determinadas funcionalidades de negócios fiquem agrupados em um único microsserviço (FOWLER; LEWIS, 2014). A Figura 13.2 ilustra um exemplo de design de um aplicativo de *e-commerce* que possui diversas funcionalidades, como o carrinho de compras, o sistema de busca por produtos, a visualização de produtos na tela inicial e a seleção especializada de produtos melhor avaliados de acordo com as compras do usuário.



Figura 13.2: Exemplo de aplicativo de e-commerce

Na Figura 13.3, os elementos do aplicativo foram separados em microsserviços diferentes de acordo com o seu valor de negócio gerado. Os microsserviços ilustrados referem-se ao serviço de produtos melhor avaliados, chamado de "ms-melhor-avaliados", ao sistema da tela inicial, chamado de "ms-tela-inicial", ao serviço do carrinho, chamado de "ms-carrinho", e ao serviço de busca, chamado de "ms-busca".

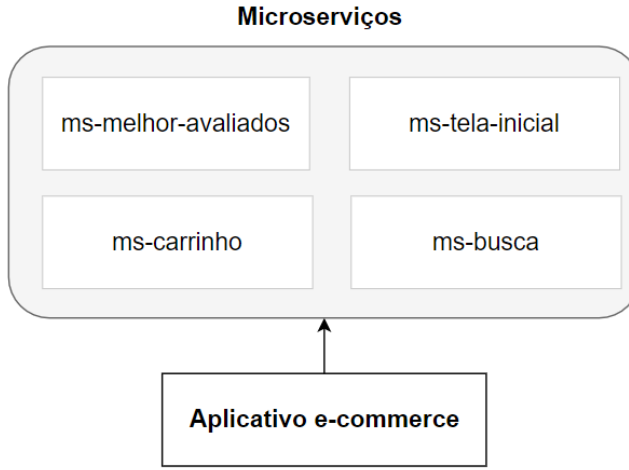


Figura 13.3: microsserviços do aplicativo de *e-commerce*

Cada microsserviço é independente dos demais, o que significa que o time de desenvolvimento pode projetar, desenvolver, testar e implementar cada microsserviço separadamente, sem impactar os demais. Dessa forma, diminuem-se os riscos de que uma modificação em determinado microsserviço venha a afetar negativamente outras funcionalidades do sistema, reduzindo o acoplamento do sistema e o esforço humano necessário para construir e manter novas funcionalidades no software (VALENTE, 2020c).

Como todos os microsserviços são implementados de forma independente, cada serviço pode ser implementado por um time específico (VALENTE, 2020c), além de utilizar um conjunto de tecnologias próprias ao serviço. Como exemplo, considere ainda o aplicativo de *e-commerce*, cada microsserviço ilustrado na Figura 13.3 é implementado por um time diferente e pode ser construído utilizando linguagens de programação diferentes. Além disso, o time responsável por implementar a funcionalidade da busca poderia utilizar a linguagem de programação Python para construir algoritmos de aprendizado de máquina, enquanto que os outros serviços poderiam ser implementados em Java, como ilustra a Figura 13.4:

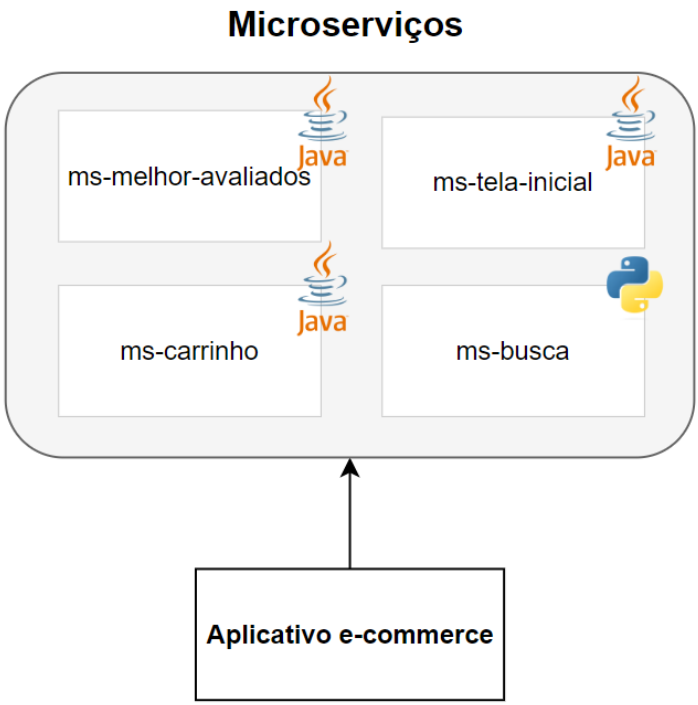


Figura 13.4: Conjuntos de tecnologias de microsserviços

Na Figura 13.4, cada microsserviço do sistema de *e-commerce* é ilustrado como sendo desenvolvido com uma linguagem de programação diferente, sendo os

microsserviços de melhor avaliação, tela inicial e carrinho, desenvolvidos em Java, e o do mecanismo de busca, desenvolvido em Python.

Outro ponto positivo da adoção da arquitetura de microsserviços é a escalabilidade dos sistemas que a adotam. Caso o servidor que executa os serviços esteja enfrentando problemas de desempenho, uma possível solução seria aumentar a capacidade de memória e de processamento do servidor para que ele possa dar conta da demanda dos serviços. Assim, chama-se de escalabilidade vertical o processo de aumentar a capacidade de um servidor para que ele consiga atender a demanda dos seus serviços (VALENTE, 2020c). Ademais, é possível instanciar mais servidores para processarem serviços específicos que estejam com gargalo no processamento, aumentando assim a quantidade de servidores disponíveis para dar conta da demanda do sistema, caracterizando assim a escalabilidade horizontal (VALENTE, 2020c).

Os diferentes microsserviços podem se comunicar entre si de diversas maneiras. As duas principais formas de implementar esta comunicação são via chamadas de API, utilizando protocolos de comunicação HTTP ou REST (VALENTE, 2020c; FOWLER; LEWIS, 2014), ou por meio de serviços de mensagens, como *RabbitMQ*. Diferentemente dos sistemas monolíticos, nos quais a comunicação do sistema ocorre por meio de chamadas de funções e métodos, na arquitetura monolítica torna-se necessária a implementação de comunicações mais complexas, dada a divisão dos serviços (FOWLER; LEWIS, 2014).

A Figura 13.5 ilustra a comunicação entre serviços por meio de chamadas a APIs. Assim, cada microsserviço possui um *endpoint*, utilizado por outros microsserviços para realizar requisições via protocolos HTTP ou HTTPS, que retorna as respostas correspondentes às requisições aos microsserviços que as solicitam.

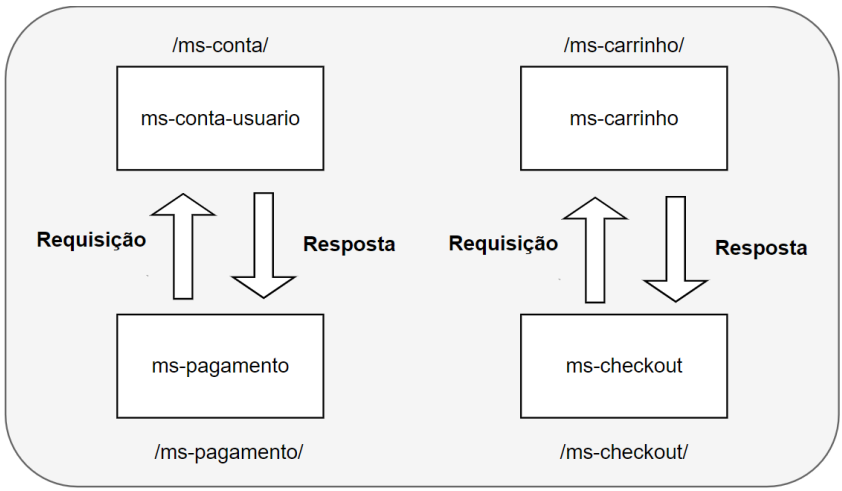


Figura 13.5: Chamada via API

Na estratégia de comunicação síncrona entre microsserviços via chamadas de API, cada microsserviço possui um *endpoint* específico e único. Assim, os outros microsserviços realizam chamadas via protocolo HTTP ou HTTPs pela rede à qual estão conectados para o *endpoint* do serviço desejado. Após isso, a chamada será tratada pelo serviço requisitado e será retornada ao serviço requisitante uma resposta, que pode conter dados ou um código HTTP de erro.

A segunda principal forma de comunicação entre microsserviços é por meio de um intermediário de mensagens, em que um conjunto de microsserviços publica mensagens em um corretor de mensagens, uma estrutura intermediária que contém filas de mensagens que serão enviadas por microsserviços publicantes e recebidas por microsserviços assinantes, como ilustra a Figura 13.6.

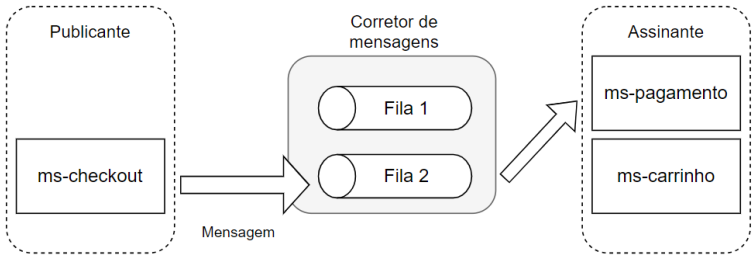


Figura 13.6: Comunicação assíncrona via corretor de mensagens de microsserviços

As principais desvantagens da utilização de uma arquitetura de microsserviços referem-se à complexidade adicionada à organização do sistema ao utilizá-la (FOWLER; LEWIS, 2014). Mesmo que o acoplamento de código entre os elementos do software tenha sido reduzido entre os microsserviços, esses ainda dependem muito uns dos outros para a troca de dados e mensagens. Dessa forma, caso a comunicação entre microsserviços falhe, ou até mesmo venha a tomar um tempo considerável para ser realizada, o funcionamento dos microsserviços pode ser comprometido, gerando falhas visíveis ao usuário final.

Ainda analisando a complexidade de organização advinda da arquitetura de microsserviços, dificulta-se a análise de erros e gargalos nos microsserviços em ambiente produtivo, que é o ambiente onde o produto utilizado pelo usuário final se encontra, também chamado de produção, visto que o software não se encontra concentrado em uma única estrutura e código para se realizar a depuração, mas em inúmeros serviços separados, independentes e por muitas vezes construídos em linguagens de programação diferentes, tornando-se necessário esforço e capital humano para realizar o monitoramento e depuração dos serviços em produção.

Existem duas estratégias principais para organizar os arquivos de código-fonte: a primeira, a estratégia de MONO REPOSITÓRIO, e a segunda, a estratégia de POLI REPOSITÓRIO. A Figura 13.7 ilustra a organização dos artefatos dos microsserviços

em um único repositório, em que os desenvolvedores organizam seus artefatos em uma estrutura de pastas, podendo o repositório conter uma pasta por projeto.

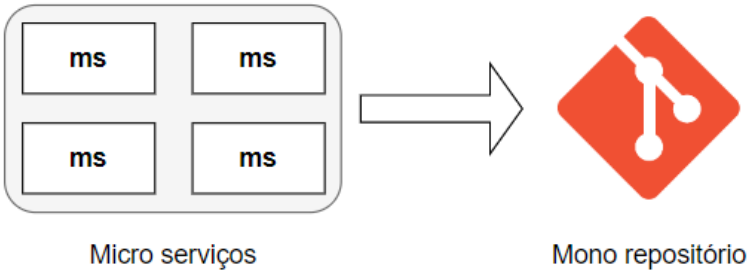


Figura 13.7: Estrutura mono repositório de microsserviços

Pode-se tomar como exemplo a estrutura de pastas ilustrada na Figura 13.8, que representa a organização dos microsserviços ilustrados na Figura 13.4: microsserviços de seleção de produtos melhor avaliados e de busca:

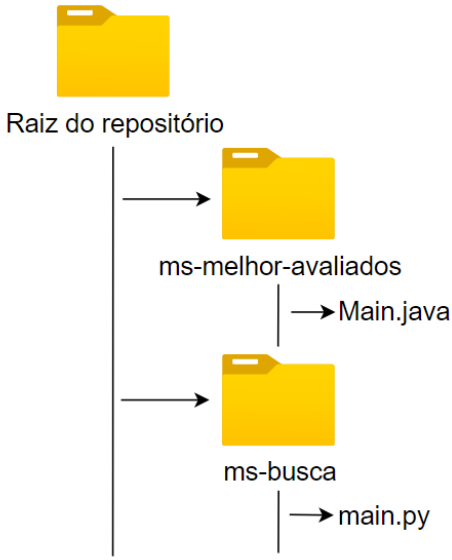


Figura 13.8: Estrutura das pastas de microsserviços no mono repositório

Perceba que, na Figura 13.8, cada microsserviço possui sua própria pasta no repositório, onde os desenvolvedores centralizam todos os artefatos necessários à execução independente dos serviços.

As principais vantagens de trabalhar com uma estrutura de MONOREPÓS referem-se à centralização e organização dos artefatos em um único repositório, o que facilita atividades de desenvolvimento, governança etc. Além disso, pode-se criar cenários de testes, esteiras de desenvolvimento e entrega contínua (CI/CD) e a implantação em ambiente produtivo de todos os serviços do repositório, ou, de maneira mais complexa, criar todas estas esteiras para os serviços de forma separada.

Entretanto, utilizando a estrutura de MONO REPOSITÓRIO, torna-se fácil para o desenvolvedor quebrar a premissa de desacoplamento entre os elementos da arquitetura de microsserviços, pois todos os códigos-fonte são de fácil acesso tanto para o desenvolvedor quanto para o serviço que ele está desenvolvendo. Além disso, caso a esteira de implantação em produção seja a mesma para todos os serviços, e a implantação ocorra com todos os microsserviços ao mesmo tempo, não há muita diferença entre a utilização desta arquitetura e o uso da arquitetura monolítica.

Dessa forma, a estrutura de MONOREPÓS torna-se ideal para projetos com poucos microsserviços e com uma esteira de implantação em ambiente produtivo, seja segregada para cada microsserviço ou para um conjunto deles. Ademais, o time de desenvolvimento deve ser maduro o suficiente para não criar dependência e acoplamento entre os microsserviços, visto que esse tipo de problema torna-se mais fácil com o uso de MONOREPÓS, por centralizar todos os artefatos em um único diretório raiz.

Outra forma de organizar os artefatos dos microsserviços é utilizar uma estrutura de POLI-REPOSITÓRIO. Nesta estrutura, cada microsserviço possui um único repositório que reúne todos os seus artefatos, incluindo código-fonte e outros arquivos necessários. A Figura 13.9 ilustra esta forma de organizar microsserviços utilizando POLI REPOSITÓRIOS, onde cada microsserviço possui seu próprio repositório para organização de artefatos, sendo ilustrados os microsserviços de carrinho, melhor avaliados e sistema de buscas em repositórios diferentes:

Quando utiliza-se a estrutura POLI REPOSITÓRIO para organizar os artefatos dos microsserviços, é possível perceber uma maior facilidade que um desenvolvedor possui para começar a trabalhar em um microsserviço, visto que ele trabalhará em um único repositório, sem a necessidade de conhecer os demais. Outra vantagem advinda da utilização de POLI REPOSITÓRIOS é a segregação das esteiras de desenvolvimento, testes e implantação dos microsserviços em ambiente produtivo. A Figura 13.10 ilustra a utilização de esteiras de implantação únicas em cada repositório, em que cada microsserviço se encontra em um repositório único, e cada repositório possui sua própria esteira de desenvolvimento, testes e implantação em ambiente produtivo:

Agora, considere um exemplo em que se deseja construir um software simples para uma livraria utilizando microsserviços. Para exemplificar, considere que a livraria deve possuir um sistema de controle de usuários e de controle de ordens



Figura 13.9: Estrutura poli repositório de microsserviços

(pedidos) de livros. A Figura 13.11 ilustra as funcionalidades a serem implementadas no sistema da livraria, sendo elas *ms-gerenciamento-usuario* e *ms-gerenciamento-pedido*, responsáveis pelo gerenciamento de cadastros de usuários e de pedidos de compra de livros, respectivamente. Para o exemplo, será implementada apenas a criação de usuários e de pedidos, utilizando microsserviços, comunicação via API e uma estrutura organizacional MONOREPÓS.

A Figura 13.12 ilustra um diagrama de modelos de dados do sistema de microsserviços da livraria. A classe USUÁRIO, representada à esquerda da Figura 13.12 representa um usuário com os seguintes atributos: *id_usuario*, nome (*nome*), e-mail (*email*), endereço (*endereco*) e telefone (*telefone*). Ademais, o outro modelo de dados criado refere-se ao pedido de livros por um usuário, representa um pedido com os seguintes atributos *id_pedido*, itens do pedido chamado *itens* e status do pedido chamado *status*. Por fim, os dois modelos de dados possuem um relacionamento 1:N entre si, em que um usuário possui N pedidos, mas um pedido só pode ser feito por um único usuário.

Antes de iniciar a codificação dos microsserviços da livraria, é considerada uma boa prática centralizar todas as variáveis constantes em um arquivo chamado *commons.py*, responsável por centralizar todas as configurações da API de comunicação

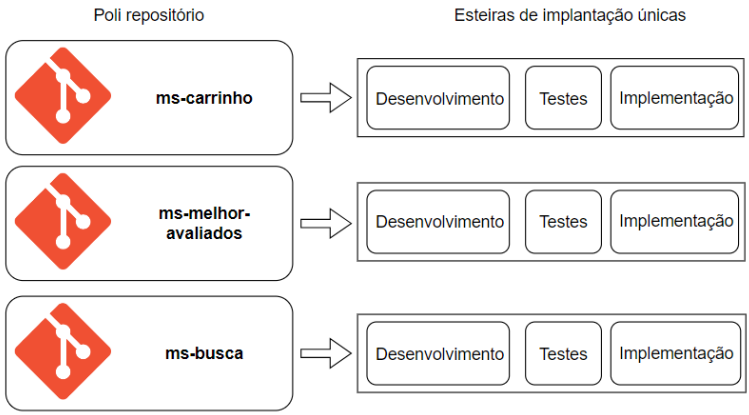


Figura 13.10: Estrutura poli repositório de microsserviços

entre os microsserviços da livraria. A Listagem 13.1 ilustra as configurações de variáveis comuns a todos os microsserviços, sendo elas: status HTTP de sucesso com o valor 200, status HTTP de criação com o valor 201, status http de falha com o valor 404, porta do microsserviço de gerenciamento de usuários com o valor 5000 e porta do microsserviço de gerenciamento de pedidos com o valor 5002. Por mais que esses sejam os códigos HTTP definidos na configuração da API, eles não são os únicos utilizados por aplicações web.

```
1 HTTP_STATUS_SUCESSO = 200
2 HTTP_STATUS_CRIADO = 201
3 HTTP_STATUS_FALHA = 404
4 PORTA_GERENCIAMENTO_USUARIO = 5000
5 PORTA_GERENCIAMENTO_PEDIDO = 5002
```

Listagem 13.1: Arquivo *api_config.py* da estrutura de microsserviços de livraria

A Listagem 13.2 ilustra a implementação do microsserviço de gerenciamento de usuários, representado como *ms-gerenciamento-usuario*. Neste microsserviço, utiliza-se a biblioteca *Flask* do *Python* para implementar uma função de criar usuário, chamada de *criar_usuario*, utilizando dados passados para o *endpoint* chamado */usuarios*, quando a necessidade é de fazer uma operação HTTP POST. Caso a operação seja realizada com sucesso, o usuário criado será adicionado à lista de usuários chamada de *usuarios*, e retorna um status HTTP de sucesso.

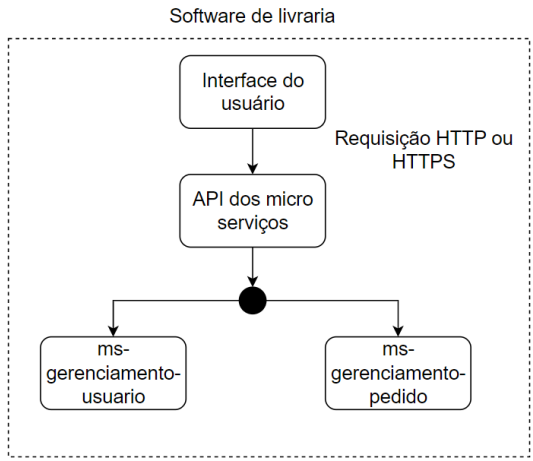


Figura 13.11: Diagrama ilustrativo dos microsserviços das funcionalidades do sistema da livraria

```python
from flask import Flask, jsonify, request
from commons.api_config import HTTP_STATUS_SUCESSO, HTTP_STATUS_CRIADO
    , HTTP_STATUS_FALHA, PORTA_GERENCIAMENTO_USUARIO

app = Flask(__name__)
usuarios = []

@app.route('/usuarios', methods=['POST'])
def criar_usuario():
    dados_usuario = request.get_json()
    usuario = {
        'user_id': len(usuarios) + 1,
        'name': dados_usuario['nome'],
        'email': dados_usuario['email'],
        'address': dados_usuario['endereco'],
        'phone_number': dados_usuario['telefone']
    }
    usuarios.append(usuario)

    return jsonify(usuario), HTTP_STATUS_SUCESSO

if __name__ == '__main__':
    app.run(port=PORTA_GERENCIAMENTO_USUARIO)
```

Listagem 13.2: Arquivo *gerenciamento_usuario.py* da estrutura de microsserviços de livraria

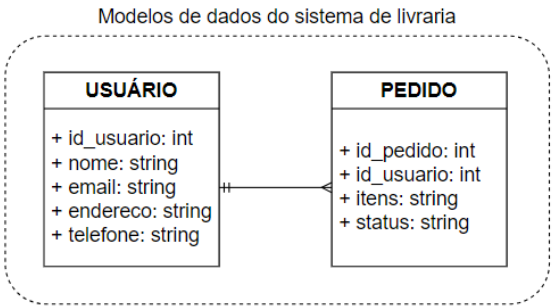


Figura 13.12: Diagrama de modelos de dados do sistema de livraria

A Listagem 13.3 ilustra a implementação do microsserviço de gerenciamento de pedidos, representado como *ms-gerenciamento-pedido* na Figura 13.11. Neste microsserviço, utiliza-se a biblioteca *Flask* do *Python* para implementar uma função de criar pedidos, chamada de *criar_pedido*, utilizando dados passados para o *endpoint* chamado */pedidos*, quando a necessidade é de fazer uma operação HTTP POST. Caso a operação seja realizada com sucesso, o pedido criado será adicionado à lista de pedidos chamada de *pedidos* e será retornado um status HTTP de criado.

```
from flask import Flask, jsonify, request
from commons.api_config import HTTP_STATUS_CRIADO, HTTP_STATUS_FALHA,
    HTTP_STATUS_SUCESSO, PORTA_GERENCIAMENTO_PEDIDO

app = Flask(__name__)
pedidos = []

@app.route('/pedidos', methods=['POST'])
def create_order():
    dados_pedido = request.get_json()
    order = {
        'id_pedido': dados_pedido['order_id'],
        'id_usuario': dados_pedido['user_id'],
        'itens': dados_pedido['items'],
        'status': dados_pedido['status']
    }
    pedidos.append(order)

    return jsonify(order), HTTP_STATUS_CREATED

if __name__ == '__main__':
    app.run(port=ORDER_MANAGEMENT_PORT)
```

Listagem 13.3: Arquivo *gerenciamento_pedido.py* da estrutura de microsserviços de livraria

A Listagem 13.4 ilustra a criação de usuários e pedidos à livraria, utilizando o módulo *requests* do *Python*. Nas Linhas 1 e 2, os pacotes necessários são importados, incluindo as variáveis constantes. As Linhas 4 a 9 demonstram a criação de um novo usuário, e na Linha 11 o módulo *request* realiza uma operação de POST no *endpoint* do serviço de gerenciamento de usuários para criar um novo usuário. A verificação do código de estado de resposta da operação POST é checada nas Linhas 13 a 17, informando os dados do usuário criado ou informando que houve erro durante a criação do mesmo. Seguindo a mesma lógica, nas Linhas 19 a 24, apresentam a criação de um novo pedido, na Linha 26 uma tentativa de realizar a operação POST no *endpoint* do serviço de gerenciamento de pedidos para criar um novo pedido. O código de status da operação é verificado nas Linhas 18 a 32, informando os dados a respeito do pedido criado ou informando a falha na criação do pedido.

```
1  import requests
2  from commons.api_config import HTTP_STATUS_CRIADO, HTTP_STATUS_SUCESSO
       , PORTA_GERENCIAMENTO_USUARIO, PORTA_GERENCIAMENTO_PEDIDO
3
4  dados_usuario = {
5      'nome': 'William',
6      'email': 'william@ufc',
7      'endereco': 'Rua rua',
8      'telefone': '123-444-555'
9  }
10
11 response = requests.post(f'http://localhost:{
       PORTA_GERENCIAMENTO_USUARIO}/users', json=dados_usuario)
12
13 if response.status_code == HTTP_STATUS_SUCESSO:
14     usuario = response.json()
15     print(f'Usuario criado! \n{usuário}')
16 else:
17     print('Erro durante a criação do usuário!')
18
19 dados_pedido = {
20     'id_pedido': 1,
21     'id_usuario': 1,
22     'itens': '5 books',
23     'status': 'payed'
24 }
25
26 response = requests.post(f'http://localhost:{
       PORTA_GERENCIAMENTO_PEDIDO}/orders', json=dados_pedido)
27
28 if response.status_code == HTTP_STATUS_CREATED:
29     pedido = response.json()
30     print(f'Pedido criado! \n{pedido}')
31 else:
32     print('Erro durante a criação do pedido!')
```

Listagem 13.4: Arquivo *app.py* da estrutura de microsserviços de livraria

Para iniciar a execução dos microsserviços de gerenciamento de usuários e pedidos, deve-se navegar para os diretórios onde seus *scripts* se encontram e executar os comandos descritos nas Listagens 13.5 e 13.6, para a execução dos serviços de gerenciamento de usuários e pedidos, respectivamente.

```
python gerenciamento_usuario.py
```

Listagem 13.5: Comando para executar o serviço de gerenciamento de usuários da livraria

Após a execução do código ilustrado na Listagem 13.5, um resultado similar ao que aparece na Figura 13.13 deve surgir.

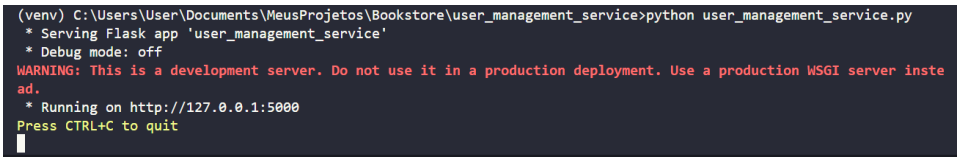


Figura 13.13: Resultado a execução da ativação do serviço de usuários

```
python gerenciamento_pedido.py
```

Listagem 13.6: Comando para executar o serviço de gerenciamento de usuários da livraria

Após a execução do código ilustrado na Listagem 13.6, um resultado similar ao que aparece na Figura 13.14 deve surgir.

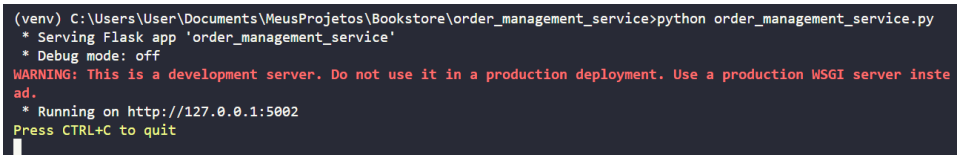


Figura 13.14: Resultado a execução da ativação do serviço de pedidos

A Listagem 13.7 ilustra o comando para executar o código-fonte da aplicação *app.py*, gerando um resultado similar ao que é ilustrado na Figura 13.15

```
python app.py
```

Listagem 13.7: Comando para executar o código da aplicação *app.py*

```
(venv) C:\Users\User\Documents\MeusProjetos\Bookstore\web_interface>python app.py
Usuário criado!
{'address': 'Rua rua', 'email': 'william@ufc', 'name': 'William', 'phone_number': '123-444-555', 'user_id': 2}
Pedido criado!
{'items': '5 books', 'order_id': 1, 'status': 'payed', 'user_id': 1}
```

Figura 13.15: Resultado a execução da aplicação web interagindo com microsserviços

Assim como na implementação de todas as outras arquiteturas, a arquitetura de microsserviços está sujeita à degradação com o passar do tempo. Desta forma, mesmo que a arquitetura de microsserviços seja definida durante a etapa de refinamento do requisito do sistema, as modificações de requisitos técnicos e de negócios que podem vir a ocorrer durante a etapa de desenvolvimento do sistema podem modificar a arquitetura inicialmente proposta. Dessa forma, é importante que o time de desenvolvimento se atente à organização dos elementos após estas modificações, para que as vantagens da utilização de microsserviços ainda possam ser mantidas sem comprometer o sistema.

13.5 Arquitetura orientada a mensagens

Na arquitetura orientada a mensagens, permanecem os papéis de serviços clientes e servidores, porém a, comunicação entre eles deixa de ser síncrona e baseada em requisições diretas de API. Nesse modelo, a interação ocorre por meio da troca de mensagens intermediadas por um broker de mensagens (VALENTE, 2020c). Conforme ilustrado na Figura 13.16, os serviços clientes publicam mensagens nesse intermediário, que, posteriormente, são consumidas pelos serviços servidores.Tanto clientes quanto servidores podem representar conjuntos de serviços, sem limitação quanto à quantidade de participantes de cada tipo.

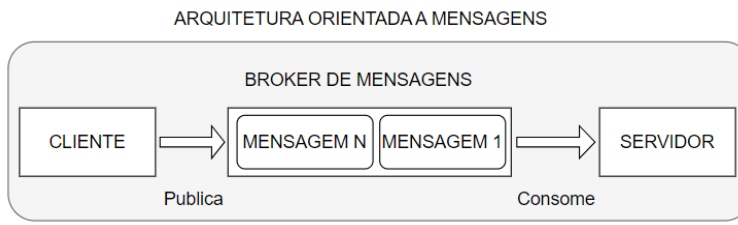


Figura 13.16: Diagrama ilustrativo da arquitetura orientada a mensagens

O uso do *broker* permite que a comunicação seja assíncrona, isto é, o envio e o recebimento das mensagens não precisam ocorrer simultaneamente. Dessa forma, clientes e servidores não dependem da disponibilidade imediata um do outro, já que as mensagens permanecem armazenadas no *broker* até serem consumidas, normalmente seguindo uma política de ordenação FIFO (*First In, First Out*) (VALENTE, 2020c).

Esse modelo de comunicação promove o desacoplamento entre os serviços, tanto no espaço quanto no tempo. O desacoplamento no espaço refere-se ao fato de que clientes não precisam conhecer detalhes de implementação ou quantidade dos servidores, e vice-versa, atuando apenas como produtores ou consumidores de mensagens. Já o desacoplamento no tempo decorre da independência de execução simultânea entre as partes, uma vez que o *broker* gerencia o armazenamento e a entrega das mensagens (VALENTE, 2020c). Por conta do desacoplamento no espaço, um único cliente pode publicar mensagens no *broker* enquanto múltiplos servidores podem consumir estas mensagens. Um servidor pode processar qualquer mensagem que for publicada na fila, ou possuir internamente um sistema de filtro para que apenas mensagens de interesse do servidor sejam consumidas (VALENTE, 2020c). A Figura 13.17 ilustra uma arquitetura orientada a mensagens onde existe um único cliente publicando diversas mensagens no *broker*, enquanto diversos servidores consomem estas mensagens, mas nem o cliente e nem os servidores sabem dos detalhes de implementação uns dos outros.

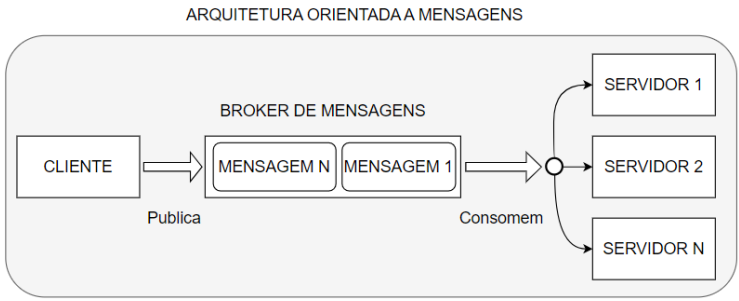


Figura 13.17: Arquitetura orientada a mensagens com múltiplos servidores

A comunicação na arquitetura orientada a mensagens é de 1 para 1, o que significa que as mensagens publicadas na fila são sempre consumidas por um único servidor, mesmo que hajam diversos servidores acoplados ao final do *broker*. Assim, torna-se difícil construir um esquema de tópicos de mensagens nesta arquitetura, onde uma mesma mensagem publicada poderia ser endereçada e consumida por diversos servidores.

Uma possível implementação da arquitetura orientada a mensagens seria na construção de um sistema que realizasse a comunicação de dados de empréstimos tomados por devedores utilizando fila de mensagens entre dois serviços, sendo um

cliente e o outro servidor. Além disso, a mensagem deve ser codificada em formato *base64* para garantir uma melhor eficiência na transmissão dos dados na fila. A Listagem 13.8 ilustra um arquivo no formato *JSON* contendo dados do devedor, a pessoa quem tomou um empréstimo, e do credor, quem cedeu o empréstimo.

```
{
    "nome": "William Bruno",
    "tipoPessoa": "pessoa fisica",
    "emprestimo": 1500.00,
    "informacoesCredor": {
        "razaoSocialCredor": "nome banco exemplo",
        "cnpjCredor": 12345678910
    }
}
```

Listagem 13.8: Arquivo *dados_emprestimo.json* do exemplo dos dados de empréstimos

Para construir a comunicação entre clientes e servidores por meio de uma fila de mensagens, pode-se utilizar o *broker* de mensagens *RabbitMQ*. A Listagem 13.9 ilustra a construção de uma classe chamada *fila_mensagem.python* responsável por configurar as conexões entre um *host* e um canal de comunicação com o *RabbitMQ* por meio das funções *estabelece_conexao_rabbitmq* e *cria_canal_rabbitmq*. Para construir uma fila de mensagens dentro do *broker* como ilustrado na Figura 13.16 utiliza-se a função *cria_fila_rabbitmq*. Para implementar o envio de mensagens à fila pelo cliente utiliza-se a função *envia_mensagem*. Para gerenciar o consumo das mensagens pelo servidor, utilizam-se as funções *configura_consumo* e *inicia_consumo*. Por fim, a classe implementa ainda uma função para que tanto o cliente quanto o servidor possam fechar as suas conexões com o *broker*.

```
import pika
from typing import Callable

class FilaMensagem:
    def __init__(self):
        self._connection = ``
        self._channel = ``

    def estabelece_conexao_rabbitmq(self, host: str):
        self._connection = pika.BlockingConnection(pika.
            ConnectionParameters(host))

    def cria_canal_rabbitmq(self):
        self._channel = self._connection.channel()

    def cria_fila_rabbitmq(self, queue_name: str):
        self._channel.queue_declare(queue=queue_name)

    def envia_mensagem(self, exchange=``, routing_key=``, body=``):
        self._channel.basic_publish(exchange=exchange, routing_key=
            routing_key, body=body)

    def configura_consumo(self, queue: str, call_back_function:
        Callable, auto_ack: bool):
        self._channel.basic_consume(queue=queue, on_message_callback=
            call_back_function, auto_ack=auto_ack)

    def inicia_consumo(self):
        self._channel.start_consuming()

    def fecha_canal(self):
        self._channel.close()
```

Listagem 13.9: Comando para executar o código da aplicação *app.py*

A Listagem 13.10 ilustra a implementação da classe *FilaMensagem*, apresentada na Listagem 13.9, para construir um serviço cliente responsável por enviar dados de empréstimos codificados em *base64* à fila de mensagens do *RabbitMQ*. A função principal é a *main*, é nela em que todo o processamento do código irá iniciar. A

função *carrega_json_para_dicionario* é responsável por carregar o arquivo *JSON* de empréstimos na memória e transformá-lo em uma variável dicionário. A função *codifica_string_para_base64* é responsável por codificar os dados de empréstimos como uma *string base64*, e as funções *envia_mensagem* e *fecha_canal* são responsáveis por enviar a mensagem à fila do *broker* e fechar a conexão com a fila, respectivamente.

```python
from modules.fila_mensagem import FilaMensagem
import json
import base64

def carrega_json_para_dicionario(caminho: str):
    with open(caminho) as arquivo:
        return json.load(arquivo)

def codifica_string_para_base64(string: str, codificacao='utf-8'):
    return base64.b64encode(string.encode(codificacao)).decode(
        codificacao)

def main():
    # Criando um objeto que implementa a classe MessageQueue
    message_queue = FilaMensagem()

    # Esabelecendo um canal de comunicação com o RabbitMQ
    message_queue.estabelece_conexao_rabbitmq(host='localhost')
    message_queue.cria_canal_rabbitmq()

    # Criando uma fila de mensagens no RabbitMQ
    nome_fila='exemplo_fila'
    message_queue.cria_fila_rabbimq(queue_name=nome_fila)

    # Carregando os dados de empréstimos
    dicionario_dados = carrega_json_para_dicionario('./data/
        dados_emprestimo.json')

    # Convertendo o dicionário para string
    dados_string = json.dumps(dicionario_dados)

    # Codificando os dados como string em uma mensagens base64
    dados_base64 = codifica_string_para_base64(string=dados_string,
        codificacao='utf-8')

    # Enviando a mensagem à fila do RabbitMQ
    message_queue.envia_mensagem(exchange=``, routing_key='
        exemplo_fila', body=dados_base64)
    print('mensagem enviada!')

    # Fechando a conexão com a fila de mensagens
    message_queue.fecha_canal()

if __name__ == "__main__":
    main()
```

Listagem 13.10: Arquivo *client.python* responsável por enviar mensagens à fila

A Listagem 13.11 ilustra a implementação da classe *FilaMensagem*, construída na Listagem 13.9, para a construção de um servidor responsável por consumir mensagens da fila do *broker*. A função principal é a *main*, é nela em que todo o processamento do código irá iniciar. A principal diferença entre o código do servidor e o do cliente se dá no final da função *main*, enquanto que no código do cliente a função *envia_mensagem* é responsável por enviar as mensagens à fila, no código do servidor as funções *configura_consumo* e *inicia_consumo* são responsáveis por configurar o consumo das mensagens e consumir as mensagens de forma contínua, respectivamente. Perceba que, na Listagem 13.11, existe uma função chamada *decodifica_mensagem_base64*, responsável por decodificar a mensagem em formato *base64* enviada pelo cliente para o formato de caracteres, de modo que os dados possam ser processados no lado do servidor. Além disso, a função *callback* é responsável por receber a mensagem da fila do *broker* e lidar da forma como for desejado.

```python
from modules.fila_mensagem import FilaMensagem
import base64

def decodifica_mensagem_base64(mensagem_base64):
    bytes_decodificados = base64.b64decode(mensagem_base64)
    mensagem_decodificadas = bytes_decodificados.decode('utf-8')
    return mensagem_decodificadas

# Função de callback para lidar com as mensagens consumidas
def callback(chanel, method, properties, body):
    base64_recebida = body.decode('utf-8')
    mensagem_decodificada = decodifica_mensagem_base64(base64_recebida
        )
    print(f'Recebida: {mensagem_decodificada}')

def main():
    fila_mensagem = FilaMensagem()

    # Estabelecendo conexão com o RabbitMQ
    fila_mensagem.estabelece_conexao_rabbitmq(host='localhost')
    fila_mensagem.cria_canal_rabbitmq()

    # Criando uma fila de mensagens
    nome_fila='exemplo_fila'
    fila_mensagem.cria_fila_rabbitmq(queue_name=nome_fila)

    # Configurando o consumo das mensagens
    fila_mensagem.configura_consumo(queue='exemplo_fila',
        call_back_function=callback, auto_ack=True)

    # Continuamente consumindo as mensagens da fila
    print('Aguardando mensagens. Para sair, tecle CTRL+C')
    try:
        fila_mensagem.start_consuming()
    except KeyboardInterrupt:
        print('Saindo...')

if __name__ == '__main__':
    main()
```

Listagem 13.11: Arquivo *receptor.python* responsável por consumir mensagens da fila

A Listagem 13.12 ilustra a forma de executar o script do servidor via terminal. Após a execução com sucesso do comando, o terminal irá ilustrar a mensagem "Aguardando mensagens. Para sair, tecle CTRL+C", e só irá ilustrar mensagens publicadas no *broker* quando as mesmas forem publicadas pelo cliente.

```
python receptor.py
```

Listagem 13.12: Comando para executar o script do servidor

A Listagem 13.13 ilustra a execução do script do cliente no terminal. Após a execução com sucesso do comando, o terminal exibirá a mensagem *mensagem enviada* e finalizará a execução do script, enviando os dados de empréstimos como uma *string base64* ao servidor.

```
python client.py
```

Listagem 13.13: Comando para executar o script do cliente

A Figura 13.18 ilustra o recebimento e decodificação dos dados de empréstimos enviados pelo cliente pela fila de mensagens.

```
(venv) C:\Users\User\Documents\MeusProjetos\software_engineering\architectures\message_oriented_architecture>python
 receiver.py
Aguardando mensagens. Para sair, tecle CTRL+C
Received: {"nome": "William Bruno", "tipoPessoa": "pessoa fisica", "emprestimo": 1500.0, "informacoesCredor": {"raz
aoSocialCredor": "nome banco exemplo", "cnpjCredor": 12345678910}}
```

Figura 13.18: Resultado do recebimento dos dados de empréstimos no servidor

Dessa forma, a arquitetura orientada a mensagens permite a construção de serviços desacoplados tanto no espaço, o que implica que um serviço cliente não necessita conhecer os detalhes de implementação nem como as mensagens são tratadas pelo servidor, e vice-versa, quanto no tempo, o que implica que os dois serviços não precisam estar ativos ao mesmo tempo para se comunicarem. Assim como nos microsserviços, a arquitetura de mensagens também pode se deteriorar durante a etapa de desenvolvimento, mas dificilmente os serviços apresentarão acoplamento forte, pois contam com um serviço de terceiros para lidar da comunicação assíncrona entre si.

13.6 Publicador-consumidor

Nesta arquitetura, tem-se três elementos principais na organização do sistema, sendo eles: serviços publicadores de eventos, um *broker* de eventos e serviços assinantes. Nesta organização, as mensagens, agora chamadas de eventos, são publicadas no *broker* de eventos por serviços publicadores e consumidas por serviços *assinantes*. Os serviços assinantes devem assinar seus eventos de interesse para serem notificados quando um evento for publicado no *broker* (VALENTE, 2020c).

A Figura 13.19 ilustra a organização dos elementos na arquitetura publicador-consumidor. Na Figura, os N publicadores publicam eventos em uma estrutura de eventos chamada *broker*, que são assinados por serviços chamados de assinantes.

A cada nova publicação de um evento no *broker*, o serviço que assinou o evento é notificado da chegada da mensagem, podendo então consumi-la de forma assíncrona (VALENTE, 2020c).

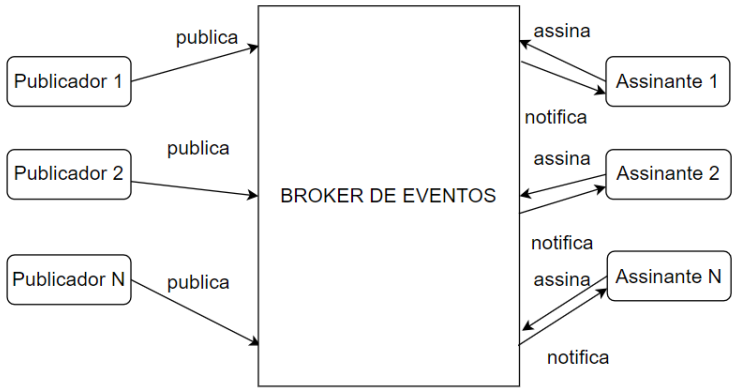


Figura 13.19: Diagrama ilustrativo de uma arquitetura publicador-consumidor

A arquitetura publicador-consumidor possui as mesmas vantagens da arquitetura orientada a mensagens, o que implica que o desacoplamento no tempo e no espaço se mantém nessa arquitetura. Dessa forma, os serviços publicadores e consumidores não precisam conhecer os detalhes de implementação uns dos outros, nem precisam estar ativos ao mesmo tempo.

No entanto, diferentemente da arquitetura orientada a mensagens, a arquitetura publicador-consumidor possui uma forma de comunicação entre os serviços própria. Enquanto que na arquitetura orientada a mensagens a comunicação ocorre ponto a ponto, ou seja: um cliente publica uma mensagem que é consumida só por um servidor, enquanto na arquitetura publicador consumidor a comunicação é de 1 publicador para N consumidores (VALENTE, 2020c).

Na comunicação do publicador-consumidor, os serviços assinantes são notificados quando há a publicação de algum evento de interesse no *broker* por serviços publicadores. Assim, os serviços assinantes não precisam consumir todos os eventos publicados, mas apenas os eventos específicos necessários ao seu processamento.

Além disso, é comum que publicadores e consumidores se comuniquem entre si por meio de tópicos no *broker*. Tópicos são categorias de eventos que atuam como centros de mensagens de determinada categoria (VALENTE, 2020c). A Figura 13.20 ilustra um diagrama esquemático no qual um serviço publicador envia eventos contendo informações de notícias do mercado financeiro e de preços de ativos a dois tópicos distintos. Enquanto isso, dois serviços assinantes consomem eventos desses tópicos separadamente e são notificados sempre que eventos são publicados.

Como exemplo de implementação da arquitetura ilustrada na Figura 13.20, pode-se construir um *script* Python contendo toda a implementação de um serviço

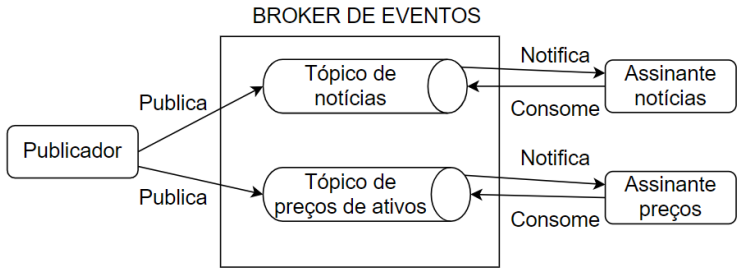


Figura 13.20: Diagrama ilustrativo de tópicos na arquitetura publicador-consumidor

publicador de eventos nos dois tópicos ilustrados na figura, e um outro *script* que implementa dois serviços assinantes que consomem eventos de apenas um tópico, ou de notícias do mercado financeiro ou de preços de ativos.

Dessa forma, a arquitetura publicador-consumidor é ideal para sistemas que necessitam que serviços se comuniquem de forma assíncrona, e que possam ser desenvolvidos de forma separada sem que a implantação de um serviço impacte no comportamento dos outros. Além disso, nesta arquitetura, os serviços assinantes não precisam estar ativos no momento da publicação dos eventos no *broker*, podendo estes receber os eventos quando voltarem a estar ativos.

A Listagem 13.14 ilustra a implementação do serviço publicador de eventos do exemplo citado. A Linha 1 realiza a importação da biblioteca necessária para implementar a arquitetura em *Python*, a Linha 4 cria um serviço publicador de eventos, enquanto que a Linha 5 conecta este publicador a um *broker* de eventos chamado "test.mosquitto.org" na Porta 1883. A Linha 8 inicia um *loop* infinito para publicar quantas mensagens o usuário deseje no tópico que ele escolher, até que ele decida não publicar mais mensagens. Das Linhas 9 a 16, o nome do tópico e a mensagem a ser publicada são coletados, e a mensagem é publicada no tópico escolhido. Na Linha 18, o publicador se desconecta do *broker* de eventos.

```
1  import paho.mqtt.client as mqtt
2
3  # Conectando com o broker de eventos
4  publicador = mqtt.Client()
5  publicador.connect('test.mosquitto.org', 1883)
6
7  # Publica mensagens em diferentes tópicos
8  while True:
9      topico = input(f'Digite o topico de interesse:
10     \n-noticias;\n-precos\n-sair')
11     if topico == 'sair':
12         break
13
14     mensagem = input(f'Digite a mensagem: ')
15
16     # Publica a mensagem para o tópico em específico
17     publicador.publish(topico, mensagem)
18
19 publicador.disconnect()
```

Listagem 13.14: Arquivo *publicador.py* da arquitetura publicador consumidor

A Listagem 13.15 ilustra a implementação de um serviço assinante do *broker* de eventos. A Linha 1 realiza a importação da biblioteca necessária para construir o serviço assinante de eventos, a Linha 4 define uma função chamada "trata_mensagem" que é responsável por realizar qualquer tratamento necessário da mensagem recebida, no caso a função apenas imprime a mensagem recebida. As Linhas 8 a 11 realizam a criação de um serviço que assina o tópico de notícias do *broker*. As Linhas 14 a 17 realizam a criação de um serviço que assina o tópico de preços de ativos. As Linhas 20 a 28 são responsáveis por manter os serviços consumindo seus respectivos tópicos até que o usuário do sistema interrompa o consumo via teclado, teclando CTRL+C. As Linhas 31 a 34 desconectam os serviços do *broker*.

```python
import paho.mqtt.client as mqtt

# Função que trata a mensagem recebida
def trata_mensagem(client, userdata, msg):
    print(f'Mensagem recebida: ', msg.payload.decode())

# Cria assinante do tópico de notícias do mercado financeiro
assinante_noticias = mqtt.Client()
assinante_noticias.connect("test.mosquitto.org", 1883)
assinante_noticias.subscribe('noticias')
assinante_noticias.on_message = trata_mensagem

# Cria assinante do tópico de preços de ativos
assinante_precos = mqtt.Client()
assinante_precos.connect("test.mosquitto.org", 1883)
assinante_precos.subscribe('precos')
assinante_precos.on_message = trata_mensagem

# Inicia um loop de consumo de eventos do broker para os assinantes
assinante_noticias.loop_start()
assinante_precos.loop_start()

# Mantém o script executando até ser interrompido
try:
    while True:
        pass
except KeyboardInterrupt:
    pass

# Disconecta ambos os assinantes do broker de eventos
assinante_noticias.loop_stop()
assinante_noticias.disconnect()
assinante_precos.loop_stop()
assinante_precos.disconnect()
```

Listagem 13.15: Arquivo *publicador.py* da arquitetura publicador consumidor

A Listagem 13.16 ilustra a execução do serviço assinante do *broker* de eventos em *Python*. É necessário executá-lo primeiro para que o serviço consome os eventos do *broker*.

```
python assinante.py
```

Listagem 13.16: Forma de executar o serviço assinante da arquitetura

A Listagem 13.17 ilustra a execução do serviço de publicador do *broker* de eventos em *Python*.

```
python publicador.py
```

Listagem 13.17: Forma de executar o serviço assinante da arquitetura

A Figura 13.21 ilustra o que seria exibido ao usuário após a execução do *script* do serviço de publicador. Percebe-se que o programa solicita o tópico e a mensagem a serem publicados no tópico continuamente ao usuário até que o mesmo solicite a saída do programa.

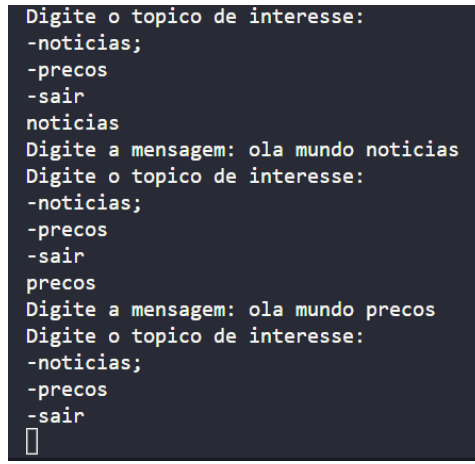


Figura 13.21: Execução do serviço publicador

A Figura 13.22 ilustra o que seria apresentado ao usuário após a execução do *script* do serviço assinante. Percebe-se que o programa continua ilustrando as mensagens recebidas pelo publicador a medida que são disponibilizadas pelo *broker*, até que o usuário tecle CTRL+C.

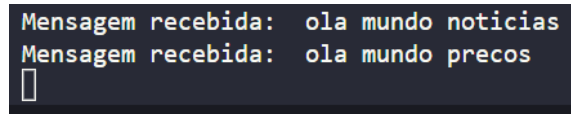


Figura 13.22: Execução do serviço assinante

13.7 Exemplos práticos

Os exemplos práticos a serem demonstrados consistem em um software de *streaming*. Neste sistema, um servidor *back-end* é capaz de enviar ou receber dados por meio de um canal de comunicação chamado *stream*. Além disso, é responsabilidade do sistema *back-end* gerenciar o número de *streams* criadas para comportar o fluxo de dados, de modo que os mesmos sejam salvos em uma base de dados de *streaming*. O objetivo é implementar o sistema em *Python*, utilizando diferentes padrões arquiteturais, a fim de gerar resultados preliminares e percepções sobre esses padrões. A Figura 13.23 ilustra a interação entre serviço de *streaming* e base de dados, na qual o serviço está hospedado em um servidor e pode realizar três ações principais: criar *streams*, enviar dados pela *stream* ou receber dados pela mesma.

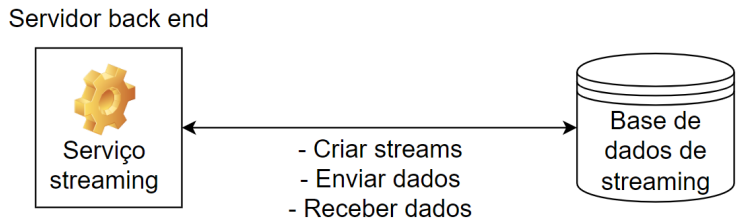


Figura 13.23: Sistema de *streaming*

Devido à alta complexidade de desenvolver um sistema como este por completo, esta seção abordará apenas uma simples implementação do sistema *back-end* de *streaming*, sem utilizar uma base de dados à parte, armazenando todos os dados no próprio servidor. Assim, esta seção direciona seu foco à organização dos elementos do software, bem como ao código e às discussões sobre a utilização das arquiteturas citadas neste capítulo na implementação do sistema de *streaming*.

13.8 Arquitetura monolítica

Um sistema construído em arquitetura monolítica possui todos os seus elementos concentrados em uma única unidade de código (Seção 13.2). A Listagem 13.18 ilustra a implementação de uma classe chamada "ServicoStreaming" em *Python*, responsável por representar o serviço de *streaming* construído com a arquitetura monolítica. A classe possui como funções: *criar_stream* e *adicionar_dados_na_stream*, *obter_dados_da_stream*. A função *criar_stream* possui a responsabilidade de criar uma *stream* de comunicação, enquanto que a função *adicionar_dados_na_stream* possui a responsabilidade de adicionar dados em uma *stream* específica de acordo com seu parâmetro de identificação e a função *obter_dados_da_stream* é responsável

por retornar dados de uma *stream* específica de acordo com seu parâmetro de identificação.

```
class ServicoStreaming:
    def __init__(self):
        self.streams = {}

    def criar_stream(self, id_stream):
        if id_stream not in self.streams:
            self.streams[id_stream] = []

    def adicionar_dados_na_stream(self, id_stream, dados):
        if id_stream in self.streams:
            self.streams[id_stream].append(dados)

    def obter_dados_da_stream(self, id_stream):
        if id_stream in self.streams:
            return self.streams[id_stream]
        else:
            return []
```

Listagem 13.18: Implementação de streaming utilizando classe na arquitetura monolítica

A Listagem 13.19 ilustra a usabilidade da classe *ServicoStreaming*. Na Listagem 13.19, é construído um objeto da classe *ServicoStreaming*, além da utilização da função *criar_stream* para criar uma *stream* com identificador "stream_1". Além disso, são adicionadas duas *strings* de dados à *stream* criada. Após isso, os dados da *stream* criada são retornados por meio da função *obter_dados_da_stream* e ilustrados ao usuário por meio da função *print* na Linha 6.

```
servico_streaming = ServicoStreaming()
servico_streaming.criar_stream("stream_1")
servico_streaming.adicionar_dados_na_stream("stream_1", "Dados 1")
servico_streaming.adicionar_dados_na_stream("stream_1", "Dados 2")
stream_data = servico_streaming.obter_dados_da_stream("stream_1")
print(stream_data)
```

Listagem 13.19: Implementação de streaming utilizando classe na arquitetura monolítica

Assim, com a construção do serviço de streaming utilizando uma arquitetura monolítica, é possível perceber que as principais vantagens de desenvolver um sistema dessa forma são as facilidades de desenvolver e testar o software, dado que tudo está sendo desenvolvido em uma única unidade, o que ainda é simples e possui poucas funcionalidades (INGALLS, 2021a).

No entanto, à medida que a aplicação monolítica cresce em número de linhas de código e de funcionalidades disponíveis, a complexidade de desenvolvê-la e testá-la aumenta exponencialmente, pois as funcionalidades e elementos do sistema tendem

a se tornar muito acoplados, dificultando atividades de modificação e criação de novos elementos na aplicação.

Além disso, quando uma aplicação monolítica se torna muito inflada de funcionalidades, torna-se exaustiva a etapa de implementação em um ambiente produtivo, visto que caso haja um pequeno erro em uma função, este erro irá impactar todo o software, pois todos os elementos estão condensados em uma única unidade. Dessa forma, o esforço e o capital humano necessários para desenvolver e manter uma aplicação monolítica crescerão significativamente à medida que novas funcionalidades são adicionadas à aplicação (VALENTE, 2020c).

13.9 Arquitetura cliente-servidor

A Listagem 13.20 ilustra a construção de uma classe chamada *ServicoStreaming* para implementar o recebimento e tratamento de mensagens do serviço de streaming em uma arquitetura cliente servidor utilizando o pacote *socket* (PYTHON, 2023). A classe possui apenas dois métodos, *__init__* e *iniciar*. O método *__init__* é responsável por instanciar o objeto da classe e inicializar os atributos *host*, *porta* e *streams*. O atributo *host* indica onde o serviço de *streaming* será hospedado, enquanto o atributo "porta" indica o endereço do *host* no qual as mensagens serão publicadas, e o atributo *streams* armazena todas as *streams* e seus respectivos dados.

Ainda, discutindo a respeito da Listagem 13.20, o método "iniciar" é responsável por configurar o acesso do servidor ao *host* e a sua porta especificada no método *__init__*. Além disso, o método "iniciar" é responsável por monitorar continuamente o acesso ao serviço de *streaming*, de modo a instanciar uma nova classe chamada "TratadorCliente", que será discutida mais adiante, para lidar com as mensagens enviadas pelos serviços clientes ao servidor.

```
import socket

class ServicoStreaming:
    def __init__(self, host, porta):
        self.host = host
        self.porta = porta
        self.streams = {}

    def iniciar(self):
        server_socket = socket.socket(socket.AF_INET, socket.
            SOCK_STREAM)
        server_socket.bind((self.host, self.porta))
        server_socket.listen(1)
        print("Servidor monitorando {}:{}".format(self.host, self.
            porta))

        while True:
            socket_cliente, endereco_cliente = server_socket.accept()
            print("Cliente conectado:", endereco_cliente)
            tratador_cliente = TratadorCliente(socket_cliente, self.
                streams)
            tratador_cliente.start()
```

Listagem 13.20: Construção de classe de serviço de *streaming* no arquivo *server.py*

A Listagem 13.21 ilustra a construção de uma classe chamada "TratadorCliente", responsável por tratar todas as mensagens enviadas ao servidor em uma *thread* a parte, de forma a garantir o processamento assíncrono da informação. A classe possui apenas dois métodos, um chamado *__init__* e o outro chamado "executar".

O método *__init__* é responsável por instanciar um objeto da classe "TratadorCliente", recebendo como parâmetro o *socket* do cliente que se conectou ao servidor, podendo, assim, tratar os dados enviados por ele. Ademais, o método "executar" é responsável por monitorar a porta e o *host* do servidor, de modo a capturar os dados enviados pelos clientes. Por fim, a função "executar" é responsável por adicionar os dados enviados pelos clientes à *stream* desejada, utilizando, para isso, um parâmetro chamado "id_stream".

```
class TratadorCliente(threading.Thread):
    def __init__(self, socket_cliente, streams):
        threading.Thread.__init__(self)
        self.socket_cliente = socket_cliente
        self.streams = streams

    def executar(self):
        while True:
            dados = self.socket_cliente.recv(1024)
            if not dados:
                break

            id_stream, dados_stream = dados.decode().split(":")
            if id_stream not in self.streams:
                self.streams[id_stream] = []
            self.streams[id_stream].append(dados_stream)

            print("Dados recebidos para a stream:", id_stream)
            print("Dados:", dados_stream)

        self.socket_cliente.close()print(stream_data)
```

Listagem 13.21: Implementação de streaming utilizando classe na arquitetura monolítica

A Listagem 13.22 ilustra a utilização da classe "ServicoStreaming" implementada na Listagem 13.20. Para que a classe seja utilizada, basta instanciar um objeto que a represente, definindo o *host* e a porta a serem utilizados para a comunicação com o servidor. Após isso, basta utilizar a função "iniciar" do objeto instanciado, e o servidor passará a monitorar o *host* e a sua porta, de modo a tratar as mensagens enviadas por um cliente em uma *thread* específica para isto (IBM, 2023b).

```
servidor = ServicoStreaming("localhost", 8000)
servidor.iniciar()
```

Listagem 13.22: Execução do servidor de streaming no arquivo *server.py*

Adicionalmente ao código do servidor, construído no arquivo *server.py*, é necessário implementar o código do cliente, que será escrito no arquivo *client.py*. A Listagem 13.23 ilustra o início da construção do serviço cliente do sistema de *streaming*. A classe do cliente construída chama-se "ClienteStreaming" e possui dois métodos: o primeiro chama-se *__init__* e o outro chama-se "enviar_dados".

Na Listagem 13.23, o método *__init__* é responsável por instanciar um objeto da classe "ClienteStreaming", configurando o *host* e a porta que o cliente utilizará para enviar mensagens. Ademais, o método "enviar_dados" é responsável por enviar dados para uma determinada *stream*, conforme o parâmetro de identificação "id_stream". Caso a *stream* ainda não tenha sido criada no servidor, o servidor a criará e armazenará seus dados, ou adicionará-os a uma *stream* com o mesmo parâmetro "id_stream" se já tiver sido criada.

```
import socket

class ClienteStreaming:
    def __init__(self, host, porta):
        self.host = host
        self.porta = porta

    def enviar_dados(self, id_stream, dados):
        socket_cliente = socket.socket(socket.AF_INET, socket.
            SOCK_STREAM)
        socket_cliente.connect((self.host, self.porta))
        mensagem = "{}:{}".format(id_stream, dados)
        socket_cliente.sendall(mensagem.encode())
        socket_cliente.close()
servidor.iniciar()
```

Listagem 13.23: Construção de classe que implementa a usabilidade do servidor de *streaming* por um cliente no arquivo *client.py*

A Listagem 13.24 ilustra a utilização da classe "ClienteStreaming", instanciando um objeto e chamando o método "enviar_dados" definido nela. Ao observar a Listagem, percebe-se que o método "enviar_dados" é utilizado para adicionar duas *strings* distintas ao *stream* "stream_1". Assim, o cliente consegue enviar quantos dados forem necessários ao servidor, ficando como responsabilidade do servidor tratar os dados de forma desacoplada do cliente.

```
cliente = ClienteStreaming("localhost", 8000)
cliente.enviar_dados("stream_1", "Dados 1")
cliente.enviar_dados("stream_1", "Dados 2")
```

Listagem 13.24: Utilização da classe ClienteStreaming no arquivo *client.py*

Após implementar o serviço de *streaming* utilizando a arquitetura cliente-servidor, percebe-se que a principal vantagem de desenvolver o serviço dessa maneira foi a capacidade de desacoplar o desenvolvimento da aplicação em dois serviços distintos. Dessa forma, um time de desenvolvimento consegue construir novas funcionalidades em clientes e servidores de forma independente, sem que o desenvolvimento em um serviço impacte o funcionamento do outro.

Além disso, devido ao desacoplamento entre cliente e servidor, todas as etapas de testes e de implantação em ambiente produtivo podem ocorrer de forma independente para cada serviço. Assim, os impactos gerados no servidor devido a um erro em uma funcionalidade nova do cliente são mitigados e vice-versa. Outrossim, torna-se mais fácil para o time de desenvolvimento testar novas funcionalidades em cada serviço, visto que precisará testar apenas um dos sistemas, seja o cliente ou o servidor.

Como último ponto a ser destacado relacionado ao desacoplamento do sistema, tem-se a separação das esteiras de implantação em ambiente produtivo do software, o que significa que, em comparação com o monólito, não é necessário substituir toda a aplicação em ambiente produtivo quando uma nova funcionalidade for desenvolvida.

Assim, a aplicação do cliente e do servidor pode ser implantada em produção de forma separada, mitigando impactos e riscos de erros no ambiente produtivo.

Ademais, utilizar a arquitetura cliente-servidor permite que o software se torne escalável quanto ao processamento e à memória do sistema. Ao utilizar esta arquitetura, o time de desenvolvimento pode desacoplar cliente e servidor para hospedá-los em um conjunto de máquinas separadas, de forma a escalar a quantidade de máquinas e suas respectivas capacidades de processamento e de memória. Esta abordagem torna-se muito útil para garantir que o software consiga acompanhar.

Entretanto, a utilização da arquitetura cliente-servidor acrescenta complexidade adicional ao sistema, pois os desenvolvedores precisam cuidar dos detalhes de comunicação e integração entre clientes e servidores. Portanto, o time de desenvolvimento deve ter conhecimento de redes, protocolos de comunicação e de formas de garantir uma transmissão de dados eficaz e segura por meio dos serviços.

13.10 Arquitetura Orientada a Mensagens

A Listagem 13.25 ilustra a construção de uma classe chamada “ServicoStreaming” em um arquivo chamado *server.py*, a qual implementa a lógica de envio e recebimento de mensagens de *streaming* utilizando a arquitetura orientada a mensagens. A classe é implementada utilizando o pacote “pika” do *Python* (PIKA, 2017) e utilizando o *broker* RabbitMQ (RABBITMQ, 2023). A classe possui quatro métodos: *__init__*, “enviar_dados”, “processar_dados” e ”iniciar_consumo”.

Ainda sobre a Listagem 13.25, o método *__init__* é responsável por instanciar um objeto da classe em questão, configurando o *host* do *broker* de mensagens e o nome da fila a ser criada. Ademais, o método “enviar_dados” é utilizado pelo publicador para enviar uma mensagem específica ao *broker*, composta pelo identificador da stream e pelos dados da mensagem. Outrossim, o método “processar_dados” é responsável por tratar as mensagens publicadas na fila do *broker* pelo publicador. Por fim, o método “iniciar_consumo” trata de consumir as mensagens do *broker* pelo consumidor, utilizando a função “processar_dados” para tratá-las.

```
import pika

class ServicoStreaming:
  def __init__(self):
    self.conexao = pika.BlockingConnection(pika.ConnectionParameters('
        localhost'))
    self.canal = self.conexao.canal()
    self.canal.queue_declare(queue='fila_stream')

  def enviar_dados(self, id_stream, dados):
    mensagem = "{}:{}".format(id_stream, dados)
    self.canal.basic_publish(exchange='', routing_key='fila_stream',
        body=mensagem)
    print("Dados enviados ao stream:", id_stream)
    print("Dados:", dados)

  def processar_dados(self, ch, method, properties, body):
    id_stream, dados_stream = body.decode().split(":")
    print("Dados recebidos na stream:", id_stream)
    print("Dados:", dados_stream)

  def iniciar_consumo(self):
    self.canal.basic_consume(queue='fila_stream', on_message_callback=
        self.processar_dados,auto_ack=True)
    print('Consumindo dados da stream...')
    self.canal.start_consuming()
```

Listagem 13.25: Construção da classe ServicoStreaming no arquivo *server.py* utilizando arquitetura orientada a mensagens

A Listagem 13.26 ilustra a utilização da classe "ServicoStreaming" construída na Listagem 13.25, ainda no mesmo arquivo *server.py*. O objeto instanciado da classe, chamado de "servico_streaming', envia duas *strings* de dados diferentes para a mesma fila de mensagens, e para a mesma *stream* utilizando a função "enviar_dados". Por fim, o mesmo serviço de *streaming* consume as mensagens publicadas no *broker* utilizando a função "iniciar_consumo".

```
servico_streaming = ServicoStreaming()
servico_streaming.enviar_dados("stream_1", "Dados 1")
servico_streaming.enviar_dados("stream_1", "Dados 2")
servico_streaming.iniciar_consumo()
```

Listagem 13.26: Utilização classe "ServicoStreaming" no arquivo "*server.py*" utilizando arquitetura orientada a mensagens

Após implementar o serviço de *streaming* utilizando uma arquitetura orientada a mensagens, é possível perceber semelhanças na forma como a classe de serviço de *streaming* foi implementada na Listagem 13.26 em relação à arquitetura monolítica. Na Listagem, é possível perceber que o serviço foi instanciado no arquivo *server.py* e, ainda nesse mesmo arquivo, os dados foram enviados e consumidos pelo mesmo objeto. O exemplo foi construído desta forma para manter a simplicidade, sem deixar de evidenciar a semelhança com o sistema monolítico já discutido, cuja principal vantagem é a facilidade de desenvolver sistemas simples com esta arquitetura.

No entanto, é importante ressaltar que esta prática não é recomendada para a construção de sistemas complexos e de alta necessidade de escalabilidade, visto que sua implementação por meio de um sistema monolítico tornaria muito complexa a manutenção por um time de desenvolvimento. Portanto, o meio correto de se desenvolver com a arquitetura orientada a mensagens seria utilizando a separação entre serviços publicadores e consumidores de mensagens, mas foi desenvolvido de uma forma que remonta ao monólito para manter a simplicidade do exemplo.

As principais vantagens de desenvolver o sistema de *streaming* com a arquitetura orientada a mensagens são: redução do acoplamento entre os serviços, escalabilidade e tolerância a falhas. Quando se desenvolve um sistema com arquitetura orientada a mensagens, utiliza-se um *broker* de mensagens, um terceiro elemento entre os serviços publicadores e os consumidores. Assim, como a comunicação entre os serviços é intermediada por este terceiro, reduz-se o acoplamento entre os serviços e reforça-se o desacoplamento no espaço, o que dá mais liberdade aos times para aprimorar e desenvolver, de forma independente, as funcionalidades dos serviços.

Além disso, aproveitando-se da perda de acoplamento de serviços, a utilização de um *broker* permite que a quantidade de serviços, tanto publicadores quanto consumidores, seja escalada com mais facilidade para atender à demanda de uso do sistema, como já ilustrado na Figura 13.16. Como exemplo, pode-se supor a necessidade de adicionar novos serviços de publicação durante um período de feriado, em que a demanda pelos canais de *streaming* aumenta. Assim, a quantidade de serviços pode ser escalada com mais facilidade para atender à demanda de negócios.

A arquitetura orientada a mensagens permite a construção de sistemas tolerantes a falhas, por meio da comunicação assíncrona e da persistência de mensagens no *broker*. Dessa forma, a arquitetura reforça o desacoplamento no tempo do sistema, permitindo que a comunicação entre os serviços ocorra mesmo que ambos não estejam ativos simultaneamente. Assim, as mensagens publicadas no *broker* podem

ser persistidas, caso os consumidores não estejam ativos no momento, de modo que consumam as mensagens salvas quando voltarem a estar ativos.

Mesmo com todas as vantagens, a arquitetura orientada a mensagens ainda apresenta algumas desvantagens. Como desvantagem relacionada ao desenvolvimento do sistema por um time, ressalta-se a complexidade de construí-lo dessa maneira, pois impõe ao time capacitar-se para desenvolver sistemas por meio de comunicação assíncrona e de *brokers* de mensagens. Além disso, a comunicação assíncrona entre os serviços pode aumentar a latência na troca de mensagens, o que pode ser um fator crítico, dependendo dos requisitos e do tipo de sistema a ser desenvolvido.

13.11 Arquitetura publicador consumidor

A Listagem 13.27 ilustra a construção de uma classe que implementa a lógica do serviço de *streaming* utilizando a arquitetura de publicador-consumidor. A classe implementada utiliza o Redis (REDIS, 2023a) como *broker* de eventos. A classe possui apenas três métodos: o primeiro chama-se "*__init__*, o segundo chama-se "enviar_dados" e o terceiro chama-se "iniciar_consumo".

Ainda na Listagem 13.27, o método *__init__* é responsável por instanciar o objeto da classe "ServicoStreaming", configurando o *host* do Redis e a porta que o *broker* utilizará. Além disso, o método "enviar_dados" é responsável por receber, como parâmetros, o identificador da *stream* a enviar os dados e as informações em si, e por enviá-las ao tópico "canal_stream". Por fim, o método "iniciar_consumo" é responsável por consumir os eventos de um determinado tópico à medida que são publicados no *broker*.

```
import redis

class ServicoStreaming:
    def __init__(self):
        self.cliente_redis = redis.Redis(host='localhost', port=6379)
        self.pubsub = self.cliente_redis.pubsub()
        self.pubsub.subscribe('canal_stream')

    def enviar_dados(self, id_stream, dados):
        mensagem = "{}:{}".format(id_stream, dados)
        self.cliente_redis.publish('canal_stream', mensagem)
        print("Dados enviados para a stream:", id_stream)
        print("Dados:", dados)

    def iniciar_consumo(self):
        for mensagem in self.pubsub.listen():
            if mensagem['type'] == 'message':
                id_stream, dados_stream = mensagem['data'].decode().
                    split(":")
                print("Dados recebidos da stream:", id_stream)
                print("Dados:", dados_stream)
```

Listagem 13.27: Construção da classe ServicoStreaming em um arquivo chamado *server.py*

A Listagem 13.28 ilustra a utilização da classe "ServicoStreaming" descrita na Listagem 13.27. Na listagem em questão, a classe é instanciada como um objeto que assina um tópico chamado "canal_stream", e, em seguida, o objeto envia duas *strings* de dados para a *stream* "stream_1". Após isso, o mesmo serviço inicia o consumo dos eventos publicados no *broker*, sendo notificado sempre que um novo evento é publicado no tópico em que assinou.

```
servico_streaming = ServicoStreaming()
servico_streaming.enviar_dados("stream_1", "Dados 1")
servico_streaming.enviar_dados("stream_1", "Dados 2")
servico_streaming.iniciar_consumo()
```

Listagem 13.28: Utilização da classe ServicoStreaming em um arquivo chamado *server.py*

É importante perceber que a construção e a utilização da classe de serviço de *streaming*, utilizando a arquitetura publicador-consumidor, na Listagem 13.27 e na Listagem 13.28 seguem a mesma lógica da implementação com arquitetura orientada a mensagens. Ou seja, o exemplo foi construído em um único arquivo, que remonta ao monólito, mas apenas para fins didáticos. Assim, se houver necessidade de construir um sistema mais complexo, é preciso separar o código em serviços publicadores e assinantes.

As vantagens de utilizar a arquitetura publicador-consumidor são as mesmas de utilizar uma arquitetura orientada a mensagens: perda de acoplamento entre

serviços, separação de responsabilidades, escalabilidade, etc. No entanto, a principal vantagem de desenvolver um sistema com uma arquitetura publicador-consumidor é, como se pode observar nas listagens, um maior grau de desacoplamento entre os serviços publicadores e os consumidores.

Esta maior perda de acoplamento entre os serviços ocorre porque a comunicação não se dá mais de forma direta por meio de uma fila de mensagens, mas sim por meio de um *broker* de eventos, responsável por notificar todos os serviços assinantes sobre a presença daquele evento no *broker*, formando uma comunicação de 1 publicador para N assinantes. Assim, a comunicação ocorre de forma ainda mais desacoplada no espaço e no tempo, visto que nem os publicadores nem os consumidores precisam conhecer os detalhes de implementação uns dos outros, nem precisam estar ativos ao mesmo tempo.

As principais desvantagens da utilização da arquitetura publicador-consumidor são a complexidade e a confiança nas mensagens. Ao adicionar um sistema de *broker* de eventos, aumenta-se a complexidade do software construído, exigindo capacitação do time de desenvolvimento e cuidados adicionais no processo de desenvolvimento. Por fim, dependendo do *broker* utilizado, pode ou não ser possível persistir eventos, o que leva à possibilidade de utilizar outro serviço para persistir os eventos publicados no *broker*.

Após implementar o sistema de *streaming* utilizando as arquiteturas de monólito, cliente-servidor, orientadas a mensagens e publicador-consumidor, torna-se possível entender em que contextos uma arquitetura é melhor do que as demais e quais são suas desvantagens. Por exemplo, a arquitetura monolítica é ideal para sistemas simples, pois é muito fácil desenvolver, testar e implementar uma aplicação monolítica. No entanto, à medida que a aplicação se torna mais complexa, torna-se difícil manter o sistema e escalar a aplicação, pois tudo o que é desenvolvido é feito em uma única unidade.

Ao utilizar uma arquitetura cliente-servidor, o sistema torna-se mais desacoplado e escalável, pois os dois serviços são desenvolvidos e implementados de forma independente. Entretanto, pelo fato de que, para manter uma comunicação entre os serviços, torna-se necessário configurar a comunicação dos serviços, além de enviar, processar e receber requisições por meio de uma rede, pode haver um crescimento no tempo de resposta de uma determinada funcionalidade.

Utilizar a arquitetura orientada a mensagens permite que os serviços publicadores e consumidores de mensagens sejam desacoplados no tempo e no espaço, o que significa que os serviços não precisam estar ativos ao mesmo tempo para que a comunicação ocorra, e que não precisam conhecer os detalhes de implementação uns dos outros. No entanto, dada a utilização de um *broker* de mensagens e a necessidade de integrar os serviços de forma efetiva, aumenta-se a complexidade de manter esse sistema pelo time de desenvolvimento.

Utilizar a arquitetura publicador-consumidor traz quase as mesmas vantagens da arquitetura orientada a mensagens, mas com foco na comunicação de 1 publicador para N consumidores. Assim, a comunicação entre os serviços fica sob

responsabilidade do *broker* de eventos, que distribui os eventos publicados a todos os serviços assinantes do tópico em questão. Entretanto, assim como na arquitetura orientada a mensagens, a complexidade de configurar e manter um sistema com esta arquitetura é elevada e requer capacitação dos desenvolvedores.

Capítulo 14

Ferramentas para Web

Izaias Machado Pessoa Neto (UFC)

Fischer Ferreira (UNIFEI)

Este capítulo tem como objetivo apresentar algumas das principais ferramentas web e explorar suas funcionalidades. Inicialmente, é definido o escopo das ferramentas que apoiam o desenvolvimento web (Seção 14.1). Desta forma, nas seções seguintes são apresentados os contextos das principais ferramentas e também um exemplo prático. As demais seções que compõem o capítulo discutem: controle de versão com Git (Seção 14.2), compartilhamento e gerenciamento de código com GitHub (Seção 14.3), virtualização de aplicações com Docker (Seção 14.4) e, por fim, gerenciamento de identidade e controle de acesso com WSO2 Identity Server (Seção 14.5).

14.1 Ferramentas que apoiam o desenvolvimento Web

Uma característica importante da web é o desacoplamento entre o *front-end* e o *back-end*. Ferramentas web são utilizadas para construir tanto aplicações *front-ends*, quanto *back-ends*. O *front-end* abrange os sistemas de *software* executados no navegador do cliente. Já as tecnologias executadas no lado do servidor da aplicação são chamadas de *back-end*. O *front-end* faz requisições por meio de chamadas HTTP e, em resposta, o servidor (*back-end*) retorna o resultado do processamento (MOZILLA, 2021).

Essa separação entre *front-end* e *back-end* permite que cada camada seja desenvolvida de forma independente, facilitando a colaboração entre equipes especializadas em suas respectivas áreas. Esse desacoplamento traz vantagens, como flexibilidade na escolha das tecnologias mais adequadas a cada camada, o que permite o uso das melhores soluções disponíveis.

Como há necessidades comuns na construção de aplicações web, algumas ferramentas podem ser empregadas independentemente do domínio da aplicação. Algumas dessas ferramentas podem oferecer funcionalidades essenciais, como controle de versão, compartilhamento de código, documentação, portabilidade da aplicação e gerenciamento de identidade.

Essas são algumas das funcionalidades muito importantes no contexto do desenvolvimento web. Ao utilizar essas ferramentas, os desenvolvedores podem aprimorar os processos de desenvolvimento, colaboração e implantação de aplicações web, independentemente do tipo de projeto em que estejam trabalhando.

Para apoiar a comunicação e a colaboração eficazes entre as equipes, existem ferramentas, como o Git[1] e o GitHub[2]. O Git é um sistema de controle de versão distribuído que permite rastrear as alterações no código-fonte ao longo do tempo. Ele também permite que os desenvolvedores trabalhem em paralelo na construção de novas funcionalidades e na correção de *bugs* e, posteriormente, integrem suas alterações ao código principal. Já o GitHub, por sua vez, é uma plataforma de

[1] https://git-scm.com/
[2] https://github.com

hospedagem de código que utiliza o Git como base. Ele oferece recursos adicionais, como o gerenciamento de *bugs* e de solicitações de novas funcionalidades, e também facilita a revisão e a discussão do código entre os membros da equipe. Isso possibilita a colaboração em projetos, a revisão de alterações propostas e a resolução de problemas de forma centralizada.

Além do controle de versão e de comunicação, o desenvolvimento e a implantação de aplicações web podem se beneficiar do uso do Docker. Essa ferramenta permite a criação de ambientes consistentes e portáteis, chamados *containers*, que encapsulam as dependências e configurações necessárias para a execução de uma aplicação. Com o Docker, é possível garantir que a aplicação funcione de forma consistente em diferentes máquinas e sistemas operacionais, facilitando a colaboração entre equipes e simplificando o processo de implantação do código (DOCKER, 2020).

Por fim, temos ferramentas que auxiliam no controle de acesso. O WSO2 Identity Server, por exemplo, fornece recursos de segurança, como autenticação e autorização, além de oferecer suporte a protocolos de login único. Com essa ferramenta, é possível garantir a proteção das aplicações e o controle de acesso adequado aos recursos. É importante ressaltar que o uso de uma ferramenta centralizada de login permite que o mesmo login seja utilizado em todas as aplicações de uma organização.

14.2 Controle de versão com Git

Os SCV (Sistemas de Controle de Versão) têm a finalidade de gerenciar as alterações realizadas ao longo das várias versões de um projeto, permitindo, caso necessário, reverter mudanças feitas em um ou mais arquivos (CHACON; STRAUB, 2014). Essa capacidade de controle é obtida por meio de um banco de dados de versão, que registra todas as modificações realizadas nos arquivos ao longo do tempo. Para o Git, após cada *commit* é criada uma nova versão, que tem um identificador único. Nesse sentido, é possível visualizar as versões por meio de um histórico, o que permite rastrear as modificações realizadas. Além disso, o Git facilita a colaboração entre os membros da equipe, permitindo que cada pessoa trabalhe em sua própria versão do projeto e, posteriormente, as versões desenvolvidas são combinadas. Essa abordagem torna mais fácil rastrear quem fez cada alteração e também facilita que várias pessoas trabalhem modificando um mesmo arquivo.

Duas entre as diferentes abordagens de versionamento de código incluem o versionamento local e o versionamento centralizado. O versionamento local é um processo de controle de versão realizado diretamente na máquina do desenvolvedor. Nessa abordagem, as modificações são feitas no diretório de trabalho, que é a pasta do projeto local, sendo rastreadas por um banco de dados de versão local. Isso permite que o desenvolvedor gerencie e acompanhe as alterações feitas no código-fonte sem depender do acesso a um servidor remoto. Dessa forma, o versionamento local oferece mais autonomia e flexibilidade ao desenvolvedor durante o processo de controle de versão. Já no versionamento centralizado, é utilizado um servidor remoto

para compartilhar as versões do projeto entre os desenvolvedores. Entretanto, o ponto negativo desses sistemas é que por serem centralizados, falhas no servidor podem acarretar na ociosidade do time de desenvolvimento, ou até a perda do histórico de mudanças.

Por outro lado, o Git se destaca por ser um sistema de controle de versão distribuído, com uma abordagem flexível que combina as vantagens do versionamento local e do centralizado. No Git, cada desenvolvedor possui uma cópia completa do repositório em sua máquina. Isso permite que trabalhem de forma independente, cada um com um histórico local de versões. O Git também permite a sincronização e o compartilhamento entre versões de diferentes colaboradores por meio de um repositório remoto, onde as modificações de diferentes desenvolvedores são mescladas. Como o repositório local é independente do repositório remoto, caso o servidor de Git fique indisponível, os desenvolvedores conseguem continuar trabalhando.

No contexto do Git, plataformas como GitHub[3], GitLab[4] e BitBucket[5] desempenham um papel fundamental no desenvolvimento colaborativo. Além de fornecerem servidores para hospedar repositórios Git, essas plataformas oferecem recursos adicionais que facilitam o trabalho em equipe. Elas fornecem recursos avançados para revisão de código, discussão de problemas, gerenciamento de tarefas, planejamento de projetos e documentação. Além disso, essas plataformas permitem que os repositórios de código aberto sejam disponibilizados para uso e colaboração.

Em um repositório Git, as modificações são realizadas de forma preliminar no diretório de trabalho. Para cada modificação que se deseja criar uma nova versão, é necessário primeiro adicionar os arquivos modificados a uma área de preparação (*staging*). Com isso, é possível realizar um *commit*, que na prática, faz com que os arquivos da área de preparação façam parte de uma nova versão do repositório. Caso haja alguma modificação no diretório de trabalho que não foi adicionada a área de preparação, essa modificação não é propagada para a nova versão. A Figura 14.1 mostra que as modificações feitas no diretório de trabalho são enviadas para a área de preparação por meio do comando *git add*. Por conseguinte, as modificações da área de preparação são salvas em uma nova versão do repositório por meio do comando *git commit*.

O Git é um SCV em que o banco de dados de versão é armazenado em uma pasta especial na raiz do projeto, chamada `.git`. Essa pasta contém todas as informações necessárias para rastrear as versões, como por exemplo, o histórico de alterações do projeto e informações sobre o autor de cada modificação. Dessa forma, mesmo em casos de perda dos arquivos do projeto, desde que a pasta `.git` seja preservada, é possível recuperar versões anteriores dos arquivos.

Nesse contexto, a Figura 14.2 apresenta as três principais áreas de um projeto Git: o diretório de trabalho chamado na imagem de *working directory*, área de preparação chamada de *staging area* e a pasta `.git` que é o repositório que contém

[3]https://github.com
[4]https://about.gitlab.com
[5]https://bitbucket.org

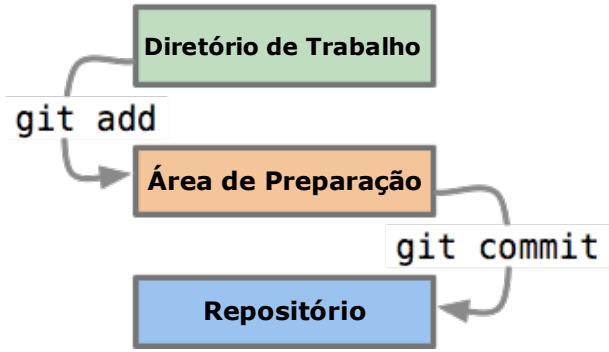


Figura 14.1: Propagação das modificações
Adaptado de Chacon (2009)

os arquivos de versionamento. A imagem contém setas que ilustram as ações realizadas para alterar a área do projeto em que os arquivos se encontram. Partindo do diretório de trabalho, as modificações são movidas para a área de preparação e, em seguida, é feito o *commit*, que registra as modificações em preparação como parte de uma nova versão. Além disso, a Figura 14.2 também ilustra que é possível trazer uma versão do repositório para o diretório de trabalho. Isso pode ser feito porque cada *commit* possui um identificador único, o que permite escolher uma versão anterior do projeto e trazer para o diretório de trabalho.

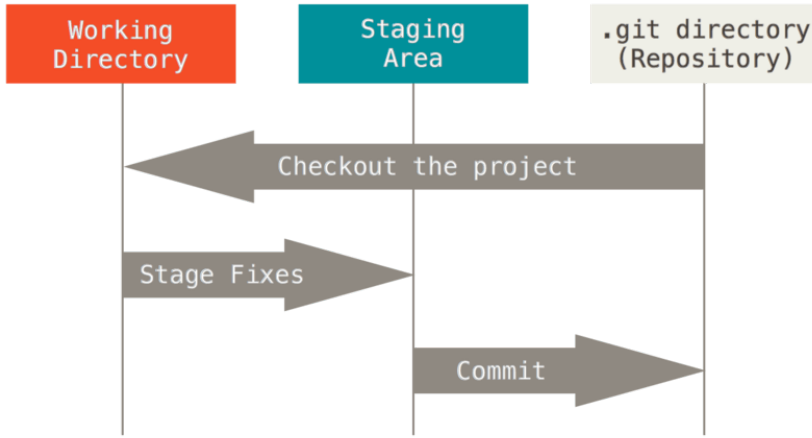


Figura 14.2: Seções de um projeto Git
Chacon e Straub (2014)

Além dos estados de um projeto Git na perspectiva de versões, o ciclo de vida dos arquivos é um conceito fundamental nesse sistema de versionamento. A Figura 14.3 ilustra esse processo, mostrando que, ao criar um novo arquivo no diretório de trabalho, ele inicia com o *status* de *untracked* (não rastreado), pois o Git ainda não foi configurado para monitorar suas alterações. Quando o arquivo é adicionado à área de preparação, seu *status* muda para *staged*, e, após o *commit*, passa a ser considerado *unmodified* (não modificado). Isso ocorre porque, nesse momento, seu conteúdo é idêntico ao da versão mais recente no repositório. Caso o arquivo seja modificado, seu estado muda para *modified*, indicando que há alterações pendentes. Ao realizar um novo *commit*, ele retorna ao estado *unmodified*, completando assim mais um ciclo no fluxo de versionamento do Git.

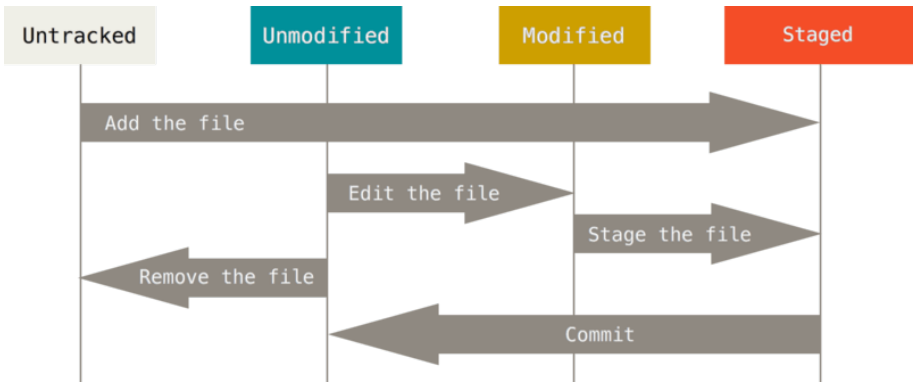


Figura 14.3: Ciclo de vida dos arquivos
Chacon e Straub (2014)

O *HEAD* é uma referência importante no controle de versão do Git. Ele aponta para um *commit* específico, marcando a posição atual do histórico de versões. Desse modo, as versões dos arquivos presentes no *commit*, que essa referência aponta, serão as mesmas dos que estão no diretório de trabalho. Essa referência permite comparar as alterações em cada arquivo e determinar se um arquivo está sendo rastreado ou não, bem como se possui modificações em relação à versão atual no histórico de versões. O comando `git checkout` é utilizado para alterar a referência do *HEAD*, possibilitando a transição entre diferentes versões de um repositório.

Ao criar um projeto Git, automaticamente é criada uma *branch* com o nome "master" em que os *commits* são realizados. Cada *commit* aponta para um *commit* anterior, conforme ilustrado na Figura 14.4. Nesse sentido, o histórico de *commits* da imagem é formado por C0, C1 e C2. Já uma *branch*, consiste em uma referência que aponta para um *commit* específico, assim como o *HEAD*. No caso da imagem, a *branch* "master" é uma referência que aponta para C2.

Quando dois ou mais *commits* têm um mesmo *commit* como ponto de partida, isso é conhecido como *branching*, o que resulta na criação de um novo ramo no

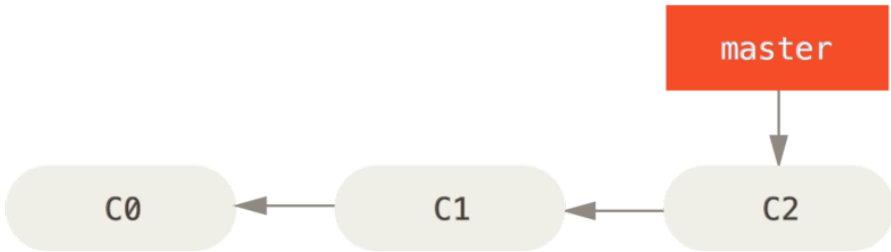


Figura 14.4: Ilustração de um histórico de *commits* linear
Chacon e Straub (2014)

histórico de versões do projeto. Essa situação cria diferentes linhas de desenvolvimento de um mesmo projeto. A abordagem facilita o desenvolvimento de recursos independentes e permite que várias alterações sejam realizadas simultaneamente sem interferir no código principal do projeto. Cada ramificação representa uma sequência única de *commits* que registram as modificações específicas feitas nessa linha de desenvolvimento.

No contexto da Figura 14.5, a partir de C2 é criada uma *branch iss53*, na qual são realizados os *commits* C3 e C5, de forma paralela, é feito o *commit* C4 na "master". O *commit* C6 é denominado *merge commit*, em que as modificações realizadas em *iss53* são mescladas com o conteúdo da "master". Ocasionalmente, o processo de *merge* não é realizado de forma automática devido a conflitos com relação ao conteúdo dos arquivos. Isso ocorre quando são feitas alterações nas mesmas partes de um arquivo em cada uma das *branches*. Neste caso, o *merge* é interrompido para que possam ser resolvidos os conflitos de forma manual, por meio da modificação do conteúdo dos arquivos.

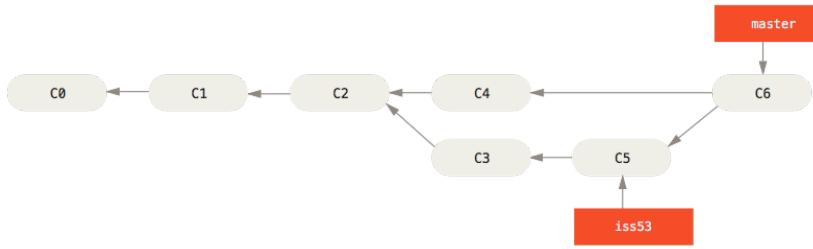


Figura 14.5: Ilustração de um histórico de *commits* com *branches* e *merges*

Fonte: Chacon e Straub (2014)

14.2.1 Comandos básicos

Os comandos apresentados na Figura 14.6, são `git init`, `ls` com o parâmetro `-a` e `git status`. O comando `git init` é usado para inicializar um novo repositório Git em uma máquina. Já o `ls` tem o intuito de mostrar quais arquivos e pastas pertencem à pasta atual. Os arquivos e pastas que têm um ponto no início do nome, como a pasta `.git`, não são mostrados por padrão pelo comando `ls`, por isso é necessário utilizar o parâmetro adicional `-a`. Por fim, o comando `git status` informa a situação dos arquivos e pastas em relação ao repositório, indicando, por exemplo, se há arquivos não rastreados, modificados ou em *staging*.

Nesse sentido, na Figura 14.6, após a inicialização do repositório, uma mensagem informando a criação do repositório é exibida. Ao listar os arquivos e pastas, observa-se a presença da pasta `.git` e do arquivo "index.html". Ao verificar o estado dos arquivos do repositório, é apresentado que o arquivo "index.html" está no estado *untracked*, ou seja, não está sendo rastreado pelo Git.

```
➜  git-example git init
Initialized empty Git repository in /Users/izaias/dev/tcc/git-example/.git/
➜  git-example git:(main) ✗ ls -a
.           ..          .git        index.html
➜  git-example git:(main) ✗ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        index.html

nothing added to commit but untracked files present (use "git add" to track)
```

Figura 14.6: Inicialização do projeto Git

Fonte: Próprio autor

Para que as modificações em um arquivo passem a ser rastreadas pelo Git, é necessário que ele faça parte de uma versão do repositório. O comando `git add`, usado na Figura 14.7, é utilizado para adicionar arquivos e pastas na área de preparação, especificando um caminho. Após adicionar o arquivo na área de preparação, é apresentado pelo comando `git status` que o estado de "index.html" é como um novo arquivo. Com isso, é necessário realizar um *commit* para que as modificações presentes na área de preparação passem a fazer parte de uma nova versão do repositório de código.

Na Figura 14.7, após realizar o *commit* do conteúdo da área de preparação, com o comando `git commit`, e utilizar novamente o comando `git status`, não há mais modificações para serem incluídas em um novo *commit*. Essa informação é indicada pela mensagem do Git após a execução do último comando. Isso ocorre porque o arquivo "index.html" agora está sendo rastreado pelo Git e seu conteúdo é idêntico ao conteúdo presente no repositório. Portanto, o arquivo está em sincronia

com a última versão do repositório e não há mais alterações a serem registradas, conforme esperado.

O comando utilizado para realizar o *commit* na Figura 14.7 possui um parâmetro `-m`, referente a uma mensagem utilizada para descrever as alterações realizadas. Caso a mensagem seja confusa ou não identifique o propósito do *commit*, isso pode atrapalhar a equipe de desenvolvimento, especialmente se for necessário encontrar um *commit* após algum tempo (GITKRAKEN, 2023). Com relação a quantidade de alterações por *commit*, é necessário que cada modificação possa ser descrita claramente por meio do parâmetro de mensagem, que tem tamanho máximo de 72 caracteres. Por conseguinte, é recomendado fazer mais de um *commit* se as modificações realizadas não possam ser descritas dada a restrição de tamanho da mensagem.

```
➜  git-example git:(main) ✗ git add index.html
➜  git-example git:(main) ✗ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   index.html

➜  git-example git:(main) ✗ git commit -m "adicionar arquivo html"
[main (root-commit) b6b45b9] adicionar arquivo html
 1 file changed, 10 insertions(+)
 create mode 100644 index.html
➜  git-example git:(main) git status
On branch main
nothing to commit, working_tree clean
```

Figura 14.7: *Commit* do arquivo *index.html*

O comando apresentado `git log` é utilizado para visualizar o histórico de *commits* de um repositório. A Figura 14.8 apresenta o resultado do uso desse comando, exibindo informações relevantes sobre cada *commit*. Para cada *commit*, é listado um identificador único, autor do *commit*, data em que foi realizado, descrição da modificação e também a indicação de qual *commit* é a referência *HEAD*. Essas informações são essenciais para acompanhar o progresso no desenvolvimento, entender as alterações realizadas ao longo do tempo e facilitar a colaboração entre os membros da equipe.

```
commit b6b45b91b5fcc96bde9eccc15879bd152f58d878 (HEAD -> main)
Author: Izaias Machado <izaiasmachado.dev@gmail.com>
Date:   Mon Jun 5 19:08:53 2023 -0300

    adicionar arquivo html
(END)
```

Figura 14.8: Histórico de modificações do repositório

Quando um arquivo existente no repositório é modificado no diretório de trabalho, o Git é capaz de identificar e registrar as alterações específicas em relação à versão anterior. Isso significa que o Git pode rastrear as modificações feitas no conteúdo dos arquivos ao longo do tempo. Por outro lado, novas pastas e arquivos são elementos que não existiam nas versões anteriores do repositório. Ao serem adicionados pela primeira vez ao diretório de trabalho, esses elementos não têm histórico de versões anteriores, por isso, o Git os trata como novos arquivos a serem rastreados (*untracked*). Dessa forma, o Git diferencia entre modificações de arquivos existentes e arquivos não rastreados, permitindo um controle mais preciso das alterações realizadas no projeto.

Nesse contexto, para demonstrar a capacidade de rastrear modificações em arquivos, bem como a criação de novos arquivos e pastas, foram feitas alterações no diretório de trabalho do exemplo anterior. Inicialmente, o arquivo "index.html", já incluído em um *commit*, teve seu conteúdo modificado. Além disso, foi criada uma pasta chamada "assets", contendo dois arquivos: "script.js" e "style.css", com o intuito de destacar a habilidade do Git de identificar novas pastas. Com essas modificações no diretório de trabalho, a Figura 14.9 apresenta o *status*. O resultado indica que o arquivo "index.html" sofreu alterações em relação à última versão e que uma nova pasta não rastreada, chamada "assets", foi adicionada. Utilizando o comando `tree` (que não faz parte do Git), é possível visualizar a estrutura de arquivos do diretório e verificar que a pasta não rastreada contém os arquivos "script.js" e "style.css".

```
➜  git-example git:(main) ✗ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        assets/

no changes added to commit (use "git add" and/or "git commit -a")
➜  git-example git:(main) ✗ tree
.
├── assets
│   ├── script.js
│   └── style.css
└── index.html

1 directory, 3 files
```

Figura 14.9: Restauração do diretório de trabalho -- Parte 01

Fonte: Próprio autor

O comando `git restore` é utilizado para restaurar o conteúdo de um ou mais arquivos para a versão do *commit* referenciado pelo *HEAD*. Na Figura 14.10, esse comando é utilizado passando o caminho dos arquivos a serem restaurados.

Adicionalmente, poderia ser passado o caminho para uma pasta ou arquivo específico, de modo a restaurar somente o conteúdo do caminho. Após restaurar os arquivos da pasta e verificar o *status*, é constatado que o arquivo "index.html" teve seu conteúdo restaurado, pois deixa de ser *modified*. Porém, a pasta "assets" continua no diretório de trabalho.

Considerando esse contexto, é necessário utilizar o comando `git clean`, para que os arquivos e pastas não rastreados pelo Git sejam removidos. O uso desse comando é exemplificado na Figura 14.10. Com isso, após a execução da diretiva `clean` é verificado o *status* novamente e o resultado é que não há nenhum *commit* que possa ser realizado, conforme apresentado pela Figura 14.10. Isso acontece porque a pasta "assets" foi removida, junto dos arquivos "script.js" e "style.css".

```
➜  git-example git:(main) ✗ git restore .
➜  git-example git:(main) ✗ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        assets/

nothing added to commit but untracked files present (use "git add" to track)
➜  git-example git:(main) ✗ git clean -f -d
Removing assets/
➜  git-example git:(main) git status
On branch main
nothing to commit, working_tree clean
```

Figura 14.10: Restauração do diretório de trabalho -- Parte 02

14.2.2 Comandos para utilização de branches

Para listar todas as *branches* no repositório local pode ser utilizado o comando `git branch`. O uso desse comando é exemplificado na Figura 14.11, o resultado lista todas as *branches*, mas também a *branch* atual é marcada em verde e com um asterisco a esquerda de seu nome. Ainda, na Figura 14.11, a fim de criar uma nova *branch* é usado o comando `git checkout` com o parâmetro `-b`, seguido de "traducao-portugues", que é o nome da ramificação. O comando `git branch` é utilizado novamente para verificar que a ramificação de tradução para português foi de fato criada. Além disso, a ramificação "traducao-portugues" passa a ser a *branch* atual.

Nesse contexto, na Figura 14.12, são feitas modificações no arquivo "index.html" que são adicionadas a área de preparação e salvas por meio de um *commit*. O comando `git checkout` é utilizado sem o parâmetro adicional para que a *branch* atual passe a ser a "main". Mudar a *branch* principal consiste em alterar o *commit* referenciado pelo *HEAD*. Nesse sentido, na *branch* "main" também é feito um *commit* que modifica o arquivo "index.html". Por fim, é realizado um *merge commit* utilizando o comando `git merge` seguido do nome da *branch* que deseja fazer o *commit*.

```
➜  git-example git:(main) git branch
* main
➜  git-example git:(main) git checkout -b traducao-portugues
Switched to a new branch 'traducao-portugues'
➜  git-example git:(traducao-portugues) git branch
  main
* traducao-portugues
```

Figura 14.11: Criação e listagem de *branches*

```
➜  git-example git:(traducao-portugues) git add index.html
➜  git-example git:(traducao-portugues) ✗ git commit -m "traduzir para portugues
"
[traducao-portugues f112357] traduzir para portugues
 1 file changed, 2 insertions(+), 2 deletions(-)
➜  git-example git:(traducao-portugues) git checkout main
Switched to branch 'main'
➜  git-example git:(main) git add index.html
➜  git-example git:(main) ✗ git commit -m "adiciona corpo do texto"
[main 65b3376] adiciona corpo do texto
 1 file changed, 4 insertions(+)
➜  git-example git:(main) git merge traducao-portugues
Auto-merging index.html
Merge made by the 'ort' strategy.
 index.html | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)
```

Figura 14.12: *Commits* e *merge* de *branches*

O comando `git log` pode ser utilizado para visualizar o histórico de modificações, como exemplificado previamente. De forma adicional, com os parâmetros mostrados na Listagem 14.1 é possível visualizar o histórico de *commits* mostrando somente as informações relevantes sobre as alterações realizadas e em que *branches* elas foram feitas. O resultado produzido pode ser visto na Figura 14.13, em que é ilustrado que os *commits* para adicionar o corpo do texto e traduzir para português foram feitos de forma paralela e mesclados na *branch* "main" por um *merge commit*.

```
1 git log --oneline --graph
```

Listagem 14.1: Mostra histórico *commits* em linha única e *branches* em modo gráfico

Ao finalizar as modificações de uma *branch*, a mesma pode ser deletada por meio do comando mostrado na Listagem 14.2. O comando `git branch` é o mesmo utilizado para listar as ramificações no repositório local, seguido de um parâmetro `-D` que é acompanhado do nome da *branch* que se deseja remover. Assim como a listagem de *branches*, esse comando serve somente para remover ramificações do repositório local.

```
1 git branch -D traducao-portugues
```

Listagem 14.2: Comando para deletar a *branch* de tradução para português

```
*   eec899e (HEAD -> main) Merge branch 'traducao-portugues'
|\
| * f112357 (traducao-portugues) traduzir para portugues
* | 65b3376 adiciona corpo do texto
|/
* ca36b7d adiciona titulo
* b6b45b9 adicionar arquivo html
(END)
```

Figura 14.13: Histórico de *commits* em linha única e *branches* em modo gráfico

Antes dos comandos mostrados na Figura 14.14, foi feita uma *branch* de nome "muda-titulo" que possui um *commit* que muda o título do arquivo "index.html". Em paralelo, na ramificação principal também foi modificado o título da página. Com isso, ao tentar realizar o *merge*, é reportado um conflito, assim como apresentado pela imagem. Ainda na Figura 14.14, é mostrado o conteúdo mesclado do arquivo "index.html", presente nas *branches* "main" e "muda-titulo", em que são adicionados delimitadores que demarcam os conflitos e a *branch* de origem de cada conteúdo.

Figura 14.14: Erro de *merge* e conteúdo conflituoso dos arquivos

```
→  git-example git:(main) git merge muda-titulo
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
→  git-example git:(main) ✗ cat index.html
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Documento</title>
  </head>
  <body>
<<<<<<< HEAD
    <h1>Porquê Git? (main)</h1>
=======
    <h1>Título Novo Aqui (branch muda-titulo)</h1>
>>>>>>> muda-titulo
    <p>
      Lorem ipsum dolor sit amet consectetur adipisicing elit. Quisquam
      voluptatibus, quibusdam, quia, quos voluptatem voluptatum quod
    </p>
  </body>
</html>
```

Por fim, para resolver os conflitos, devem ser removidos os delimitadores e o conteúdo dos arquivos deve ser mesclado manualmente pelo desenvolvedor. O conteúdo do arquivo resultante após a resolução é apresentado na Figura 14.15. Além disso, é necessário adicionar as alterações na área de *staging* e realizar o *commit*

das modificações, para que o *merge* seja finalizado. Caso o desenvolvedor precise cancelar a resolução dos conflitos, é possível utilizar o comando da Listagem 14.3.

```
➜  git-example git:(main) ✗ cat index.html
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Documento</title>
  </head>
  <body>
    <h1>Título Novo Aqui (main)</h1>
    <p>
      Lorem ipsum dolor sit amet consectetur adipisicing elit. Quisquam
      voluptatibus, quibusdam, quia, quos voluptatem voluptatum quod
    </p>
  </body>
</html>
➜  git-example git:(main) ✗ git add index.html
➜  git-example git:(main) ✗ git commit -m "fix conflicts"
[main 19e84ab] fix conflicts
```

Figura 14.15: Resolução do conflito

```
1 git merge --abort
```

Listagem 14.3: Abortar *merge commit* conflituoso

14.2.3 Uso de um servidor remoto de Git

Para importar um repositório já existente para servidores remotos de Git, o repositório deve ser criado no site e copiada a URL do repositório; com isso, o *link* encontrado é substituído no comando da primeira linha da Listagem 14.4. É possível verificar se a ação foi bem-sucedida pelo comando da segunda linha da própria listagem. Caso o repositório tenha sido criado diretamente no site que fornece um servidor remoto de Git, é possível clonar o repositório remoto para a máquina local utilizando o comando `git clone`, como exemplificado na Listagem 14.5.

```
git remote add origin <url-repositorio>
git remote -v
```

Listagem 14.4: Adicionar e verificar link do repositório remoto

```
git clone <url-repositorio>
```

Listagem 14.5: Clone de um repositório remoto

Nesse contexto, é possível utilizar o comando da primeira linha da Listagem 14.6 para enviar os *commits* feitos no repositório local para uma *branch* correspondente no repositório remoto. Entretanto, caso tenha sido criada uma *branch* no repositório local, sem existir uma correspondente no repositório remoto, essa ação resulta em um erro. Para lidar com essa situação, é necessário utilizar o comando da segunda linha da Listagem 14.6 para criar a nova *branch* no repositório remoto. Deve-se ressaltar que as modificações feitas em *branches* remotas não são sincronizadas automaticamente com o repositório local. Portanto, é importante entender as formas de sincronização descritas a seguir.

```
git push
git push --set-upstream origin <nome-da-branch>
```

Listagem 14.6: Enviando alterações para o repositório remoto

A Figura 14.16 ilustra uma ramificação em azul que corresponde à versão local da *branch* "main", já em verde é mostrada a versão do repositório remoto da *branch* principal e em roxo está destacado o último *commit* em comum entre as ramificações. Nesse sentido, ao executar o comando `git fetch`, destacado na primeira linha da Listagem 14.7, são baixadas as modificações feitas no repositório remoto, mas isso não modifica a *branch* ou o *HEAD* do repositório local. Por conseguinte, para que as duas ramificações sejam mescladas, é necessário realizar um *merge commit* com o comando `git merge` entre as *branches*, conforme explicado na seção anterior.

```
git fetch
git pull
```

Listagem 14.7: Baixando alterações do repositório remoto

Na Figura 14.17 é ilustrado o resultado do comando `git pull`, apresentado na segunda linha da Listagem 14.7, que consiste em um `git fetch` seguido de um `git merge`. Entretanto, é importante ressaltar que o uso desse comando, assim como os outros *merge commits*, está sujeito a conflitos de *merge* que precisam ser resolvidos manualmente.

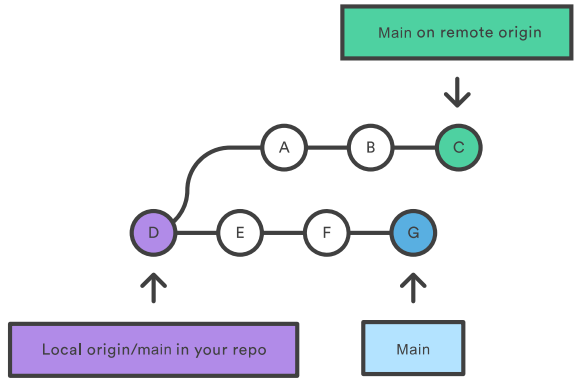


Figura 14.16: Resultado após realizar um *fetch*

Fonte: Atlassian (2023)

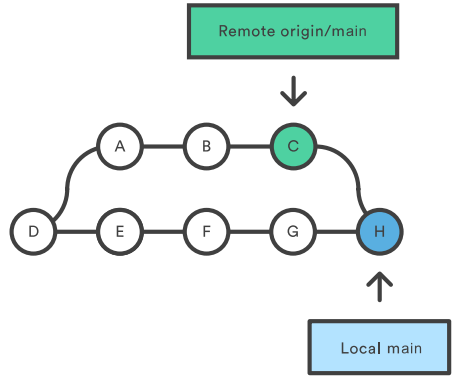


Figura 14.17: Resultado após realizar um *pull* ou *fetch* seguido de *merge*

Fonte: Atlassian (2023)

14.3 Compartilhamento e gerenciamento de código com GitHub

Além de servir como um servidor remoto para hospedar repositórios com o sistema de controle de versão Git, o GitHub é um ambiente que fornece ferramentas que visam facilitar o trabalho em colaboração com outras pessoas (BEER, 2018). Nesse sentido, é fornecida uma interface web amigável que permite visualizar, discutir e revisar as modificações. É importante mencionar que, além da colaboração em projetos pessoais ou de empresas, o GitHub também oferece funcionalidades que viabilizam a contribuição para projetos de código aberto.

14.3.1 Conceitos principais

A Figura 14.18 mostra a página acessível ao repositório de código *Developer Roadmap*[6]. A imagem apresenta os diversos arquivos e pastas do repositório, mas foram destacadas em vermelho as partes **(a)**, **(b)** e **(c)** para discussão. Em **(a)** são apresentadas uma breve descrição, palavras-chave e o link para o *website* do projeto. Já na parte **(b)** da captura de tela, no botão *watch* é mostrado que 6,9 mil pessoas são notificadas das modificações no projeto, 34 mil fizeram o *fork* e 242 mil marcaram com estrela. O significado do *fork* será explicado ainda nesta seção, já a estrela serve para que o repositório fique salvo em uma lista de repositórios favoritos. Ainda, em **(b)**, existem o nome do repositório, o nome autor do repositório e também submenus. Por fim, em **(c)**, é mostrado o número do *commits* feitos na *branch* principal do repositório. Além disso, ao clicar na quantidade de *commits*, em **(c)**, o usuário é redirecionado para o histórico de *commits*.

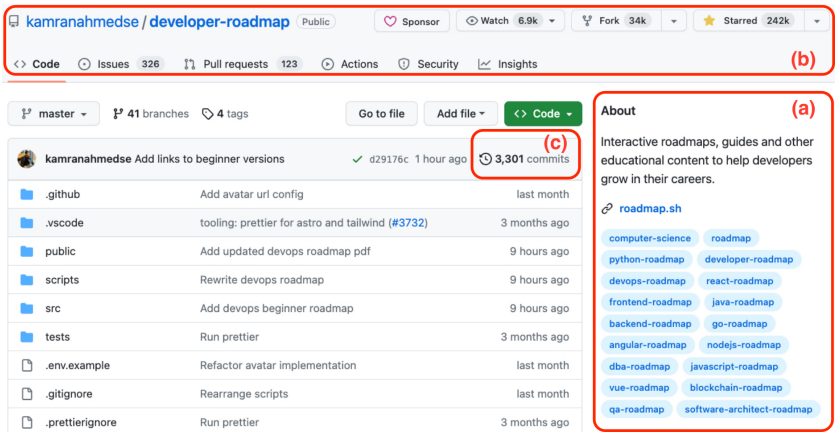


Figura 14.18: Repositório *Developer Roadmap*

[6]https://github.com/kamranahmedse/developer-roadmap

Além disso, na página do repositório, ao rolar a tela, pode ser visualizada a *README* do projeto, apresentada na Figura 14.19. Esta parte de qualquer repositório consiste em um arquivo mostrado na tela inicial do repositório, que tem nome *README.md*, e é utilizado para fornecer uma introdução e informações importantes sobre o projeto. O GitHub utiliza um padrão chamado *Markdown* para formatação de texto, inserção de imagens, tabelas, blocos de código e links. Esse tipo de formatação pode ser utilizado em campos de texto dentro da ferramenta, mas também em qualquer arquivo com extensão *md*.

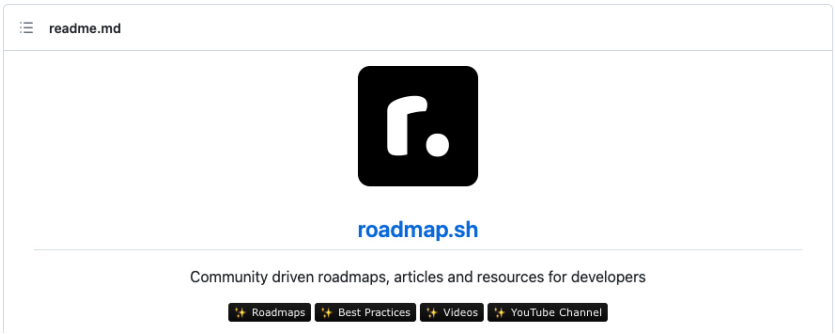


Figura 14.19: *README* do repositório *Developer Roadmap*

No repositório local é utilizado o comando `git log` para mostrar os *commits*, já no GitHub pode ser visualizada a lista de modificações no histórico de *commits*, apresentada na Figura 14.20. Cada modificação é seguida de sua descrição, do nome do colaborador, de há quanto tempo foi feita a modificação e também de um identificador único do *commit*. Note que, assim como no Git, o histórico de modificações varia conforme a *branch*. Devido a isso, existe um menu que permite alternar entre as ramificações do projeto, esse menu está destacado na captura de tela da Figura 14.20.

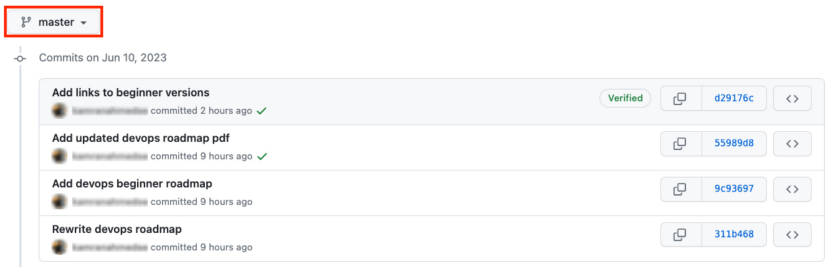


Figura 14.20: Histórico de *commits* do repositório *Developer Roadmap*

Outra funcionalidade importante é que, ao clicar no identificador único de um *commit* o usuário é redirecionado para a tela da Figura 14.21, que mostra a

modificação que foi feita. Em verde e ao lado de um sinal de mais são apresentadas as linhas que foram adicionadas e em vermelho, ao lado de um sinal de menos, estão as linhas que foram removidas no *commit*.

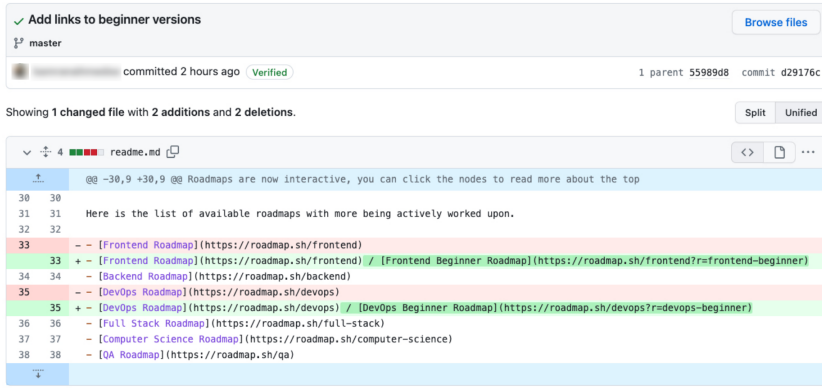


Figura 14.21: Resultado após realizar um *pull* ou *fetch* seguido de *merge*

Ao realizar o pedido de *pull request*, um desenvolvedor solicita que alterações feitas em uma *branch* sejam mescladas em uma outra ramificação, por padrão a solicitação é feita para a *branch* principal. Na Figura 14.22, é apresentado que o autor do *pull request* adiciona um título e também pode adicionar uma descrição. A descrição pode conter explicação sobre os *commits* feitos, capturas de tela e outros detalhes relevantes para a modificação que vai ser introduzida. Além disso, existe também uma seção de comentários em que outros desenvolvedores podem discutir sobre o *pull request*.

O conteúdo de uma *issue* é mostrado na Figura 14.23, que segue o mesmo padrão de um *pull request*. Ao invés de solicitar modificações, as *issues* podem ser utilizadas para sugerir novas funcionalidades, reportar *bugs* e realizar discussões em geral. Diferente do *pull request*, a *issue* não é associada com *commits*, mas também tem um tópico com descrição e outros desenvolvedores podem realizar comentários.

Cada *issue* e *pull request* no GitHub é atribuído a um identificador único, que permite referenciar e vincular discussões entre eles. Um exemplo disso é destacado na Figura 14.23, onde a *issue* possui o identificador `#3280`. Esse identificador pode ser utilizado para fazer referência a essa discussão em outras *issues*, *pull requests* ou comentários, tanto na descrição quanto nos comentários. Os identificadores únicos citados nas discussões funcionam como links para acessar rapidamente a *issues* ou *pull requests* correspondente.

14.3.2 Criação e manutenção de um projeto próprio

O GitHub oferece, no canto superior direito, um menu de criação acessível em qualquer página do *site*. A Figura 14.24 mostra esse menu, além disso, na

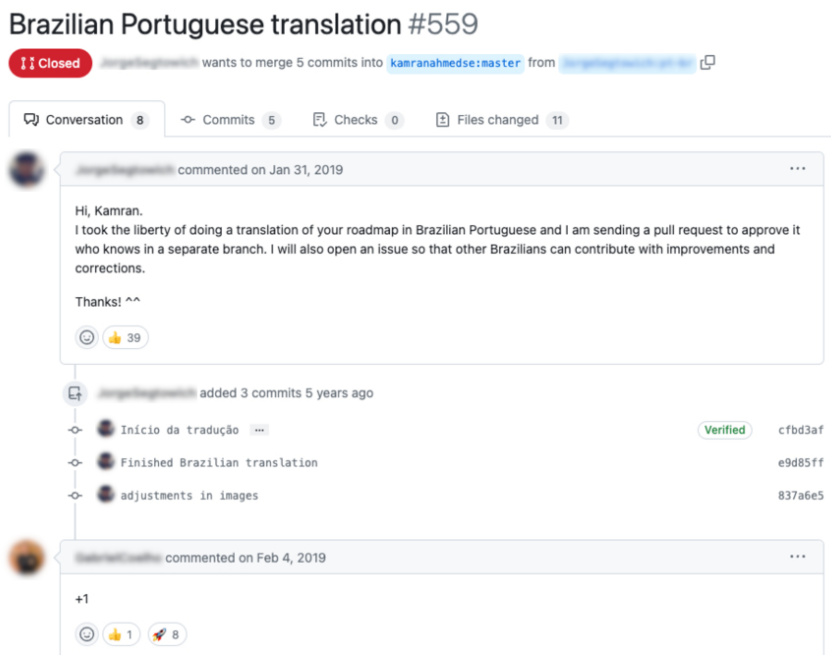


Figura 14.22: Corpo de um *pull request*

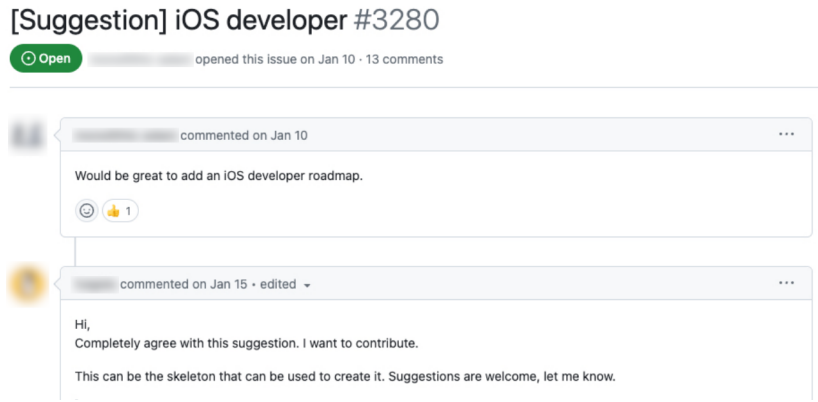


Figura 14.23: Corpo de uma *issue*

imagem foram destacadas as opções **(a)** e **(b)** para discussão. Destacado por **(a)** na imagem, temos a opção que permite criar um novo repositório. Essa é a escolha ideal para iniciar um projeto completamente novo. Ao selecionar essa opção, o usuário será direcionado para a página de criação de um novo repositório, onde

poderá inicializar o novo projeto. Por outro lado, o botão destacado por **(b)**, na Figura 14.24, oferece a possibilidade de importar um repositório existente de outra plataforma de hospedagem de repositório Git, como GitLab ou Bitbucket. Essa opção é útil para migrar um projeto para o GitHub, aproveitando todos os recursos e funcionalidades oferecidos pela plataforma. Ao utilizar a importação do GitHub, é possível transferir projeto, incluindo o histórico de versões (GITHUB, 2023).

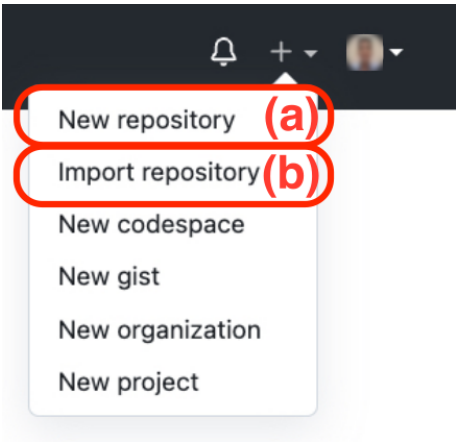


Figura 14.24: Menu de criação

A Figura 14.25 e a Figura 14.26 ilustram a página de criação de um novo repositório no GitHub. É importante notar que a Figura 14.26 é uma continuação da Figura 14.25 e também que ambas têm áreas destacadas para a discussão. Essas imagens fornecem uma visão geral do processo de criação de um novo repositório e ajudam os usuários a entender as diferentes opções disponíveis ao configurar um projeto no GitHub.

O primeiro fragmento da página para criação de um novo repositório é apresentado na Figura 14.25. Em **(a)**, há um menu que permite navegar entre os repositórios *templates* salvos pelo desenvolvedor, que podem ser usados como base para a criação do novo repositório. Em **(b)**, há uma caixa de texto na qual é possível nomear o novo repositório. Nessa mesma área, um texto em verde destaca que o nome escolhido está disponível. Isso ocorre porque não é permitido ter dois repositórios com exatamente o mesmo nome em uma mesma conta do GitHub. Já em **(c)**, é apresentada outra caixa de texto que permite adicionar uma descrição, que é mostrada na página do repositório.

A Figura 14.26 apresenta o segundo fragmento da página de criação de um novo repositório. Em **(d)**, é possível escolher entre disponibilizar o repositório de forma pública ou privada. Ao torná-lo público, qualquer pessoa da internet poderá visualizar o repositório, mas apenas as pessoas selecionadas poderão contribuir para ele. Já na opção privada, é necessário adicionar pessoas específicas a uma lista para

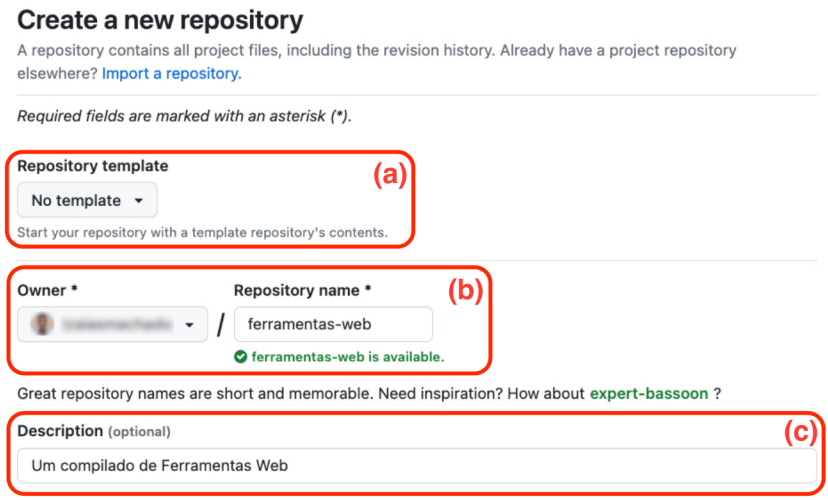


Figura 14.25: Criação de um repositório -- Parte 01

que possam ver o repositório e fazer contribuições. É importante destacar que, após a criação do repositório, é possível alterar sua configuração de público para privado e vice-versa por meio da página de configurações do repositório.

Na área destacada por **(e)**, na captura de tela da Figura 14.26, pode-se marcar uma opção para que um arquivo *README* seja criado de forma automática. Essa opção permite a criação do arquivo no processo de inicialização do repositório, eliminando a necessidade de criar o arquivo usando um cliente Git. Destacado por **(f)**, existe um menu que permite selecionar um dos *templates* disponíveis para o arquivo `.gitignore`. Esse arquivo é utilizado para especificar quais arquivos ou diretórios devem ser ignorados pelo sistema de controle de versão Git ao realizar um *commit*. Na opção destacada por **(g)**, é possível escolher a licença de uso para o código do repositório. Uma das licenças de código aberto mais comuns é a Licença MIT. Essa licença permite o uso, modificação e distribuição do código, tanto para fins comerciais quanto não comerciais. Caso selecionada essa opção, será adicionado um arquivo de licença ao repositório com termos da licença.

Para finalizar o processo de criação de um novo repositório deve ser clicado no botão destacado por **(h)**, apresentado na Figura 14.26, para criar o repositório. Neste exemplo, o repositório tem o nome *ferramentas-web*, possui uma descrição associada a ele e foi criado de forma pública, incluindo automaticamente um arquivo *README.md*. Nenhuma opção de arquivo `.gitignore` foi selecionada e a licença MIT foi escolhida como a licença de uso para o código.

A Figura 14.27 mostra o repositório após a sua criação. Ao criar um repositório no GitHub, é automaticamente feito um *commit* inicial com a descrição "Initial commit". Nesse *commit* inicial, são adicionados os arquivos da licença escolhida e o

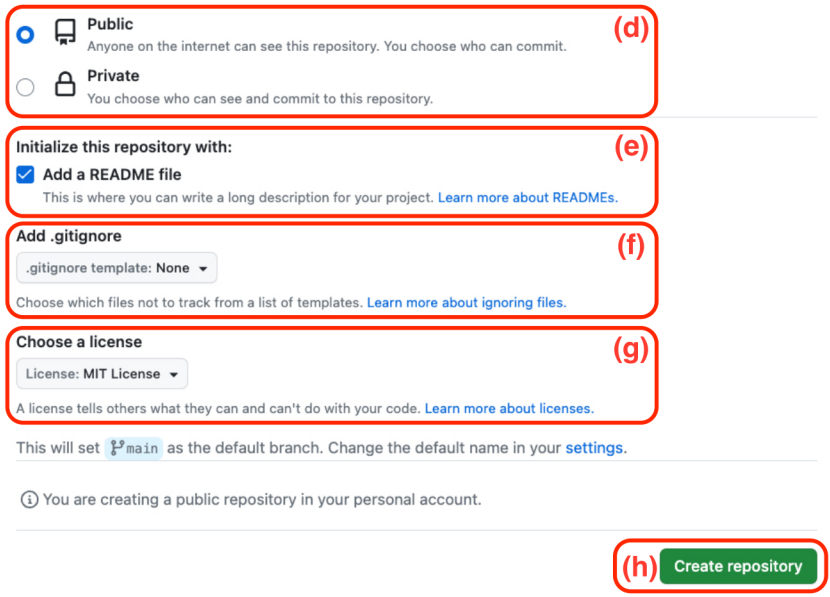


Figura 14.26: Criação de um repositório -- Parte 02

arquivo *README*, que contém informações importantes sobre o projeto. No exemplo apresentado na Figura 14.27, os arquivos *README* e licença foram adicionados, enquanto o arquivo `.gitignore` não foi incluído. É importante ressaltar que caso nenhum arquivo seja criado ou selecionado durante a criação do repositório, não haverá o *commit* inicial e o repositório será criado vazio, sem nenhum *commit* registrado no histórico de versões.

Está destacado em vermelho, na Figura 14.27, os botões: "Add file" e "Code". Com isso, o botão "Add file" está presente para adicionar novos arquivos ao repositório por meio da interface do GitHub, sem precisar executar comandos pelo Git. Já o botão verde de nome "Code" é utilizado para abrir o menu de opções relacionadas ao código do repositório, permitindo baixar ou também acessar o link do repositório, para realizar o clone com o Git.

A Figura 14.28 apresenta o menu exibido ao clicar no botão "Code". Esse menu fornece algumas informações e também opções para baixar e editar o código do repositório. Dentre essas opções, conforme destacado em vermelho na imagem, é listado o endereço do repositório remoto em HTTPS e SSH. Esse endereço é utilizado para baixar o repositório remoto para a máquina do desenvolvedor usando o comando `git clone` do Git, que é explicado detalhadamente na Seção 14.2.3.

Ao clicar no botão "Add file", disponível na página do repositório, é mostrado um menu com duas opções para a criação de um novo arquivo. Esse menu é apresentado na Figura 14.29, em que foram destacadas as seções **(a)** e **(b)** da

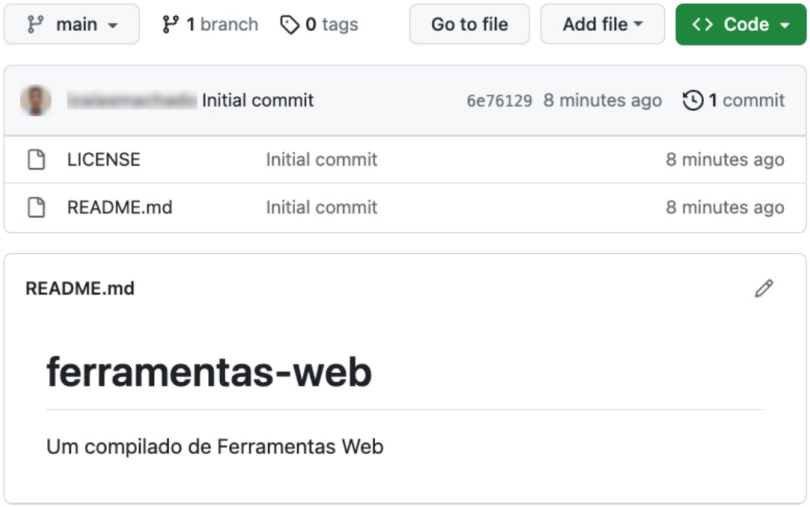


Figura 14.27: Repositório criado

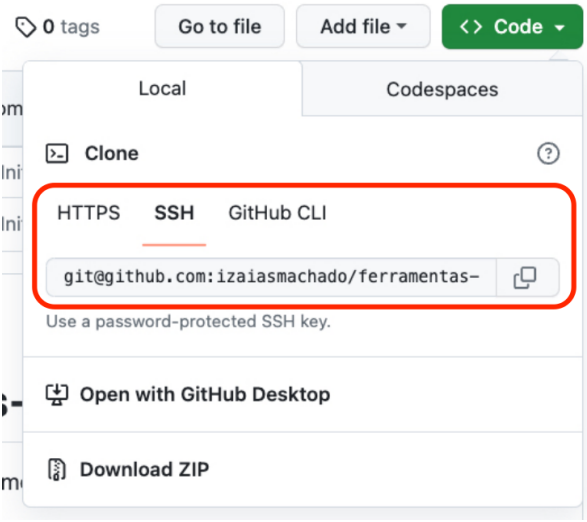


Figura 14.28: Fazer o clone do repositório

imagem. É importante ressaltar que, ao criar arquivos por meio da interface do GitHub, o processo de *commit* é realizado internamente pela plataforma, sem a necessidade de usar o Git. Nesse sentido, é apresentada em **(a)** a opção de criar um novo arquivo utilizando a interface web do GitHub. Já em **(b)**, encontra-se a opção

de realizar o *upload* de um arquivo da máquina do desenvolvedor diretamente para o repositório.

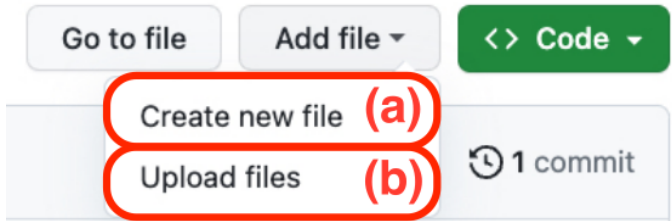


Figura 14.29: Menu para adicionar arquivos

Ao clicar na opção para criar um novo arquivo pela interface do GitHub, é aberta a página mostrada na Figura 14.29. Nessa página, além de adicionar o conteúdo do arquivo no editor de texto do site, há uma caixa de texto, destacada em vermelho na imagem, onde é possível adicionar o nome do arquivo. No exemplo dado, o nome do arquivo digitado foi *docs/introducao.md*, indicando que um arquivo chamado *introducao.md* será criado na pasta *docs*. Quando o desenvolvedor terminar de adicionar o conteúdo do arquivo, deve-se clicar no botão verde "Commit changes..." para fazer o *commit* do novo arquivo. Em um caso adverso, se necessário, é possível clicar no botão "Cancel changes" para descartar a criação do novo arquivo.



Figura 14.30: Página para criação de novo arquivo

Ao clicar no botão para realizar o *commit* do novo arquivo, uma tela de configurações do *commit* é aberta. Essa tela é apresentada na Figura 14.31, em que existem os campos de texto para a mensagem de *commit* e descrição estendida. A mensagem de *commit* consiste em uma breve descrição das alterações realizadas. Além disso, na caixa de texto maior é possível inserir uma descrição mais detalhada das modificações feitas. O preenchimento da mensagem de *commit* é obrigatório, enquanto a descrição mais detalhada é opcional. Destacado na imagem, há a opção de escolher entre fazer o *commit* diretamente na *branch* principal "main" ou criar uma nova *branch* para fazer o *commit*. No exemplo apresentado, foi escolhida a criação de uma nova *branch* com o nome *adiciona-introducao*.

Ao realizar o *commit* da nova *branch*, por estar utilizando a interface do GitHub, o desenvolvedor é redirecionado para uma tela de criação do *pull request*. É importante enfatizar que, geralmente, ao criar uma nova *branch* utilizando o Git

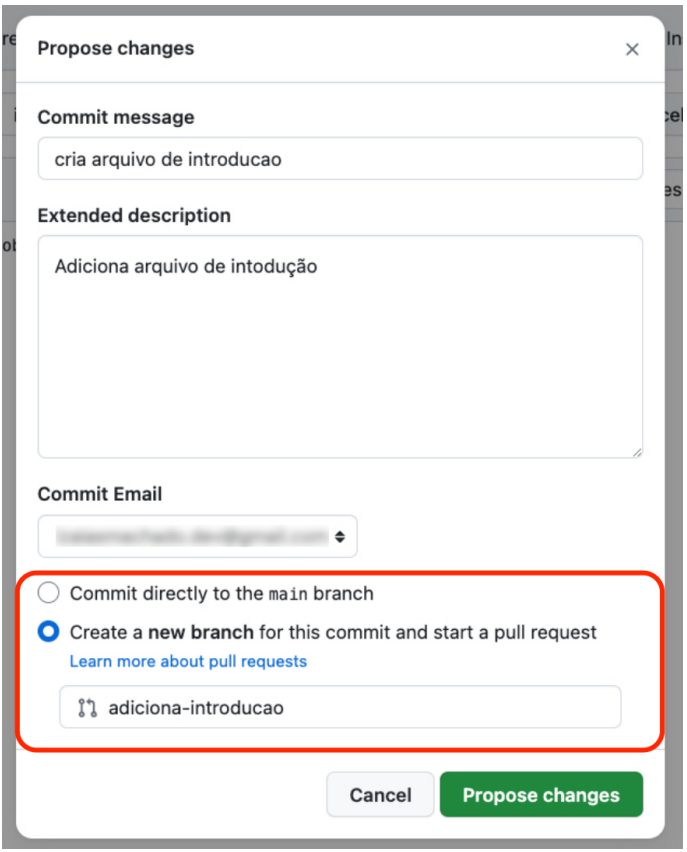


Figura 14.31: *Commit* para criação do novo arquivo

em uma máquina local, o desenvolvedor não é automaticamente direcionado para a tela de criação do *pull request* no GitHub. Essa transição ocorre apenas quando são feitos *commits* pela interface do GitHub.

O *pull request* abre uma solicitação de *merge* de *branches* por meio da interface do GitHub. Após aberta, a solicitação pode ser revisada e discutida pelos colaboradores do projeto. É importante ressaltar que o *merge* não necessariamente precisa ser realizado na *branch* principal. Dessa forma, o *pull request* permite que as modificações sejam incorporadas ao código de forma controlada e após a revisão dos colaboradores.

A Figura 14.32 apresenta um fragmento da tela do *pull request* que cria o arquivo *introducao.md*. Nessa parte da tela do *pull request*, pode-se observar no título, uma descrição do que está sendo proposto e também a mensagem do *commit* que foi feito. É também destacado que esse *pull request* solicita a realização do

merge da *branch adiciona-introducao* na branch principal “main”. Além disso, é mostrado um estado que é *Open*, para indicar o que está disponível para revisão e discussão pelos colaboradores do projeto.

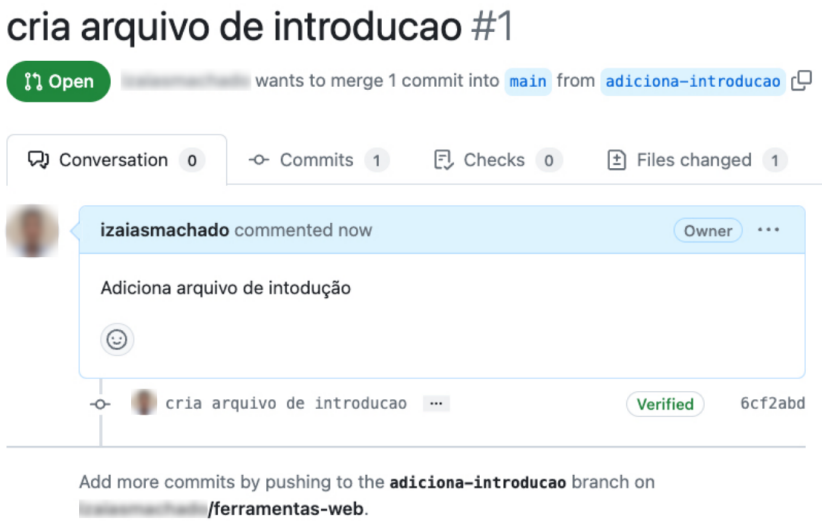


Figura 14.32: *Pull request* -- Parte 01

Outra seção da tela do *pull request* apresenta a informação de que não há conflito entre as *branches* envolvidas, como pode ser observado pelo texto em negrito na Figura 14.33. No caso de haver conflitos no conteúdo dos arquivos entre as *branches* do *pull request*, não será possível realizar o *merge* diretamente. Para resolver esses conflitos, é necessário utilizar um editor de texto no próprio GitHub ou o Git localmente.

O botão em destaque na cor vermelha, na Figura 14.33, permite realizar o *merge* proposto pelo *pull request*. É importante mencionar que apenas os colaboradores do projeto com permissão para realizar o *merge* têm acesso a esse botão. Após a conclusão do *merge*, o estado do *pull request* é alterado de *Open* para *Merged*, indicando que as modificações foram incorporadas à *branch* principal.

Seguindo a discussão sobre o *pull request*, é mostrado na Figura 14.34 a parte da página que permite fazer comentários e fechar o *pull request*. Nessa seção, os colaboradores têm a possibilidade de adicionar comentários com sugestões e observações sobre as modificações propostas. Desse modo, é por meio dos comentários que podem ser abordados ajustes de código e possíveis melhorias nas funcionalidades implementadas.

Para finalizar a discussão da Figura 14.34, destacado junto do botão para deixar um comentário, há também o botão para fechar o *pull request*. Ao clicar nesse botão, o *pull request* passa do estado aberto *Open* para o estado fechado *Closed*. O fechamento do *pull request* indica que a discussão está concluída, mas

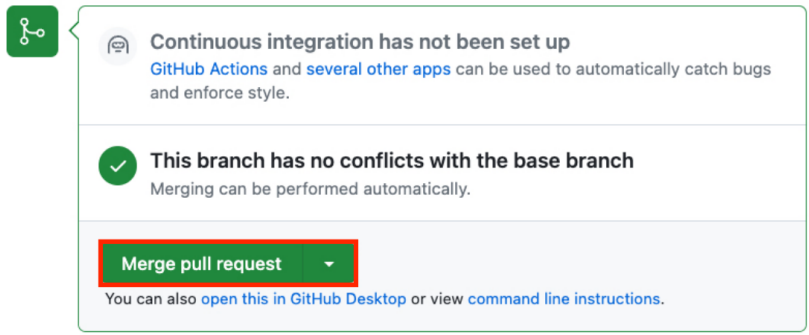


Figura 14.33: *Pull request* -- Parte 02

as modificações não são mescladas na *branch* de destino. Somente *pull requests* abertos permitem fazer o *merge*. Nesse sentido, caso seja decidido reconsiderar a solicitação, é possível reabrir o *pull request*, para que volte do estado *Closed* para *Open* novamente.

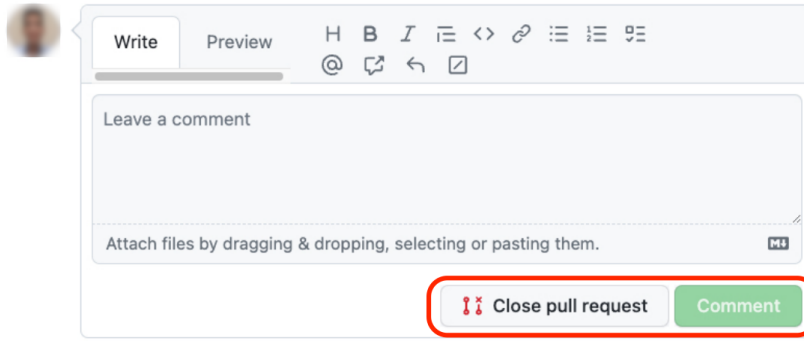


Figura 14.34: *Pull request* -- Parte 03

14.3.3 Como colaborar com projetos de código aberto?

Repositórios de código aberto são projetos de *software* disponibilizados publicamente para acesso, colaboração e contribuição da comunidade de desenvolvedores. Esses repositórios permitem que o código-fonte seja compartilhado de forma transparente e que pessoas de todo o mundo possam os estudar e contribuir. Muitas bases de código tem encontrado no GitHub uma plataforma ideal para sua hospedagem e colaboração. Podem ser citados alguns projetos de grande importância

que estão hospedados no GitHub, como o *kernel* do Linux[7], o *framework* Node.js[8] e a biblioteca do *React*[9]. Essa escolha se deve à ampla adoção do GitHub pela comunidade de desenvolvedores e os recursos que a plataforma oferece.

Para contribuir de forma direta, como na Seção 14.3.2 é necessário ser dono ou possuir permissões de colaborador em um repositório. No entanto, é possível contribuir para quaisquer repositórios fazendo uma cópia desse repositório em sua própria conta do GitHub, o que é conhecido como *fork*. Ao fazer um fork, é possível realizar modificações na cópia do repositório e, em seguida, solicitar que essas alterações sejam incorporadas ao repositório original por meio de um *pull request*. Dessa forma, o *fork* permite que os desenvolvedores contribuam com projetos em que não possuem acesso direto, possibilitando o envolvimento em projetos de código aberto e a colaboração com a comunidade de desenvolvedores.

O repositório *GitHub Example*[10] foi escolhido para exemplificar uma contribuição em um projeto por meio de um *fork*. Nesse sentido, na Figura 14.35, é mostrado a página principal do repositório em questão. O botão de *fork* está destacado em vermelho, e ao clicá-lo, inicia-se o processo de criação de uma cópia do repositório na conta do GitHub do desenvolvedor que realiza o *fork*.

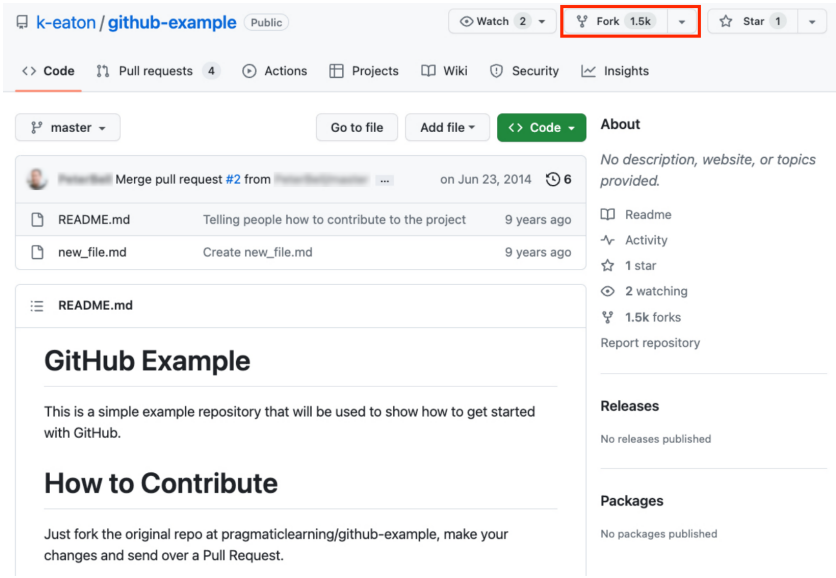


Figura 14.35: Repositório *github-example*

[7]https://github.com/torvalds/linux
[8]https://github.com/nodejs/node
[9]https://github.com/facebook/react
[10]https://github.com/k-eaton/github-example

Após clicar no botão de *fork*, o desenvolvedor é redirecionado para a página mostrada na Figura 14.36. Nessa imagem, existem algumas partes destacadas em vermelho e identificadas pelas letras **(a)**, **(b)** e **(c)**. Na Parte **(a)**, há uma caixa de texto na qual é possível dar um nome ao repositório que será criado. Por padrão, os *forks* têm o mesmo nome do repositório original, mas o desenvolvedor pode modificá-lo conforme necessário. Em **(b)**, é possível adicionar uma descrição ao novo repositório. Essa descrição pode ser útil para fornecer informações adicionais sobre o objetivo ou conteúdo do repositório. Posteriormente, após a criação do repositório *fork*, também será possível modificar a descrição. Por fim, em **(c)**, encontra-se o botão para efetuar a criação do repositório *fork*. Ao clicar nesse botão, o GitHub irá criar uma cópia do repositório original, associando-o à conta do desenvolvedor que realizou o *fork*.

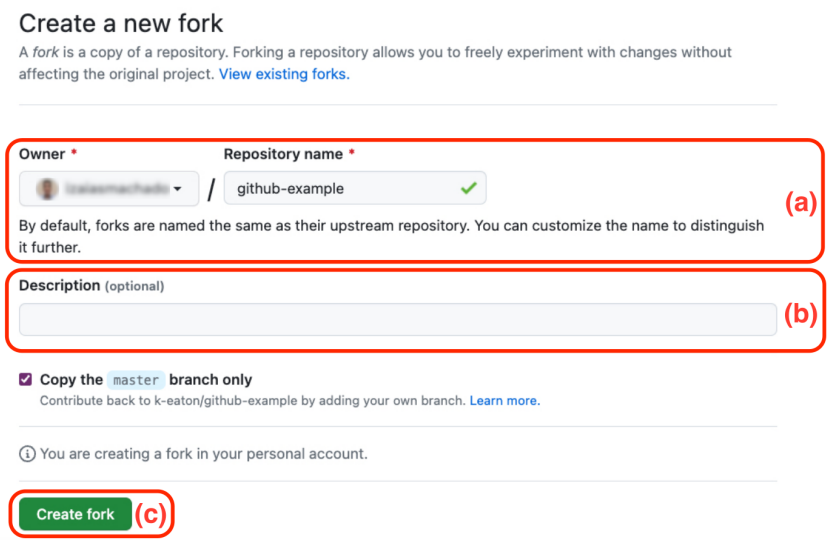


Figura 14.36: Criação de um repositório *fork*

Após criar o repositório *fork*, o resultado é mostrado na Figura 14.37. Está destacado em **(a)**, o nome do repositório e também que esse é um repositório *fork*. Nesse sentido, na seção **(a)**, o repositório exibido não é mais o repositório original, mas sim o *fork* criado a partir dele. Isso significa que o desenvolvedor fez uma cópia do repositório base em sua própria conta do GitHub. Em **(b)** existe uma barra de *status* que compara os *commits* do repositório *fork* com o repositório base. Nesse caso específico, o *status* indica que o *fork* não possui modificações em relação ao repositório base. Isso significa que o desenvolvedor ainda não fez alterações no *fork*. Na barra de *status*, também existem os botões *contribute* e *sync fork*, que serão discutidos a seguir.

O botão *contribute*, mostrado em **(b)** da Figura 14.37, é uma opção que permite ao desenvolvedor contribuir para o repositório base por meio do processo de *pull request*. Ao clicar nesse botão, o desenvolvedor é redirecionado para uma página na qual pode comparar as alterações feitas em seu *fork* com o repositório base e solicitar a incorporação dessas alterações ao projeto original. Para contribuir, é necessário fazer *commits* em seu *fork*, registrando as alterações realizadas nos arquivos do repositório.

Para finalizar a discussão da Figura 14.37, o botão *sync fork* permite que o desenvolvedor atualize o seu *fork* com as alterações mais recentes feitas no repositório base. No entanto, é importante ressaltar que essa funcionalidade só está disponível se houverem modificações no repositório base. Ao clicar no botão *sync fork*, será possível realizar uma sincronização, incorporando os *commits* mais recentes do repositório base ao repositório *fork* do desenvolvedor. Essa funcionalidade é útil quando o desenvolvedor deseja manter seu *fork* atualizado com as modificações feitas no projeto original, permitindo acompanhar o progresso e as melhorias feitas pela comunidade.

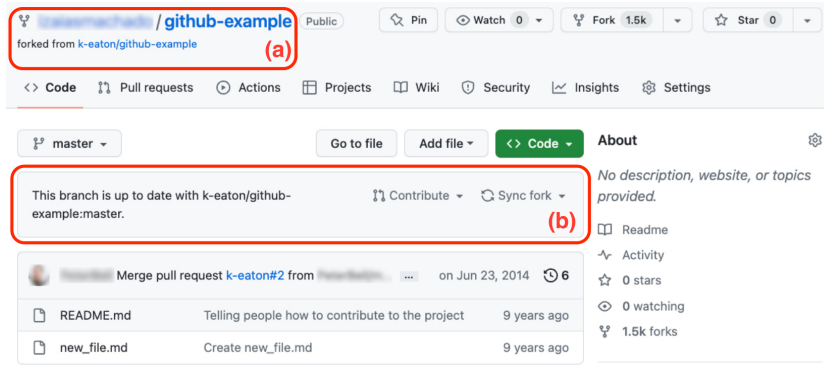


Figura 14.37: Repositório *fork* criado

A Figura 14.38 exibe uma mensagem, destacada por **(a)**, indicando que o repositório *fork* está um *commit* à frente do repositório base. Isso significa que foram feitas alterações no *fork* não incorporadas ao projeto base. No exemplo em questão, foi feito um *commit* que modifica o conteúdo do arquivo *README*, traduzindo-o para português. Anteriormente, foi discutido sobre o botão para contribuir, mas não haviam modificações para a criação de um *pull request*. Nesse caso, pelo fato de possuir um *commit* a frente do repositório base, é possível clicar no botão para contribuir e visualizar a área destacada por **(b)**. Nesta parte destacada da Figura 14.38, é informado novamente que a *branch* principal do repositório *fork* está um *commit* à frente do repositório base e existe um botão que permite abrir um novo *pull request*.

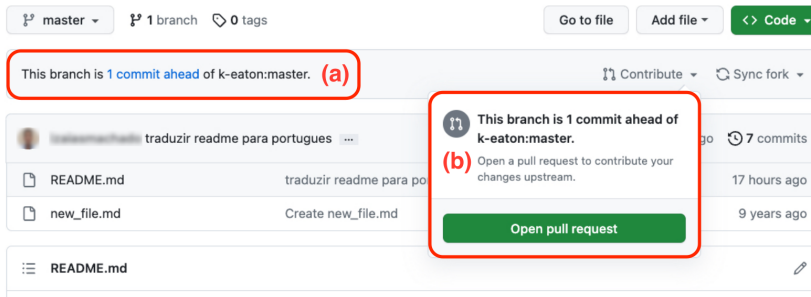


Figura 14.38: Repositório *fork* após um *commit*

Já a Figura 14.39 mostra a tela para criar um novo *pull request* do repositório *fork* para o repositório base. Nela, está destacado em vermelho o repositório de origem e a *branch* de origem, assim como o repositório de destino e a *branch* de destino. Nos campos correspondentes, é exibido o nome do repositório *fork* e sua *branch* principal que é a "master", seguido pelo nome do repositório base e sua *branch* principal "master". Essas informações são essenciais para definir a origem e o destino do *pull request*, possibilitando que as modificações feitas no repositório *fork* sejam solicitadas e incorporadas ao projeto original.

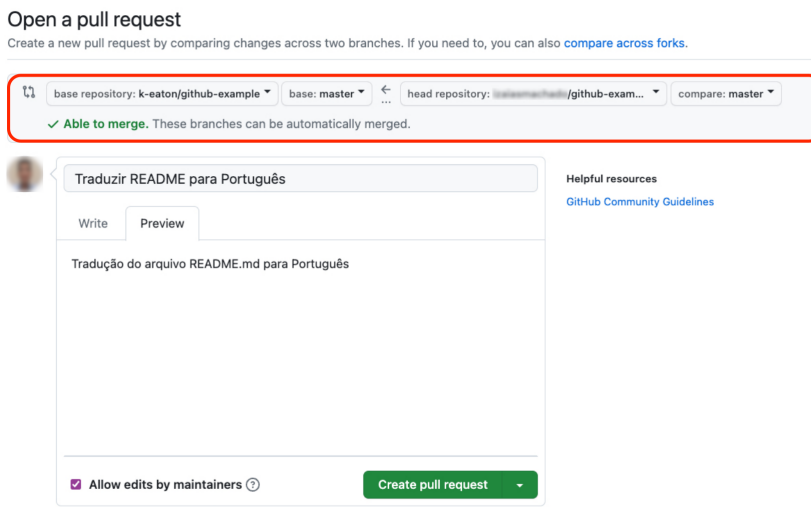


Figura 14.39: Criação do *pull request* do repositório *fork* para o repositório base

É importante ressaltar que, ao criar um *pull request* com destino o repositório base, ele será listado no repositório base, para que os colaboradores desse projeto

possam revisar e analisar as modificações propostas. Portanto, é fundamental fornecer um título descritivo para o *pull request*, juntamente com uma descrição detalhada do que foi alterado e por quê. Essa abordagem deve facilitar a identificação e compreensão das modificações pelos desenvolvedores do projeto de código aberto.

14.4 Virtualização de aplicações com Docker

Docker[11] é uma plataforma aberta que permite o desenvolvimento e implantação de aplicações em ambientes de execução chamados de *containers* (DOCKER, 2020). Esses *containers* permitem isolar uma aplicação junto de suas dependências da máquina hospedeira, garantindo que os processos da máquina não tenham acesso aos processos em execução dentro do *container*, e vice-versa. Em síntese, um *container* é uma instância isolada que contém tudo o que é necessário para a execução de um aplicativo, incluindo sistema de arquivos, dependências, configurações e o código da aplicação.

Uma das vantagens dos *containers* é serem autocontidos, ou seja, podem ser facilmente executados em diferentes ambientes sem necessidade de configurações adicionais. Isso é especialmente útil quando desenvolvedores de uma mesma aplicação utilizam sistemas operacionais diferentes. Nesse sentido, com o uso do Docker, é possível manter um ambiente virtual padrão para desenvolvimento e implantação de aplicações. Assim, diferente de ferramentas que rodam nativamente em mais de uma plataforma, a utilização de *containers* oferece uma solução consistente, evitando erros decorrentes de configurações inadequadas, pois a execução ocorre em um único ambiente virtual padronizado.

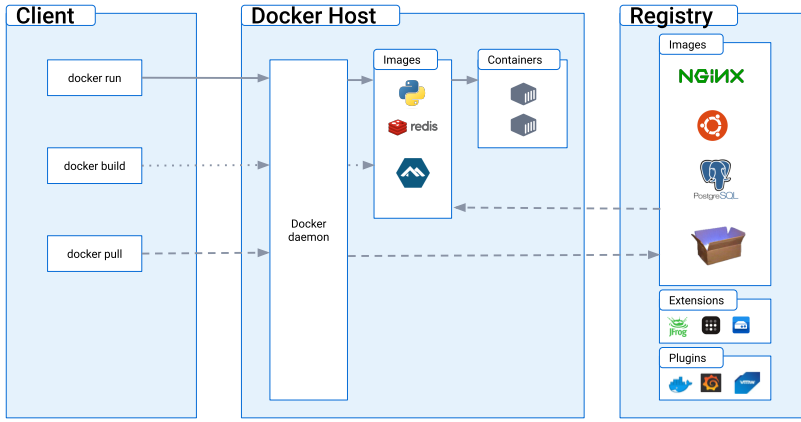
14.4.1 Conceitos básicos e arquitetura

Detalhes essenciais da arquitetura Docker são apresentados na Figura 14.40. Nessa arquitetura, o cliente, representado como *client*, desempenha um papel fundamental, pois é por meio dele que os usuários interagem com o Docker. Ele estabelece a comunicação com o *daemon*, também conhecido como *dockerd*, que é responsável por receber e processar as requisições, além de gerenciar os objetos Docker, como *containers* e imagens. O *host* mencionado na figura se refere à máquina na qual o *daemon* está em execução. É importante destacar que o cliente e o *daemon* podem ser executados em máquinas diferentes e ainda assim se comunicarem.

Além disso, na arquitetura Docker, existe o conceito de *registry*, que é uma aplicação de código aberto responsável pelo armazenamento e distribuição de imagens Docker. No contexto da Figura 14.40, um exemplo de *registry* é o Docker Hub. O Docker Hub é uma plataforma que oferece um *registry* público na qual desenvolvedores e organizações podem armazenar suas imagens Docker. Essas imagens podem ser distribuídas de forma pública, permitindo que outros usuários possam utilizá-las, ou de forma privada, restringindo o acesso apenas a usuários

[11] https://www.docker.com

autorizados. O Docker Hub é uma opção popular para compartilhar e descobrir imagens prontas para uso.



Docker (2020)

Figura 14.40: Arquitetura do Docker

Uma imagem serve como um modelo para criar um *container*, de modo que podem haver vários *containers* da mesma imagem em uma máquina. Desenvolvedores podem optar por baixar imagens públicas ou também criar suas próprias imagens. Por sua vez, um *container* é uma instância em execução de uma imagem. Após criado, a execução pode ser parada, retomada ou também o *container* pode ser deletado. O sistema de arquivos utilizado é próprio e não compartilhado com outras instâncias e é fornecido pela imagem usada como base na criação da imagem.

Conforme mencionado anteriormente, cada *container* requer uma série de configurações, como o sistema operacional e *scripts* necessários para a execução do projeto. Para criar uma imagem, é necessário descrever essas configurações em um arquivo chamado Dockerfile. O Dockerfile permite utilizar outras imagens como base para construir a nova imagem. Por exemplo, é possível usar uma imagem do sistema operacional Alpine Linux como base e adicionar os componentes e configurações específicas necessários para executar a aplicação em desenvolvimento.

14.4.2 Exemplo prático com *Dockerfile*

Para a inicialização do projeto, em um mesmo repositório são criados dois arquivos, um chamado *index.js* e outro *Dockerfile*. O primeiro, armazena o código de uma aplicação escrita em Node.js[12]. Já o outro, armazena as informações da

[12]https://nodejs.org

imagem utilizada para criar o *container* da aplicação. O conteúdo do primeiro arquivo é mostrado na Listagem 14.8 e consiste em um servidor web, que escuta requisições do tipo GET e retorna uma mensagem. Nas Linhas 4 e 5 são utilizadas variáveis de ambiente para receber, respectivamente, os valores da porta que a aplicação deve rodar e o nome do ambiente de execução. Variáveis de ambiente são parâmetros globais que podem ser acessados por todos os programas que rodam em uma máquina. Caso o valor de *env* seja *development*, na resposta da requisição GET também é enviado o valor da variável que define o ambiente de execução.

```
1  const express = require("express");
2  const app = express();
3
4  const port = process.env.PORT;
5  const env = process.env.NODE_ENV;
6
7  app.get("/", (req, res) => {
8    const response = {
9      message: "Hello World!",
10   };
11
12   if (env === "development") {
13     response.env = env;
14   }
15
16   res.json(response);
17 });
18
19 app.listen(port);
```

Listagem 14.8: Arquivo *index.js* do código do servidor Web

Na Listagem 14.9, é mostrado o *Dockerfile*, que é o arquivo de configuração da imagem dessa aplicação. A primeira linha define que a imagem utilizada como base para montar o *container* é do sistema operacional Alpine Linux que já vem com a Versão 16 do Node.js instalada. Nas Linhas 3 e 5 é definido o diretório usado para salvar os arquivos no *container* e em seguida copiado o arquivo da Listagem 14.8 para dentro do *container*. A Linha 7 define um valor padrão para a variável de ambiente que é usada no código da aplicação para definir a porta. Na Linha 9 é inicializado um arquivo de configurações do gerenciador de pacotes NPM[13] e na 10 instalada a biblioteca Express[14] utilizando o gerenciador de pacotes. O Express foi escolhido, pois é um *framework* que permite a criação de servidores web com poucas linhas de código. Por fim, a Linha 12 expõe a porta 3000, isto é, permite

[13]https://www.npmjs.com
[14]https://expressjs.com

que a porta seja acessada pela interface de rede da máquina *host* e a 14 executa código do servidor web.

```
FROM node:16-alpine

WORKDIR /app

COPY . index.js

ENV PORT=3000

RUN npm init -y
RUN npm install express

EXPOSE 3000

CMD ["node", "index.js"]
```

Listagem 14.9: Arquivo *Dockerfile*

Por meio do Terminal ou Prompt de Comando do Windows, é possível acessar o diretório onde os arquivos foram criados e executar o comando de construção da imagem, conforme exemplificado na Listagem 14.10. Esse comando utiliza a *flag* `-t` seguida pelo nome *node-api* para nomear a imagem criada a partir desse arquivo. Além disso, no final do comando, há um ponto que indica que a execução ocorreu no mesmo diretório onde estão os arquivos *Dockerfile* e da aplicação. Adicionalmente, após criar a imagem, pode-se executar o comando da Listagem 14.11, responsável por listar todas as imagens disponíveis no *host*. Além do nome das imagens, mostrado na Figura 14.41, também são exibidos uma *tag* referente à versão da imagem, um identificador único *image id*, o tempo desde a criação (*created*) e o espaço ocupado em disco (*size*).

```
docker build -t node-api .
```

Listagem 14.10: Criação da imagem por meio do *Dockerfile*

```
docker images
```

Listagem 14.11: Listar imagens disponíveis

```
→  node-api-example docker images
REPOSITORY                      TAG             IMAGE ID       CREATED         SIZE
node-api                        latest          033c16c18774   2 days ago      125MB
```

Figura 14.41: Lista de imagens disponíveis

Diante da imagem criada, o comando *run* permite instanciar o *container* a partir de uma imagem em um *host*. O comando da Listagem 14.12 é um *run* partindo da imagem *node-api* fornecendo configurações adicionais. A *flag* `--name` é utilizada para nomear o *container*, caso não seja nomeado, o *daemon* gera um nome de forma aleatória. Já a configuração `-p` mapeia a porta 3000 interna do *container* para a porta 3001 do *host*. Como já mencionado, o código do servidor web utiliza variáveis de ambiente como parâmetros e o valor de *NODE_ENV* não foi definido no arquivo *Dockerfile* da Listagem 14.9. Dito isso, o parâmetro `-e` do comando *run* define valores para variáveis de ambiente. Por fim, a *flag* `-d` é utilizada para que o *container* execute em segundo plano.

```
1 docker run --name node-api-dev -p 3001:3000
2 -e NODE_ENV=development -d node-api
```

Listagem 14.12: Instanciando *container* em ambiente simulado de desenvolvimento

O comando da Listagem 14.13 é muito similar ao da Listagem 14.12 que foi discutido, as únicas diferenças são que a porta mapeada no *host* foi a 3002 ao invés de 3001 e também que a variável de ambiente *NODE_ENV* tem o valor *production*. Desse modo, as duas instâncias explicadas são utilizadas para simular ambientes independentes de desenvolvimento e produção.

```
1 docker run --name node-api-prod -p 3002:3000
2 -e NODE_ENV=production -d node-api
```

Listagem 14.13: Instanciando *container* em ambiente simulado de produção

O comando da Listagem 14.14, com saída mostrada na Figura 14.42, lista todos os *containers* em execução no *host*. Além dos nomes definidos para cada *container*, são listados os identificadores únicos *container id*, imagem que é utilizada como base para a instância, comando utilizado para a inicialização e há quanto tempo foi criado. Por padrão, o comando lista somente as instâncias com *status up* e ao lado o tempo em que se encontra nesse estado. Ao usar o parâmetro `-a`, todos os *containers* do *host* são exibidos. No caso de mostrar todos os *containers*, alguns *status* que poderiam aparecer são o *exited* quando foi encerrado e o *paused* quando foi pausado pelo *daemon*. Por fim, ainda se referindo ao resultado de execução da Figura 14.42, é possível visualizar a faixa de portas ocupadas pelo *container* e seu nome.

```
1 docker ps
```

Listagem 14.14: Listando *containers* em execução

Com as instâncias Docker *node-api-dev* e *node-api-prod* dos ambientes de desenvolvimento e produção, podem ser feitas requisições para cada servidor web

Figura 14.42: Lista de *containers* em execução

```
→  node-api-example docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED          STATUS          PORTS                    NAMES
190b41692f35   node-api   "docker-entrypoint.s…"   41 minutes ago   Up 41 minutes   0.0.0.0:3001->3000/tcp   node-api-dev
d6d947a07746   node-api   "docker-entrypoint.s…"   41 minutes ago   Up 41 minutes   0.0.0.0:3002->3000/tcp   node-api-prod
```

e verificado se as aplicações se comportam da maneira esperada. A Figura 14.43 mostra a resposta de uma requisição do tipo GET para a porta 3001 em que o ambiente de desenvolvimento está executando. É retornado o valor de *env* que só é enviado como resposta se o ambiente for de desenvolvimento. Nesse sentido, a variável de ambiente *NODE_ENV* recebeu corretamente o valor *development*. Já na resposta da requisição da Figura 14.44 é retornado somente a mensagem e não mais o valor da variável ambiente de execução.

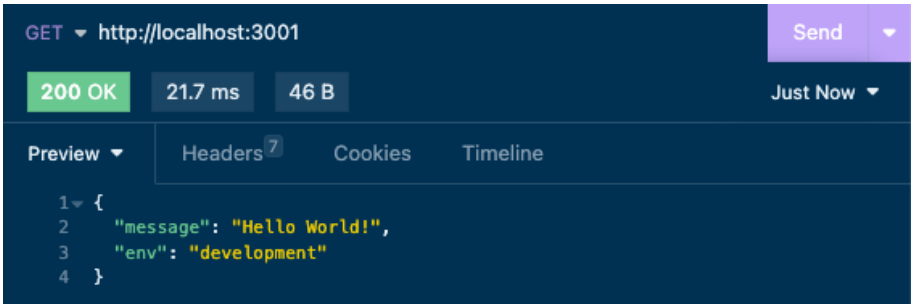


Figura 14.43: Requisição GET para *node-api-dev*

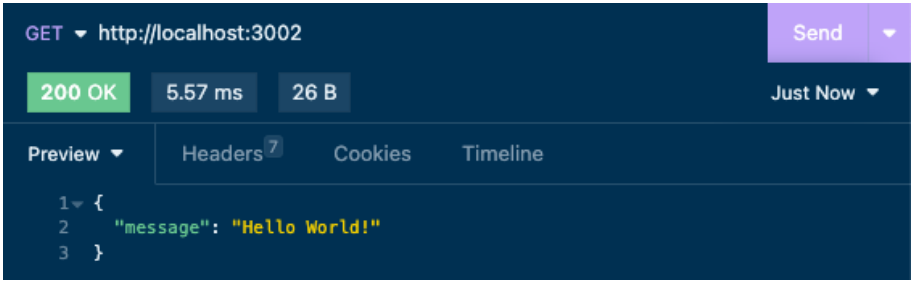


Figura 14.44: Requisição GET para *node-api-prod*

O comando *stop* pode ser usado para parar um ou mais *containers* que se encontram em execução, ou seja, *status up*. Na Listagem 14.15, o comando é utilizado para parar as duas instâncias da imagem *node-api* que estavam em execução. Após isso, as instâncias passam a ter *status stopped*. Neste caso, é utilizado o nome do *container*, mas também poderiam ser usados os identificadores únicos *container id* que são mostrados na Figura 14.42.

```
docker stop node-api-dev node-api-prod
```

Listagem 14.15: Parando os dois ambientes

Outrossim, com o intuito de voltar a executar um ou mais *containers* pausados, pode ser utilizado o comando *start* para que os *status* passem de *paused* para *up* novamente. Sabendo que esse comando funciona de forma similar ao *stop*, o comando *start* é exemplificado na Listagem 14.16, em que é retomada a execução do *container* de produção.

```
docker start node-api-prod
```

Listagem 14.16: Retomando a execução do ambiente de produção

Já para deletar *containers* e imagens, são utilizados, respectivamente, os comandos *rm* e *rmi*. A Listagem 14.17 exemplifica a remoção da instância de desenvolvimento que já estava parada. Para que um *container* seja removido da forma exemplificada, é necessário que se encontre pausado. De mesmo modo, para deletar uma imagem de uma máquina é necessário que todas as instâncias da imagem sejam removidas. Neste sentido, é necessário primeiro pausar e depois remover o *container* de produção para que o comando da Listagem 14.18 seja executado com sucesso.

```
docker rm node-api-dev
```

Listagem 14.17: Remoção da instância de desenvolvimento

```
docker rmi node-api-image
```

Listagem 14.18: Remoção da imagem

14.5 WSO2 Identity Server: Gerenciamento de identidade e controle de acesso

O WSO2 Identity Server é uma ferramenta que facilita o acesso seguro a aplicativos utilizando um login único (WSO2, 2023e; KARUNARATHNA; KARUNARATNE, 2017). Por meio dessa solução, organizações conseguem gerenciar de forma eficiente identidades de seus usuários. Uma das vantagens do uso deste servidor de identidade é que, por ser de código aberto, oferece flexibilidade e personalização para atender as necessidades de cada empresa. Os dois protocolos suportados para autenticação são o OpenID Connect e o OAuth 2.0. Além disso, o próprio WSO2 Identity Server também fornece conectores já prontos para ligar o

WSO2 a bancos de dados com credenciais de usuários. Isso simplifica o processo de integração com sistemas já existentes.

Para servidores de identidade, recurso é a combinação de dados que podem ser acessados ou ações que podem ser realizadas em uma aplicação (AUTH0, 2023). Adicionalmente, autenticação é definida como o processo de verificação de identidade do usuário. Quanto ao termo autorização, é utilizado no contexto de verificar se o usuário pode ter acesso a um recurso protegido dentro de uma aplicação. Além disso, a autorização pode também ser empregada no contexto de delegar quais recursos do usuário uma aplicação tem acesso.

14.5.1 Instalação utilizando Docker

Para agilizar o processo de instalação do WSO2 API Manager, pode ser executado o comando da Listagem 14.19 para subir um *container* Docker. Com isso, a imagem é baixada, executada e em alguns instantes a aplicação fica disponível no endereço `https://localhost:9443/carbon`. Como não foi feita nenhuma configuração de usuário, o login padrão é com usuário e senha sendo *admin*.

```
1 docker run -d -p 8280:8280 -p 8243:8243 -p 9443:9443
2 --name api-manager wso2/wso2am:latest
```

Listagem 14.19: Comando para instalação do WSO2 API Manager utilizando Docker

Ao entrar com as credenciais, é possível visualizar a tela inicial do API Manager, que é apresentada na Figura 14.45. Na imagem, foram destacadas em vermelho as partes **(a)** e **(b)** para discussão. Em **(a)**, existe uma barra lateral por meio do qual se navega nos menus de configuração do API Manager. Já em **(b)**, são mostradas informações relativas ao ambiente que o *container* está rodando, dois exemplos dessas informações são o endereço da máquina e também tempo que o sistema está rodando.

14.5.2 Configurar a conexão com uma aplicação

Qualquer entidade que fornece serviços web é chamada por um provedor de identidade de *service provider* (WSO2, 2023b). Nesse sentido, para que uma aplicação use o WSO2 Identity Server é necessário realizar o cadastro do *service provider*. Por conseguinte, por meio do menu lateral apresentado previamente, é possível clicar na opção *add* para adicionar um novo *service provider*. Com isso, é acessado uma página para realizar o cadastro do novo *service provider*. Um fragmento dessa página é mostrado pela Figura 14.46. Nessa tela, deve ser escolhido um nome para o *service provider* e depois clicado para registrar.

Após o cadastro, o usuário é redirecionado para uma tela de configuração do *service provider*, um fragmento dessa página é mostrado na Figura 14.47. Porém, a imagem mostra o acesso ao menu *Inbound Authentication Configuration* e o menu

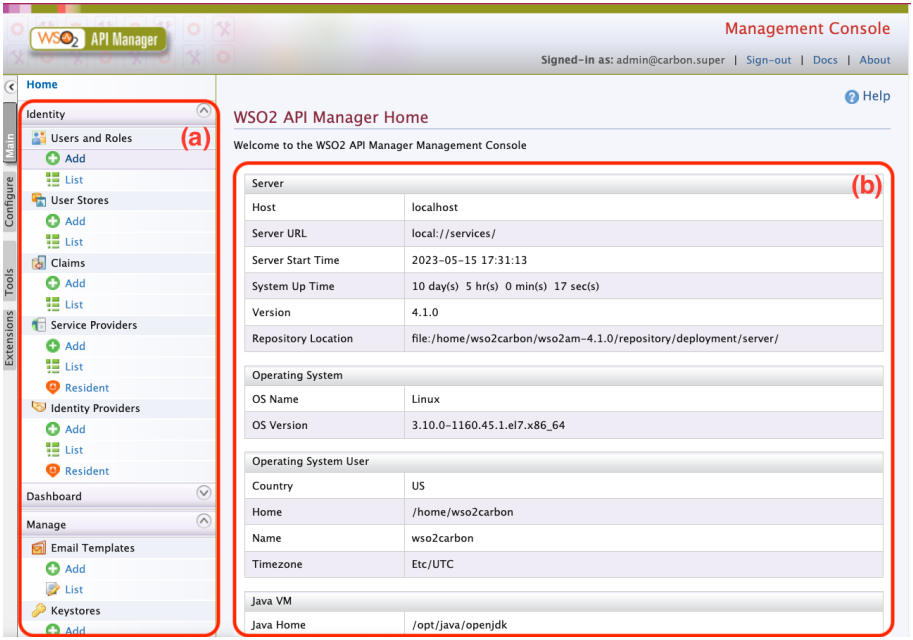


Figura 14.45: Tela inicial do WSO2 API Manager

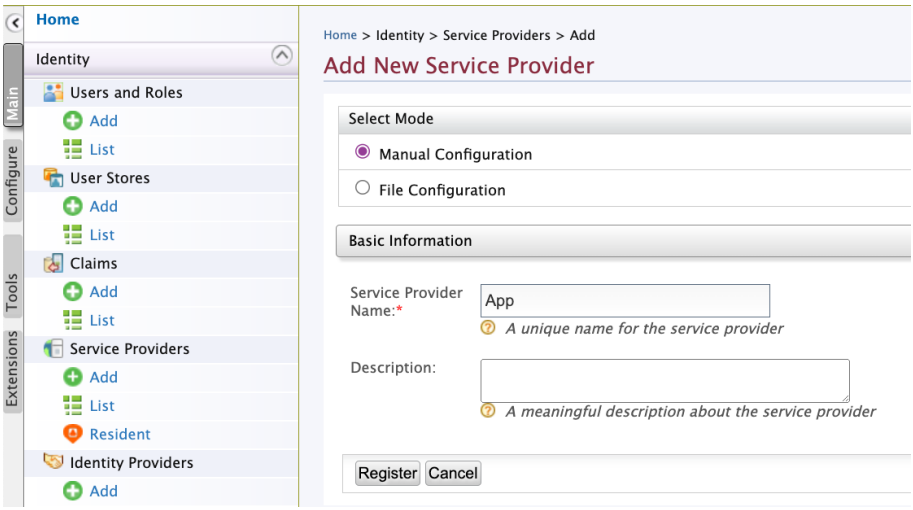


Figura 14.46: Registro de *service provider*

aninhado *OAuth/OpenID Connect Configuration*. Na imagem, são mostradas as

credenciais *client_id* e *client_secret* do *service provider*. Ainda, na Figura 14.47, ao clicar *edit*, pode ser acessado as configurações de conexão do protocolo OIDC.

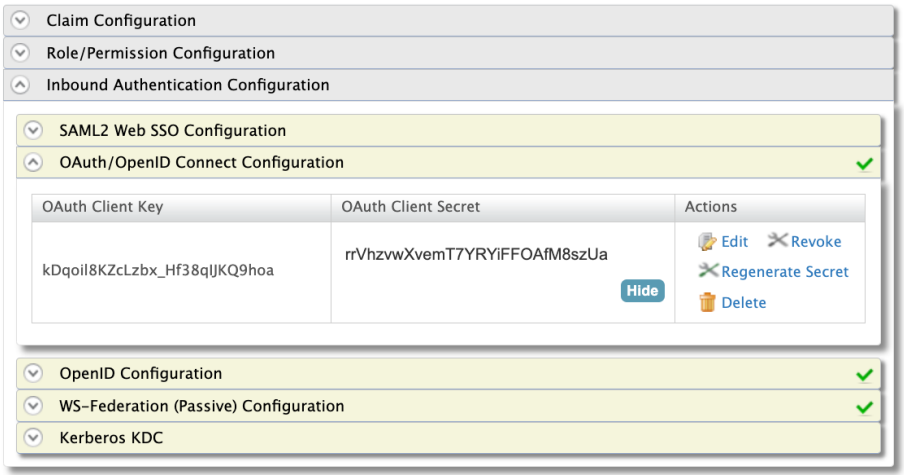


Figura 14.47: Fragmento da tela de configuração do *service provider*

Um recorte da página de configurações de conexão do protocolo OIDC é mostrado na Figura 14.48. A configuração que será necessária consiste em marcar a versão do protocolo OAuth como *code* e preencher uma URL de *callback* com uma rota para o *front-end* da aplicação.

14.5.3 Conectando a uma aplicação

A Figura 14.49 é um digrama que visa ilustrar o fluxo de autenticação e autorização *authorization code* (WSO2, 2023a). Na imagem, são apresentadas as entidades usuário, aplicação cliente, servidor de autorização e também o navegador do usuário. Além disso, existem setas numeradas que ligam as entidades e cada seta descreve uma etapa no fluxo. As ações que são realizadas por meio de um navegador tem as setas que as descrevem passando pelo navegador do usuário na ilustração. Isto posto, as ações 4 e 5 não são feitas por meio de um navegador, pois suas setas não passam por essa entidade na ilustração.

No âmbito da Figura 14.49, em 1 o usuário acessa a aplicação por meio de um navegador. Já na seta 2, a aplicação cliente requisita ao servidor de autorização acesso aos dados do usuário. Para obter acesso aos dados do usuário é necessário que o usuário esteja autenticado. Por isso, na etapa 3 o usuário digita suas credenciais para se autenticar em uma tela de login do servidor de autorização. Na etapa 3, o usuário também autoriza acesso da aplicação aos recursos requisitados por meio de uma tela de consentimento. Caso a autenticação e a autorização sejam bem sucedidas, a aplicação recebe um parâmetro *code* que é apresentado pela seta 4.

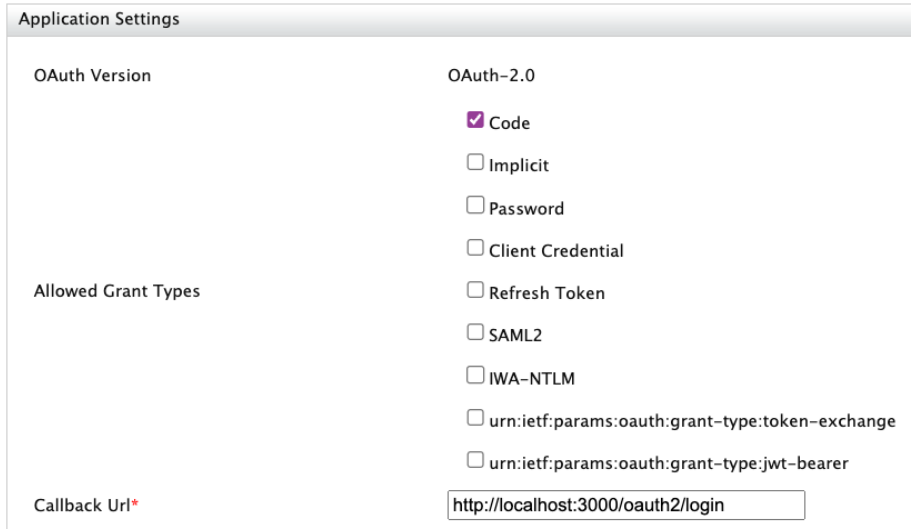


Figura 14.48: Menu de configuração do protocolo *OpenID Connect*

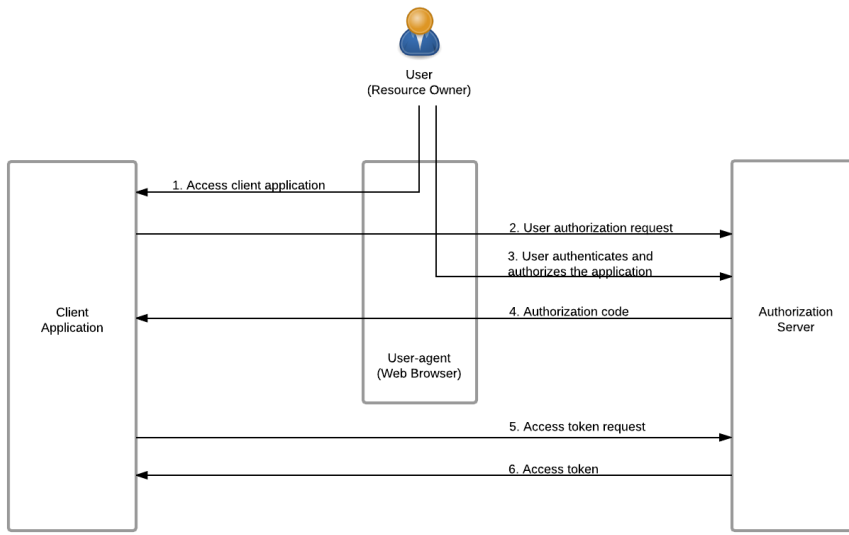
Tabela 14.1: Requisições HTTP do fluxo *authorization code*

Requisição	Método	Rota	Descrição
1	GET	/oauth2/authorize	Utilizado para acessar a tela de login
2	POST	/oauth2/token	Solicitar o token de acesso
3	POST	/oauth2/userinfo	Requisitar informações do usuário

Sendo assim, o parâmetro recebido é utilizado para solicitar um *token* de acesso, como descrito por 5. Por fim, caso o parâmetro *code* seja válido, o servidor de autorização retorna um *token* de acesso, mostrado em 6.

Os processos realizados no diagrama da Figura 14.49 são feitos por meio de requisições HTTP que devem ser feitas diretamente no navegador, por meio de redirecionamentos, ou também por requisições AJAX. A Tabela 14.1 apresenta as requisições que devem ser feitas, o método HTTP utilizado, a rota que é empregada e também uma breve descrição. As requisições dessa tabela são discutidas a seguir.

Com as credenciais do *service provider*, é possível montar uma URL que dá acesso a página de login do WSO2, utilizando a rota da requisição 1 da Tabela 14.1. A URL de login deve ser acessada por meio de um navegador qualquer e, inicialmente, o login pode ser feito usando o mesmo usuário utilizado para entrar no painel de configuração do WSO2 API Manager. A montagem da URL de login é ilustrada pela Figura 14.50, em que o *scope* é relativo ao escopo de dados que se quer obter após a finalização do login. Já o *client_id*, é uma parte das credenciais da aplicação.



WSO2 (2023a)

Figura 14.49: Ilustração do fluxo *authorization code*

A *redirect_uri* é a URL de redirecionamento, que deve ser a mesma fornecida na configuração do *service provider*, descrita na Seção 14.5.2. Por fim, o *response_type* refere-se ao formato da resposta esperada pela aplicação cliente, no caso do fluxo *authorization code* a resposta esperada após o login é o *code*.

Ao acessar por meio de um navegador a URL de login montada, o usuário é redirecionado para a página de login, mostrada na Figura 14.51. Após entrar com as credenciais e consentir com o acesso do escopo de dados solicitado pela aplicação, o usuário é redirecionado para a *redirect_uri*, que é uma rota dentro do *front-end* da aplicação.

A Listagem 14.20 mostra o formato da URL do redirecionamento feito após o login, em que o parâmetro *url-redirecionamento* destacado se refere ao *redirect_uri*. Além disso, também é mostrado que o *front-end* da aplicação recebe por meio da URL os parâmetros *code* e *session_state*.

```
1 <url-redirecionamento>?code=<code>&
2 session_state=<session-state>
```

Listagem 14.20: Formato da URL de retorno após login com WSO2

É ilustrada na Figura 14.52 a requisição 2 da Tabela 14.1, que é feita utilizando o parâmetro *code*, as credenciais do *service provider*, *redirect_uri* e *grant_type*. O

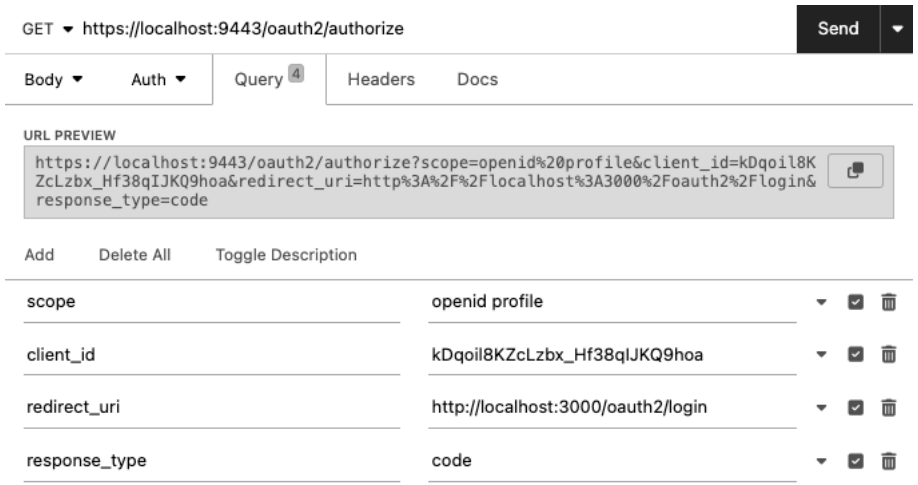


Figura 14.50: Montagem da URL de login

grant_type define o fluxo de autenticação utilizado. Nesse caso, o fluxo de autenticação é o *authorization_code*, em que se tem um *code* e a partir dele é requisitado um *token* de acesso. Os parâmetros dessa requisição são passados na forma de *Form URL Encoded*. Sendo assim, ao passar os parâmetros pela URL é necessário também enviar o cabeçalho `Content-Type: application/x-www-form-urlencoded`. Após a requisição, o *code* enviado não pode ser utilizado novamente. Em contrapartida, o *token* de acesso recebido como resposta da requisição, que é apresentado na Figura 14.52 com nome *access_token*, pode ser utilizado várias vezes e sua duração é definida pelo servidor de autorização.

A requisição 3 da Tabela 14.1 é importante, pois além de mostrar os dados do usuário, é por meio desta que a aplicação consegue saber se o usuário está devidamente autenticado, essa requisição é apresentada na Figura 14.53. Para ser realizada, é necessário passar um cabeçalho de nome *Authorization* e valor `Bearer <access-token>`. A resposta pode ser maior ou menor dependendo do escopo de dados solicitado durante o processo login (WSO2, 2023c). No caso da Figura 14.53, a resposta da requisição foi somente o nome de usuário. Se a requisição falhar porque o *token* de acesso está inválido, é necessário fazer o login novamente e conseguir um novo *token* de acesso. Já caso falhe porque o *token* se encontra expirado, é possível implementar o fluxo opcional de *refresh_token* para que a aplicação receba um novo *token* válido sem que o usuário tenha que entrar novamente na aplicação (WSO2, 2023d).

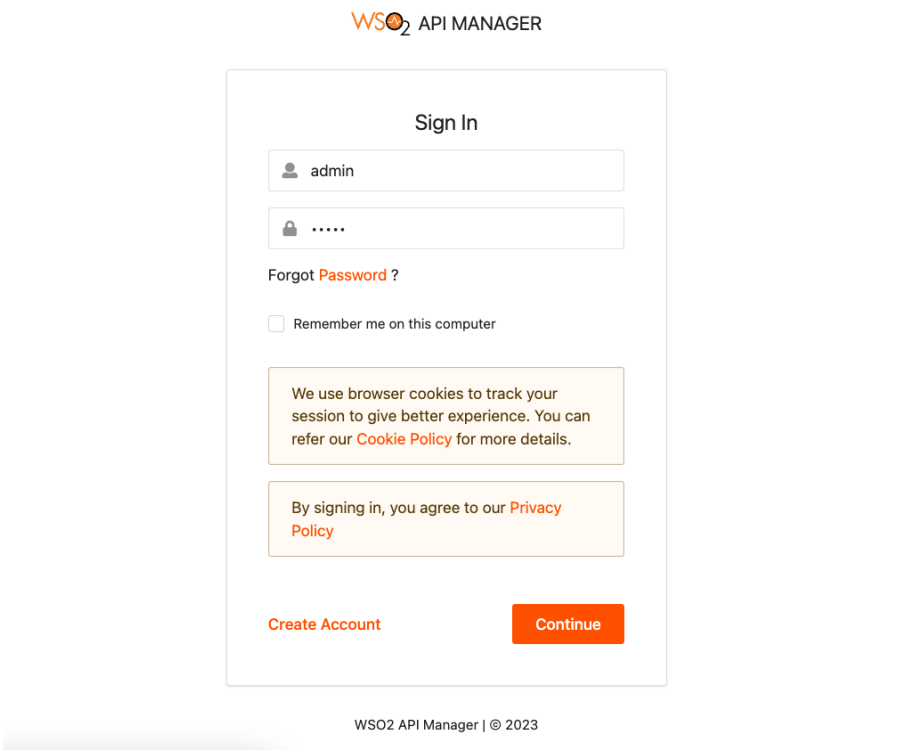


Figura 14.51: Página de login da aplicação

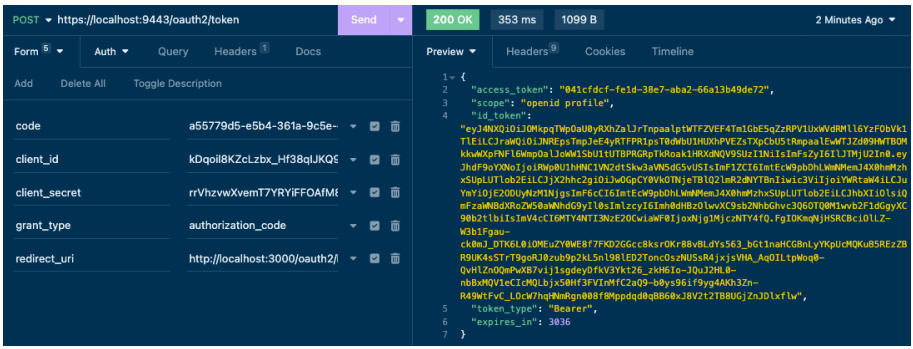


Figura 14.52: Requisição do token de acesso

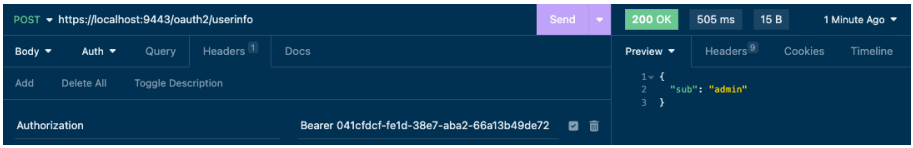


Figura 14.53: Requisição de informações do usuário

Capítulo 15

As Principais Hard Skills de um Desenvolvedor para web Atual

Maxela Martins Pontes (UFC)

Johnatan Alves Oliveira (UFLA)

Jermana Lopes de Moraes (UFC)

Fischer Ferreira (UNIFEI)

A dinâmica do mercado de trabalho, especialmente no contexto do desenvolvimento web, caracteriza-se pela busca incessante por profissionais altamente qualificados e com competências cada vez mais sofisticadas. Nesse cenário, a formação e a qualificação tornam-se desafios essenciais para a criação e consolidação de habilidades técnicas e estratégicas. Para aqueles que desejam ingressar na área, a ampla variedade de tecnologias disponíveis pode gerar dúvidas sobre quais são as mais relevantes e quais devem ser priorizadas.

Paralelamente, a complexidade do software está em constante crescimento, impulsionada pelo surgimento de novas tecnologias, pela exigência de maior segurança e pela necessidade de atender a padrões rigorosos de qualidade. Além disso, o dinamismo do mercado impõe desafios adicionais, como prazos cada vez mais curtos e a pressão por redução de custos, exigindo dos desenvolvedores não apenas conhecimento técnico aprofundado, mas também habilidades de adaptação, gestão de tempo e resolução de problemas de forma eficiente.

Desta forma, este capítulo apresenta um estudo quantitativo sobre os dados extraídos da plataforma *LinkedIn*, visando uma análise das *hard skills* demandadas no cenário de desenvolvimento web. O foco central foi explorar a base de dados fornecida pelo *LinkedIn*, que abrange as vagas frequentemente disponibilizadas para desenvolvedores web. Para atingir esse propósito, a metodologia utilizou a utilização da plataforma *Power Automate* [1] da Microsoft uma ferramenta gratuita, reconhecida como uma ferramenta eficaz de automação robótica de processos. A aplicação desta ferramenta permitiu a extração sistemática de dados relacionados às vagas para desenvolvimento web no *LinkedIn*, por meio da criação de duas etapas distintas.

A primeiro etapa foi responsável pela extração dos links relacionados aos cargos de *front-end*, *back-end*, *full stack* e *Quality Assurance (QA)*. Em seguida, a segunda etapa foi dedicada à extração das competências presentes nos links obtidos na primeira etapa. Após a conclusão da extração, foi realizada a limpeza, organização e contagem dos dados extraídos, sendo essas etapas conduzidas pelo uso do *MATLAB*[2], visando fornecer uma análise estatística descritiva das *hard skills* demandadas no mercado de desenvolvimento web.

Foram sumarizado quais são as principais tecnologias utilizadas para *front-end*, *back-end*, *full stack* e QA, organizado da seguinte forma: a Seção 15.1 apresenta uma introdução a *Hard Skills*. As Seções 15.2 e 15.3 apresentam algumas competências relacionadas ao desenvolvimento web e as ferramentas utilizadas para realizar este estudo. Ainda, A Seção 15.4 apresenta os passos realizados para conduzir o estudo. Finalmente, a Seção 15.5 apresenta os resultados alcançados listando as principais *hard skills* encontradas.

[1] https://learn.microsoft.com/pt-br/power-automate/desktop-flows/install
[2] https://www.mathworks.com/products/matlab.htm

15.1 Introdução as *Hard Skills*

O conceito de competência, é pensado como conjunto de conhecimentos, habilidades e atitudes (isto é, conjunto de capacidades humanas) que justificam um alto desempenho, acreditando-se que os melhores desempenhos estão fundamentados na inteligência e personalidade dos indivíduos (FLEURY; FLEURY, 2001). Em essência, a competência é percebida como conjunto de recursos que o indivíduo detém.

No contexto de desenvolvimento web, os programadores são contratados com base nas competências que possuem e que são pertinentes ao cargo. No campo do desenvolvimento de software, essas competências técnicas são denominadas *hard skills*. O termo *hard skills* pode ser compreendido de acordo com sua tradução para o português; enquanto "skills"se refere a "competências"ou "habilidades", o termo "hard"é interpretado como "duro". Essa combinação de palavras cria uma metáfora para descrever as habilidades técnicas de um profissional, ou seja, conhecimentos que podem ser mensurados e comprovados por meio de testes e certificações (PORTALPOS, 2022).

Essas habilidades técnicas compreendem os fundamentos teóricos e a experiência prática que um indivíduo deve possuir para desempenhar uma tarefa específica com sucesso (AHMED et al., 2012). No mundo do desenvolvimento de software, desenvolvedores necessitam possuir *hard skills*, que envolvem conhecimento técnico em áreas como linguagens de programação, ferramentas de *software* e *frameworks* (OLIVEIRA et al., 2023). Ainda, essas competências são essenciais para a realização eficaz das tarefas diárias de um engenheiro de software, sendo avaliadas por meio de entrevistas técnicas, testes de codificação e contribuições do desenvolvedor para projetos de código aberto.

As vagas de emprego na área de desenvolvimento de software geralmente dividem as habilidades e competências em duas categorias, *hard skills* e *soft skills* (AHMED et al., 2012). As *hard skills* e *soft skills* são elementos cruciais no processo de contratação de desenvolvedores de software (OLIVEIRA et al., 2023). O termo *hard skills* refere-se a habilidades técnicas e específicas, geralmente mensuráveis e adquiridas por meio de treinamento, educação formal ou experiência prática. Estas são habilidades tangíveis e específicas relacionadas a tarefas ou funções específicas. Em contraste com as *soft skills* (habilidades interpessoais e comportamentais), as *hard skills* são mais facilmente quantificáveis e podem ser demonstradas por meio de testes práticos ou certificações.

Os recrutadores acreditam que as habilidades *soft* complementam as *hard skills*. Embora as *soft skills* desempenhem um papel relevante na escolha de candidatos, o foco específico deste capítulo está nas *hard skills*. Nesse contexto, a identificação e a compreensão das *hard skills* são vitais para este estudo. Porém, as habilidades técnicas frequentemente exigidas para desenvolvedores web são numerosas, tornando inviável para o programador dominar todas as habilidades exigidas no mercado de trabalho. Portanto, é necessário que o desenvolvedor conheça bem as principais *hard skills* do cargo em que deseja atuar, pois as empresas modernas de TI alocam

desenvolvedores para trabalhar em áreas específicas, como banco de dados, segurança, interface do usuário (design front-end), recursos principais (design back-end), desenvolvimento móvel, entre outras (MONTANDON et al., 2021).

Uma das plataformas mais destacadas em temas relacionados à carreira, como busca por emprego, práticas de seleção e recrutamento, e decisões de carreira é o *LinkedIn* [3] (LEVINE; ALEY, 2020). Nesta plataforma, os profissionais podem explorar vagas de emprego de maneira abrangente, filtrando suas buscas conforme a função desejada na empresa, a localização geográfica e as preferências quanto ao formato de trabalho, seja remoto, híbrido ou presencial. O *LinkedIn* oferece uma interface intuitiva e personalizada, permitindo aos usuários encontrar oportunidades alinhadas com suas aspirações profissionais e necessidades específicas, tornando-se assim uma ferramenta valiosa na busca por empregos na área de desenvolvimento web. Com isso, o *LinkedIn* proporciona uma base de dados que pode ser analisada para indicar as *hard skills* mais requisitadas para o desenvolvedor web atual.

15.2 Competências Relacionados ao Desenvolvimento Web

15.2.1 Desenvolvedor *Front-End*

O profissional *front-end* é aquele que domina as tecnologias essenciais para proporcionar uma experiência de navegação fluida, intuitiva e agradável ao usuário. Em resumo, quem atua nessa área deve possuir habilidades tanto na construção do *layout* quanto na implementação de recursos interativos, garantindo que os elementos da interface sejam visualmente atraentes e funcionalmente eficazes.

Na área de *front-end*, é crucial dominar diversas ferramentas, cada uma desempenhando funções específicas. O HTML, por exemplo, é empregado para a marcação, estruturação e organização do conteúdo das páginas web, garantindo que os elementos essenciais sejam corretamente dispostos para aprimorar a experiência do usuário. O CSS, por sua vez, permite estilizar as páginas, aplicando cores, tipografias, espaçamentos e efeitos visuais, além de proporcionar responsividade, tornando o design adaptável a diferentes dispositivos e tamanhos de tela.

Uma tecnologia indispensável para o desenvolvimento *front-end* é o *JavaScript*. Essa linguagem de programação permite adicionar interatividade às páginas web, possibilitando a criação de animações, manipulação dinâmica do DOM, comunicação assíncrona com servidores (AJAX e *fetch API*), além da integração com bibliotecas e *frameworks* como *React* e *Angular*, que facilitam o desenvolvimento de interfaces mais sofisticadas e dinâmicas.

Além do domínio técnico, um desenvolvedor *front-end* precisa ter habilidades analíticas para solucionar problemas de usabilidade, otimizar o desempenho das aplicações e garantir acessibilidade digital. A atenção aos detalhes e a capacidade

[3] https://br.linkedin.com/

de acompanhar tendências tecnológicas são diferenciais importantes para criar experiências inovadoras e eficientes na web (SILVA, 2023b).

15.2.2 Desenvolvedor Back-End

O desenvolvedor *back-end* desempenha um papel fundamental na construção da infraestrutura técnica de aplicações web, sendo responsável pelo processamento de dados, implementação de regras de negócio e integração entre sistemas. Seu trabalho garante que as funcionalidades de um site ou aplicativo operem corretamente, proporcionando um desempenho eficiente e seguro. Para isso, ele precisa compreender minuciosamente os objetivos do projeto e desenvolver soluções robustas que englobem gerenciamento de banco de dados, otimização de processamento, escalabilidade, segurança e alta disponibilidade (BRAGA, 2022).

Para atuar nessa área, é essencial que o profissional *back-end* tenha domínio de linguagens de programação como *Python*, *PHP*, *JavaScript (Node.js)*, *Java* e *C#*, além de conhecimento aprofundado em bancos de dados relacionais (*MySQL* e *PostgreSQL*) e não relacionais (*MongoDB* e *Redis*). Ele também deve ser proficiente no gerenciamento de APIs (*REST* e *GraphQL*), garantindo a comunicação eficiente entre diferentes partes da aplicação e serviços externos.

Além das competências técnicas, a capacidade de depuração e resolução de problemas é indispensável, uma vez que falhas e bugs são inevitáveis e exigem correções rápidas e eficazes. A segurança da informação também é uma preocupação constante, demandando conhecimento sobre práticas como criptografia, controle de acesso e proteção contra ataques cibernéticos.

Outro fator essencial é a colaboração entre o desenvolvedor *back-end* e o desenvolvedor *front-end*. A sinergia entre esses profissionais é crucial para garantir que os sistemas funcionem de forma integrada, resultando em aplicações modernas, ágeis e inovadoras. Um *back-end* bem estruturado permite que o *front-end* entregue uma experiência fluida e responsiva ao usuário, assegurando o sucesso do projeto (BRAGA, 2022).

15.2.3 *Desenvolvedor Full Stack*

Os desenvolvedores *back-end* e *front-end* são colaborativos, mas segundo Braga (2022) existe um profissional versátil capaz de desempenhar ambas as funções. O desenvolvedor *full stack* é um programador com uma abordagem abrangente, competente tanto na criação da interface gráfica gráfica quanto na sustentação da infraestrutura do software, garantindo o funcionamento completo da aplicação (BRAGA, 2022).

O desenvolvedor *full stack* destaca-se por sua habilidade de atuar em todas as fases do desenvolvimento, incluindo a construção de aplicações *mobile*, o que representa uma vantagem significativa. Sua versatilidade permite que ele acompanhe projetos desde a concepção até a manutenção, supervisionando tanto a experiência do usuário quanto o processamento de dados no servidor. Para desempenhar essas

funções, é essencial que o *full stack* domine as responsabilidades do *back-end*, como gerenciamento de banco de dados, configuração de servidores e implementação de lógicas de negócio, enquanto mantém um conhecimento sólido em linguagens *front-end*, como *HTML*, *CSS* e *JavaScript*. Além disso, o uso de *frameworks* modernos, como *Angular* e *Django*, otimiza o desenvolvimento e melhora a eficiência do trabalho (BRAGA, 2022).

Uma pesquisa realizada pela Associação Catarinense de Tecnologia ACATE (2022) projetou a abertura de mais de 16,6 mil novas vagas na área de tecnologia em Santa Catarina até 2023. As funções mais demandadas incluem desenvolvedores *full stack*, *back-end* e *front-end*. O levantamento revelou que os desenvolvedores *full stack* representam 23% da demanda total, evidenciando a crescente valorização desse perfil. Além disso, os profissionais especializados em *back-end* e *front-end* também estão em alta no mercado, com uma procura de 18% e 12%, respectivamente (ACATE, 2022).

As competências mais valorizadas pelos empregadores incluem domínio de metodologias ágeis, experiência prática em desenvolvimento de software, habilidade na execução de projetos e proficiência em múltiplas linguagens de programação. Essa análise auxilia na formulação de cursos voltados à capacitação de profissionais, alinhando-se às necessidades específicas das empresas e destacando as áreas com maior escassez de talentos qualificados (ACATE, 2022).

15.2.4 Desenvolvedor em Garantia de Qualidade

Definir qualidade de software não é uma tarefa simples, pois o conceito abrange diversos fatores interdependentes que impactam diretamente a experiência e a satisfação dos usuários. No contexto do desenvolvimento de software, a qualidade está associada a um conjunto de atributos essenciais que garantem o funcionamento eficaz, a confiabilidade e a usabilidade da solução, atendendo tanto às necessidades do negócio quanto às expectativas dos usuários (DESENVOLVEDOR, 2021).

Na área de tecnologia, o profissional de *Quality Assurance* (QA) desempenha um papel fundamental ao assegurar que produtos e serviços atendam aos padrões de qualidade estabelecidos (DIO.ME, 2023). Ter um especialista dedicado a essa função é essencial para identificar falhas e comunicar melhorias aos desenvolvedores, contribuindo para um software mais estável, seguro e eficiente. Um controle de qualidade bem estruturado não apenas reduz o risco de erros em produção, mas também aumenta a competitividade do produto no mercado e pode impulsionar suas vendas. Além disso, o profissional de QA tem uma visão ampla do produto, o que lhe permite sugerir aprimoramentos que vão além da simples detecção de falhas (SILVA, 2023a).

De acordo com qamadness (2021), algumas das principais habilidades técnicas (*hard skills*) essenciais para um profissional de QA incluem:

- Compreensão do ciclo de vida do desenvolvimento de software;
- Conhecimento dos tipos, metodologias e técnicas de teste de software;

- Experiência com ferramentas de rastreamento de bugs e gerenciamento de testes;
- Familiaridade com linguagens de programação e boas práticas de automação de testes;
- Capacidade de planejar e executar diferentes tipos de testes para garantir a qualidade do software.

O domínio dessas competências permite que o profissional de QA atue de forma estratégica, garantindo que os sistemas sejam entregues com alta qualidade, reduzindo custos com correções e aumentando a satisfação dos usuários.

15.3 Ferramentas utilizadas no estudo

15.3.1 *Linkedin*

As redes sociais têm se consolidado como ferramentas fundamentais na construção de conexões profissionais e na busca por oportunidades de emprego (LEVINE; ALEY, 2020). Com o avanço da Internet, o recrutamento, a procura por trabalho e a construção de redes profissionais tornaram-se processos cada vez mais digitais, sendo amplamente utilizados tanto por empresas quanto por candidatos. Além disso, jovens profissionais e estudantes recorrem a serviços de redes profissionais (*Professional Networking Services* – PNS), que são plataformas sociais voltadas para negócios e empregabilidade, com o objetivo de planejar e desenvolver suas carreiras. Essas redes têm como propósito auxiliar na gestão da identidade profissional, na ampliação do *networking* e no acesso a conhecimentos e oportunidades relevantes para o crescimento profissional (MA; LEUNG, 2019).

Uma pesquisa realizada em 2017 por Ma e Leung (2019) indicou que o *LinkedIn* se destacava entre as principais redes profissionais, contando com mais de 467 milhões de usuários na época. Atualmente, a plataforma possui mais de 1 bilhão de usuários em 200 países e regiões ao redor do mundo, com 257 milhões na Europa e 233 milhões na América do Norte (LINKEDIN, 2023a). De acordo com Pena et al. (2022), o *LinkedIn* é reconhecido como a principal rede social profissional, oferecendo aos usuários a possibilidade de fortalecer sua presença digital, promover sua imagem profissional e expandir conexões estratégicas no mercado de trabalho.

Dessa forma, o *LinkedIn* tornou-se uma plataforma essencial para empresas que buscam divulgar vagas e identificar novos talentos, clientes e fornecedores. Aproximadamente 90% dos recrutadores utilizam a rede regularmente para analisar perfis e encontrar candidatos adequados (MA; LEUNG, 2019). Além disso, um estudo da *Statistic Brain* revelou que cerca de 122 milhões de pessoas conseguiram uma entrevista por meio da plataforma, sendo que 35,5 milhões foram contratadas graças às conexões estabelecidas no site (NOSSOMEIO, 2022).

O *LinkedIn* também se destaca como um exemplo notável de *big data*, pois reúne um volume massivo de informações sobre profissionais e empresas, fornecendo

insights valiosos para recrutadores e tomadores de decisão (BRADBURY, 2011). Um perfil bem elaborado na plataforma inclui dados sobre cargos, experiências profissionais, formação acadêmica e área de atuação, permitindo que empresas e candidatos se conectem de forma mais eficiente (BRADBURY, 2011). Além disso, a plataforma oferece recursos avançados, como um campo de pesquisa na página inicial e uma seção exclusiva para vagas, facilitando a busca e candidatura a oportunidades alinhadas às competências e interesses dos usuários (LINKEDIN, 2023b).

15.3.2 *Microsoft Power*

A plataforma *Microsoft Power* integra quatro tecnologias avançadas da Microsoft (*Power BI*, *Power Apps*, *Power Automate* e *Power Virtual Agents*) em um ecossistema unificado e altamente funcional. Esse conjunto de ferramentas capacita os usuários a desenvolver soluções personalizadas, automatizar processos, analisar dados e criar agentes virtuais de forma eficiente e inovadora (MICROSOFT, 2023a).

Segundo Microsoft (2023a), essas plataformas adotam uma abordagem *low-code*, permitindo o desenvolvimento de software com pouca ou nenhuma necessidade de codificação. Dessa forma, usuários sem experiência em programação podem criar aplicativos personalizados de maneira intuitiva, enquanto desenvolvedores experientes conseguem otimizar e acelerar a implementação de soluções complexas.

Essas ferramentas se destacam por oferecer uma interface visual intuitiva, conectores para diversos serviços de *back-end* e um sistema integrado de gerenciamento do ciclo de vida do aplicativo. Isso possibilita a criação, depuração, implantação e manutenção ágeis, garantindo maior eficiência e flexibilidade no desenvolvimento de aplicações empresariais (MICROSOFT, 2023a).

15.3.3 Automação com Power Automate

Os fluxos de trabalho no Microsoft Power Automate permitem a automação da comunicação entre serviços modernos (APIs) baseados em nuvem e sistemas legados (MICROSOFT, 2023c). Com uma interface intuitiva, a ferramenta facilita a conexão entre aplicativos populares e a criação de fluxos automatizados por meio de modelos, conectores e ações pré-definidas (PEARSON et al., 2020). Além disso, os usuários podem utilizar gravadores de área de trabalho e web para capturar interações, como cliques e digitação, e incorporá-las diretamente ao designer visual (MICROSOFT, 2023c).

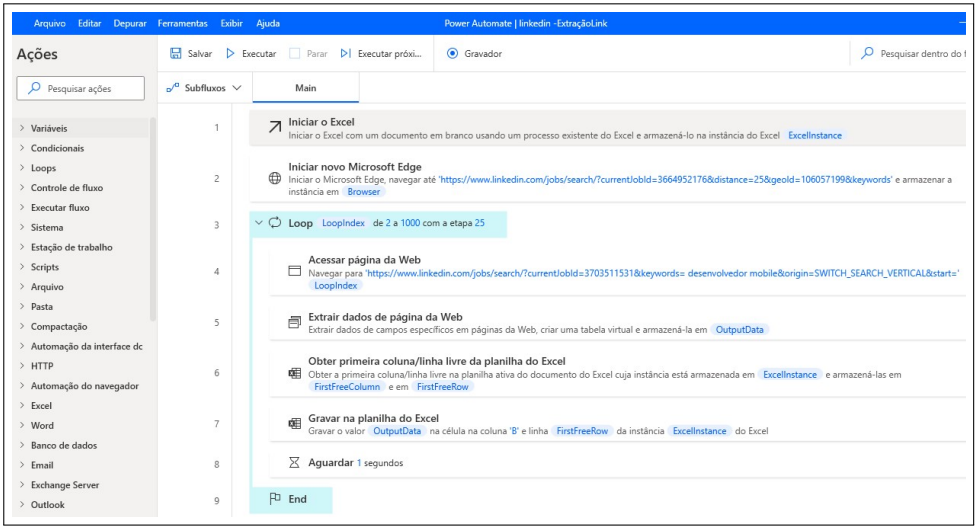
Os conectores no *Microsoft Power Automate* viabilizam a integração com serviços como *SharePoint*, *Office 365*, *Outlook*, *OneDrive* e redes sociais, permitindo a execução de operações automatizadas mediante permissões específicas (HAFENSCHERER et al., 2023). Essa funcionalidade transforma o conector em um intermediário que gerencia fluxos de trabalho sem necessidade de codificação (PEARSON et al., 2020).

A ferramenta também utiliza Robotic Process Automation (RPA), permitindo a automação de tarefas repetitivas sem intervenção humana. Com um designer

visual intuitivo e centenas de ações predefinidas, a RPA possibilita a criação de uma força de trabalho digital capaz de interagir com diversos sistemas para executar operações como copiar e colar, extrair dados, processar e-mails e conectar-se a APIs (ANYWHERE, 2024). Apesar de suas funcionalidades abrangentes, a configuração de automações é relativamente simples.

A Figura 15.1 apresenta o painel de ações, que ocupa o lado esquerdo do designer de fluxo e exibe todas as ações disponíveis no fluxo da área de trabalho. Além disso, é possível observar o passo a passo do processo de extração de dados.

Figura 15.1: Automação Power Automate



15.4 Configuração do estudo

Esta seção apresenta a configuração do estudo utilizada para encontrar as principais *Skills* segundo a oferta de vagas de emprego disponibilizadas no *LinkedIn*.

As postagens de emprego disponíveis na plataforma *LinkedIn* foram extraídas ao longo de quatro dias, coletando diariamente os dados relevantes para a identificação das vagas disponíveis no repositório no *GitHub*. Cada dia de extração foi dedicado a uma função específica, abrangendo *front-end*, *back-end*, *full stack* e QA. Inicialmente, foi a coleta dos dados feita em duas etapas, conforme demostrando nas Seções 15.4.1 e 15.4.2

15.4.1 Extração de dados Etapa 1

Para a mineração de dados no *LinkedIn*, diversos passos foram realizadas. Inicialmente, elaborou-se um fluxo na plataforma *Power Automate* versão 2401.2 para a extração dos links de vagas de emprego disponíveis no *LinkedIn*, conforme detalhado na Figura 15.2.

No *Power Automate*, todas as operações são executadas por meio de ações. Primeiramente, selecionou-se a ação que cria um documento *Excel* em branco para registrar os links extraídos. Em seguida, configurou-se o navegador com os parâmetros de entrada necessários para acessar o *LinkedIn*. Após garantir o acesso ao *LinkedIn*, adicionou-se uma ação para acessar a página da web com informações sobre as vagas de emprego desejadas.

A próxima etapa envolveu a configuração de um loop para incorporar as demais ações necessárias para a automação do acesso às páginas. O loop foi configurado para iniciar em 25, considerando que o botão na página, que dava acesso a todas as vagas, tinha o padrão de 25. Assim, a configuração de entrada variou de 25 a 1000, acessando cada 25 vagas. Essa abordagem garantiu a cobertura completa das vagas disponíveis. Entretanto, apesar de ser estabelecido um limite de 1000 links, a ferramenta permitiu a extração de apenas 274 links por vez.

Posteriormente, implementou-se uma ação específica para extrair informações da página da web. Com a janela em primeiro plano, o processo percorreu os botões em um limite de tempo de 60 segundos, extraindo os links e os nomes de cada vaga. Em seguida, utilizou-se outra ação para identificar a primeira coluna e linha livres na planilha do *Excel* para registrar os dados extraídos. Após isso, uma ação adicional foi executada para gravar os links na planilha, iniciando da coluna B, com um intervalo de 1 segundo para otimizar a gravação. Finalizada a fase de extração, uma última ação no *Excel* foi acionada para salvar o arquivo no caminho configurado, seguindo os parâmetros previamente estabelecidos. Segue o processo feito na ferramenta:

1. Iniciar o *Excel*.
2. Iniciar a navegação e adicionar o link da página do *LinkedIn*.
3. Dentro do loop, foram incorporadas as seguintes etapas:
    - Passo 1: Início do loop de 25 a 1000 com incremento de 25.
    - Passo 2: Acesso à página da web com a vaga desejada.
    - Passo 3: Extração de dados de campos específicos nas páginas da web.
    - Passo 4: Aquisição da primeira coluna livre e da linha na planilha do *Excel*.
    - Passo 5: Registro dos dados na planilha do *Excel*.
    - Passo 6: Aguardo de 1 segundo.

- Passo 7: Salvamento na planilha do *Excel*.

Essas etapas foram executadas sequencialmente dentro do loop para atingir os objetivos pretendidos. Conforme Figura 15.2.

15.4.2 Extração de dados Etapa 2

Na segunda etapa, o processo inicia-se com a adição do arquivo *Excel* extraído na Etapa 1. A ação do *navegador* é então acionada para acessar a página do *LinkedIn*, seguida da pressão da tecla *F11* para permitir uma extração mais eficiente dos dados, abrangendo todos os elementos visíveis na tela. Um *loop* é configurado para iniciar na segunda linha da planilha e prosseguir até o limite de 274, conforme permitido pela ferramenta. Dentro desse *loop*, são realizadas a leitura do arquivo *Excel* para examinar cada célula e a ação de acesso à página web. Ao entrar na página, são incluídas ações para obter detalhes da página e gravar no arquivo *"meta.txt"*.

Após essas ações, uma novo passo é inserida no fluxo, onde uma ação adicional registra o título de cada vaga, contendo o nome da empresa, com um intervalo de 1 segundo para cada gravação. Esse procedimento permite não apenas contar quantas vagas foram extraídas, mas também identificar as empresas que postaram essas vagas, fornecendo informações valiosas sobre a fonte dos dados e a distribuição das oportunidades no *LinkedIn*.

Em seguida, a ação *"enviar tecla"* é utilizada para enviar o comando *PgDn* ou *Next*, rolando a página até o ponto desejado. A ação de mover o *mouse* para o texto na tela (*OCR*) é então acionada, com parâmetros específicos, buscando a palavra-chave *"falta"*. O *mouse* clica na palavra *"falta"* e aguarda a aparição das competências faltantes na tela, extraindo as competências necessárias para aquela vaga. Posteriormente, os detalhes da página acessada são obtidos, e a extração dos textos é realizada. Uma ação é adicionada para gravar o texto extraído no arquivo *log.txt*, com um intervalo de 1 segundo para evitar a detecção pelo sistema de segurança. Dessa forma, são extraídas a descrição das vagas e as competências exigidas por cada vaga, concluindo assim o processo de extração. Conforme Figura 2:

1. Inicialização do *Excel*, com os links extraídos do *LinkedIn* na etapa anterior.
2. *Abertura* do *navegador*.
3. Envio da tecla *F11* para janelas em primeiro plano.
4. Dentro do *loop*, foram incorporadas as seguintes etapas:
    - Passo 1: Início do *loop* de 2 a 274 com incremento de 1.
    - Passo 2: Leitura da planilha do *Excel* a partir da linha B.
    - Passo 3: Acesso à página da web.

- Passo 4: Coleta de detalhes da página.
- Passo 5: Gravação de texto em arquivo.
- Passo 6: Aguardo de 1 segundo.
- Passo 7: Movimentação do cursor do *mouse* para o texto na tela (*OCR*), configurado para continuar até a palavra "falta"ser encontrada.
- Passo 8: Aguardo pelo texto na tela, com o texto a ser localizado como "detalhes".
- Passo 9: Coleta de detalhes adicionais da página da web.
- Passo 10: Armazenamento do texto em arquivo.
- Passo 11: Aguardo de 1 segundo.

5. Fechamento do Microsoft Excel.

A Figura 15.2 apresenta a sequência de ações realizadas no processo de coleta de dados.

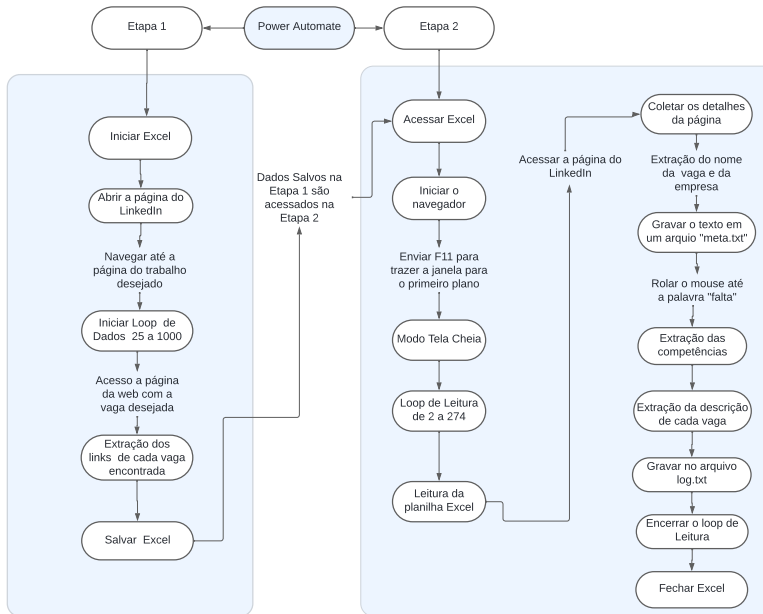


Figura 15.2: Fluxograma de extração dos dados

15.4.3 Tratamento dos Dados Extraídos

Após a extração dos dados, foram gerados os arquivos *meta.txt*, que contém o nome da vaga e da empresa, e *log.txt*, que compila as competências e os textos de cada vaga extraídos. Na Figura 15.3, é ilustrada a etapa subsequente à extração, onde foram realizados processos de limpeza, organização e contagem dos dados utilizando *MATLAB*. A escolha do *MATLAB* foi motivada pela familiaridade pré-existente do pesquisador com a ferramenta, que permite, com poucas linhas de código, a efetiva limpeza e organização dos dados.

O código-fonte e os dados utilizados nesta sessão estão disponíveis no repositório no GitHub: ⟨https://github.com/MaxelaMartins/Datasets_TCC_Maxela/tree/main/Scripts%20Matlab⟩.

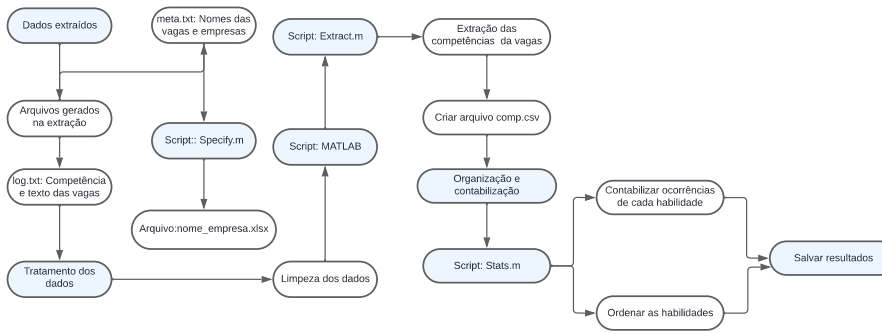


Figura 15.3: Fluxograma de Tratamento dos Dados no MATLAB

15.4.4 Limpeza dos Dados

Durante o processo de extração de dados, foram enfrentadas interrupções devido à presença de anúncios nas vagas. Para resolver essas questões, foram realizadas os seguintes passos:

1. Identificação e exclusão das linhas correspondentes aos anúncios.
2. Remoção de links inválidos.
3. Criação de s*scripts* para a limpeza e organização dos dados.

15.4.5 Organização dos Dados

Com os arquivos gerados na extração, iniciamos a organização dos dados utilizando três scripts *MATLAB*:

1. *Specify.m*: Processa o arquivo *meta.txt* para extrair e organizar os nomes das vagas e empresas em um arquivo *Excel* chamado *name.xlsx*.

2. *Extract.m*: Processa o arquivo *log.txt* para extrair as competências faltantes e criar um arquivo *CSV* (*comp.csv*) com essas competências.

3. *Stats.m*: Extrai estatísticas do arquivo *comp.xlsx*, contabiliza as ocorrências de cada requisito e salva os resultados em um novo arquivo *Excel*.

15.4.6 Contabilização dos Dados

No script *Stats.m*, foi realizada a contabilização das competências:

1. O arquivo *comp.xlsx* foi aberto e lido.

2. As ocorrências de cada requisito foram contabilizadas.

3. As competências foram ordenadas em ordem decrescente com base no número de repetições, e os resultados foram salvos em um novo arquivo *Excel*.

15.5 As principais *Hard Skills* encontradas

Esta seção apresenta as *hard skills*, obtidos a partir de dados do *LinkedIn*, mais requisitadas para o desenvolvimento web. Adicionalmente, a seção inclui uma discussão sobre os dados coletados e suas implicações para o mercado de trabalho, oferecendo recomendações sobre as *hard skills* essenciais para iniciantes no desenvolvimento web. Por fim, são ressaltadas as principais lições aprendidas ao longo do processo de análise, enfatizando a importância dessas competências no cenário atual e contribuindo para a formação de futuros profissionais na área.

15.5.1 Análise das Hard Skills

Como resultado dessa abordagem, foram extraídas 794 postagens de emprego, com distribuição de 259 vagas para *front-end*, 264 vagas para *back-end*, 267 vagas para *full stack* e 271 vagas para Garantia de Qualidade (*Quality Assurance*) QA. Em seguida, foram coletadas informações sobre a descrição da vaga que cada empresa fornecia. Esses dados estão disponíveis em no repositório do livro.

Durante a análise do conjunto de dados original, identificou-se a necessidade de uma redução do conjunto de dados alcançados, devido à baixa frequência de alguns atributos, com o objetivo de aprimorar a análise. Observou-se que a frequência das habilidades seguia um padrão de *cauda longa*, em que poucas habilidades eram responsáveis pela maioria das ocorrências, enquanto a maior parte era mencionada apenas esporadicamente. Por essa razão, selecionamos apenas as habilidades que apresentavam no mínimo 10 ocorrências, removendo aquelas com menor frequência.

Foram coletadas as competências disponíveis para cada posição, resultando na contabilização das *hard skills* de quatro cargos principais: *front-end*, *back-end*, *full stack* e QA. Por meio da redução dos dados alcançados, foram selecionadas 94 *hard Skill* e 5 *soft Skills* reportadas nesta seção. Ainda, foram identificado 2.771 menções das habilidades nas vagas coletadas.

15.5.2 *Hard Skills* para *Front-end*

A Figura 15.4 apresenta o quantitativo de ocorrências das principais habilidades demandadas no desenvolvimento web, com foco específico no contexto de *front-end*. Assim, foram encontradas 259 postagens com total de 506 menções às habilidades, ou seja, as postagens pode ter reportado mais de uma habilidade por vaga. Como pode ser observado na Figura 15.4 as linguagens de programação mais citadas foram:

- JavaScript (64 ocorrências de 259 postagens, cerca de 24%): Destaca-se como a linguagem de maior relevância no desenvolvimento *Front-end*, sendo fundamental para a criação de interfaces interativas e dinâmicas na web. O *JavaScript* é amplamente utilizado em diversas aplicações, refletindo sua centralidade no desenvolvimento web moderno.
- CSS (43 ocorrências, cerca de 16%): Fundamental para a estilização e layout de páginas web.
- HTML (33 ocorrências, cerca de 12,7%): Essencial para a estruturação de páginas web.
- Java (32 ocorrências, cerca de 12,3%): Continua a ser uma linguagem versátil, utilizada em várias aplicações, apesar de sua presença menor em comparação ao *JavaScript*.
- TypeScript (22 ocorrências, cerca de 8%): Valorizada por adicionar tipagem estática ao *JavaScript*, melhorando a segurança e a eficiência no desenvolvimento.

Ainda, foi observado outras competências voltadas para o *front-end* tais como: definição de requisitos (33 ocorrências, cerca de 12,7%), Garantia de Qualidade (22 ocorrências, cerca de 8%) e Resolução de problemas voltados para o *front-end* (17 ocorrências, cerca de 6%) Além disso, alguns *frameworks* importantes para o desenvolvimento de *front-end* foram mencionados, tais como: *React.js* (28 ocorrências, 10,8%) e *Angular* (26 ocorrências, 10%).

O domínio do *JavaScript* no *Front-end* reforça sua importância como uma habilidade essencial para desenvolvedores focados na criação de experiências de usuário ricas e eficientes. Destacando as tecnologias e práticas mais valorizadas no mercado.

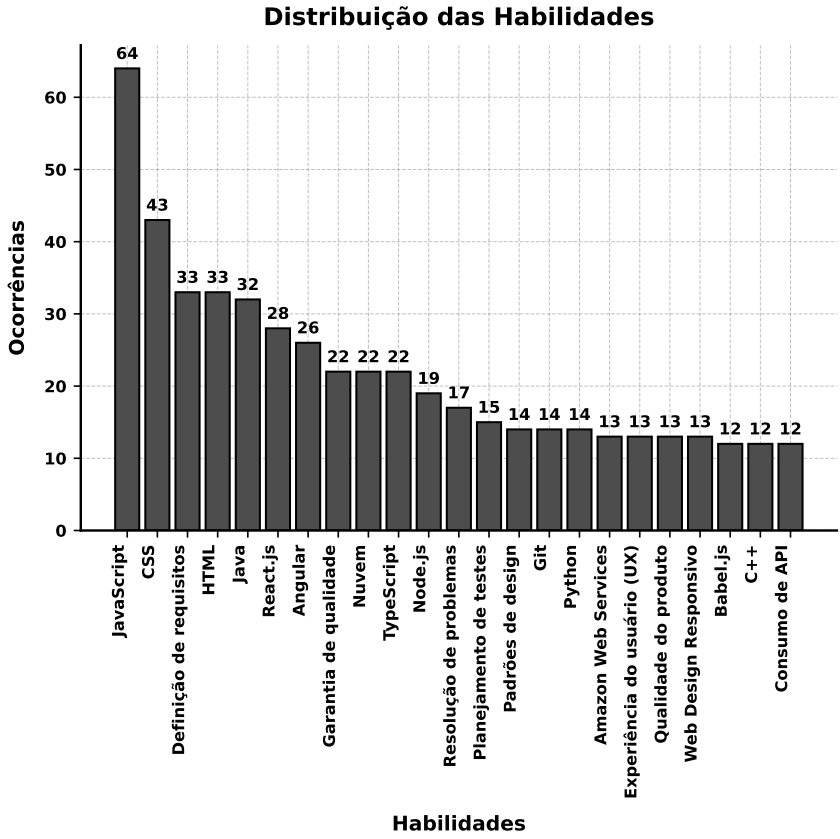


Figura 15.4: Gráfico de Habilidades sobre *Front End*

15.5.3 *Hard Skills* para *Back-end*

A Figura 15.5 apresenta as habilidades essenciais para o desenvolvimento *back-end*, destacando a importância de competências relacionadas à gestão de servidores, bancos de dados e à lógica de negócios que sustenta as aplicações web. Foram encontradas o total de 264 vagas de emprego para *back-end* sendo que foram 574 menções as habilidades relacionadas a *back-end*, como demonstrado na Figura 15.5. No desenvolvimento *back-end*, as seguintes linguagens se destacam:

- Java (96 ocorrências de um total de 264 vagas analisadas, cerca de 36%): Lidera com ampla vantagem, refletindo sua robustez e versatilidade no desenvolvimento de aplicações web complexas.
- JavaScript (61 ocorrências, cerca de 23%): Continua sendo uma habilidade essencial, evidenciando a integração crescente entre *front-end* e *back-end*.

- Python (40 ocorrências, cerca de 15%): Ganha destaque por sua popularidade no desenvolvimento de software e análise de dados.
- SQL (36 ocorrências, 13%): Reafirma sua importância no gerenciamento de dados em bancos de dados relacionais.
- CSS (29 ocorrências, cerca de 10%): Embora tradicionalmente associado ao *front-end*, o CSS também é relevante no *back-end* para a definição de estilos e layout em aplicativos que exigem integração estreita entre a interface do usuário e o servidor.

Ainda, a Figura 15.5 demonstra outras habilidades além do conhecimento em linguagens de programação, entre as principais: conhecimento sobre definição de requisitos e Microsserviços (26 ocorrências, com cerca de 9%). Ainda, conhecimento sobre API e *Amazon Web Service* com 23 (8,6%) e 29 (10%) ocorrências, respectivamente.

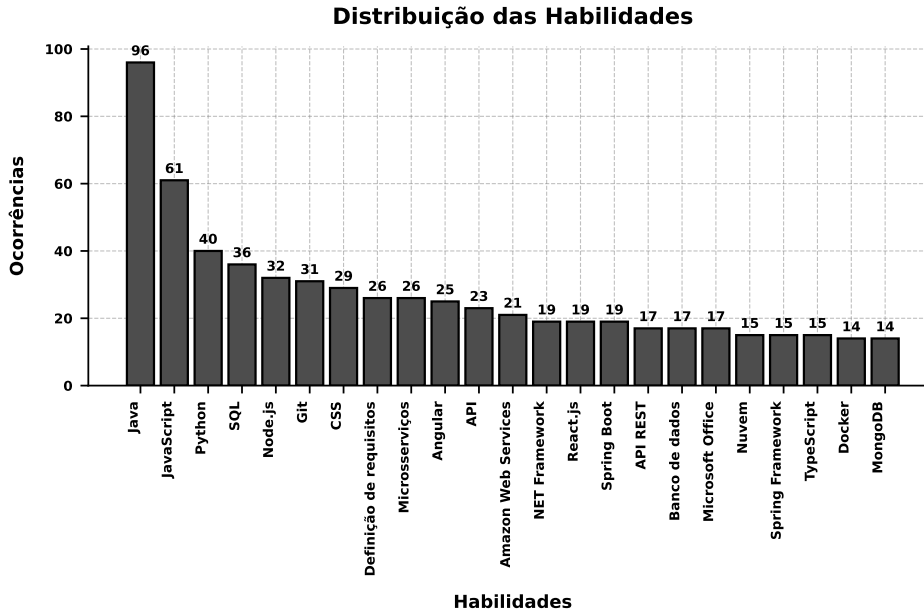


Figura 15.5: Gráfico de Habilidades Back-end

15.5.4 *Hard Skills* para *Full Stack*

A Figura 15.6 apresenta o levantamento das habilidades relacionadas aos profissionais *Full Stack* estas habilidades demonstram a versalidade que um desenvolvedor

deve dominar para transitar entre o *front-end* e o *back-end*. Por meio do levantamento feito, relatados na Seção 15.4, foram encontradas 267 ocorrências relacionadas a vagas para desenvolvedor para *Full Stack* com o total de 639 menções a habilidades relatadas nas vagas. As habilidades mais requisitadas revela uma necessidade de proficiência em múltiplas linguagens, sendo as mais citadas:

- Java (96 ocorrências de um total de 13,6%, cerca de 35%): Destaca-se como a habilidade mais requisitada, refletindo sua versatilidade tanto no *front-end* quanto no *back-end*.
- JavaScript (61 ocorrências, com cerca 22%): Evidencia sua centralidade na construção de aplicações web integradas.
- Python (40 ocorrências com cerca de 14%): Muito utilizado para vagas de Ciência dos dados.
- SQL (36 ocorrências, com cerca de 13%): Essencial para a gestão eficiente de dados.
- Node.js (32 ocorrências, com cerca de 11,9%): Utilizado para *back-end*.

Ainda, como pode ser observado na Figura 15.6 conhecimento sobre banco de dados teve 44 ocorrências (cerca de 16,4%), novamente foi citado o conhecimento do padrão arquitetural Microsserviços (42 ocorrências, cerca de 15%), API (24 ocorrências, cerca de 8,9 %). Ainda, foram citados os conhecimentos em Estrutura de dados, revisão de código, padrões de design e Padrões de codificação, 14 (5,2%), 13 (4,8%), 13 e 11 (4,1%) ocorrências, respectivamente.

15.5.5 Qualidade de Software

A Figura 15.7 apresenta as ocorrências de habilidades nas vagas analisadas que mencionavam Garantia de Qualidade (QA). Foram encontradas 271 vagas com o total de 528 ocorrências. Assim, as principais ocorrências foram a resolução de problemas de qualidade, doutrina QA, depuração, revisão de código e automação de testes, com as quantidade de ocorrências: 64 (23%), 51 (18,8%), 49 (18%), 42 (15%) e 39 (14%), respectivamente.

Adicionalmente, *frameworks* como *Angular* e *React.js* mostraram-se mais predominantes no *Front End*, refletindo sua importância na criação de aplicações web modernas e responsivas. Por outro lado, tecnologias emergentes como *DevOps* foram fortemente associadas à área de QA, indicando uma tendência crescente de integração entre o desenvolvimento e as operações para melhorar a eficiência e a qualidade dos processos de entrega de software.

Esses resultados fornecem uma visão clara das habilidades mais valorizadas no mercado de desenvolvimento web atual, permitindo uma melhor compreensão das exigências de cada área específica e auxiliando no direcionamento das estratégias de formação de desenvolvedores.

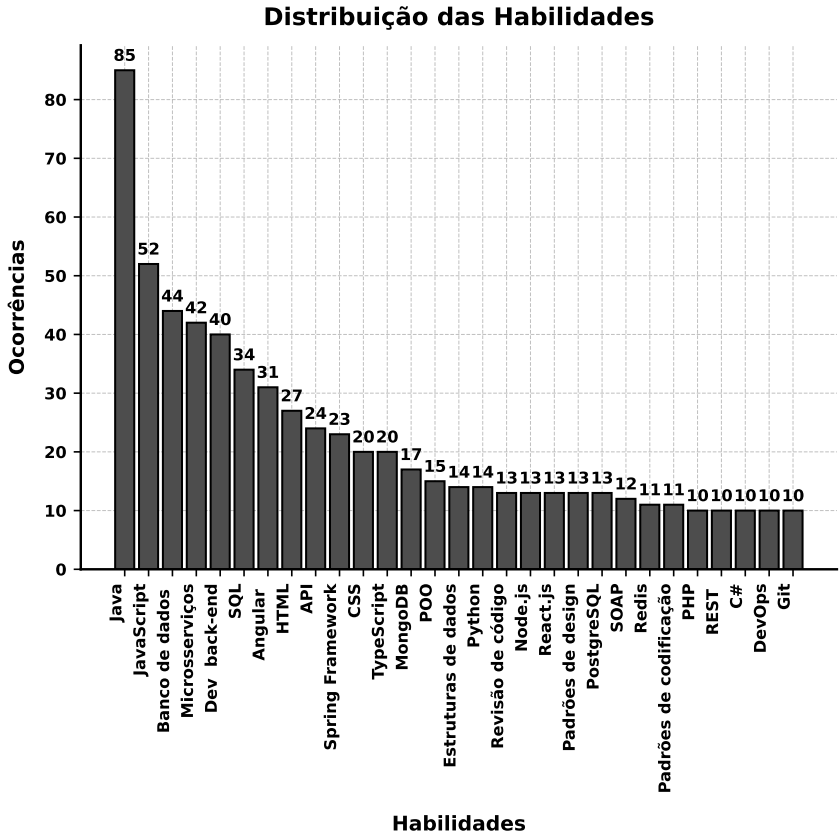


Figura 15.6: Gráfico Habilidades Full Stack

15.6 Análise das Soft Skills

Embora a análise principal deste estudo se concentre nas *hard skills* necessárias para o desenvolvimento web, é importante considerar as *soft skills*, que também são relevantes no ambiente profissional. A análise das vagas de emprego extraídas do *LinkedIn* mostra que a proficiência em *inglês* é a *soft skill* mais comum e aparece frequentemente nos perfis de *front-end*, *back-end*, *full Stack* e *Garantia de qualidade* (QA). O *inglês* é importante para acessar documentação técnica, comunicar-se com equipes internacionais e acompanhar tendências globais.

Além do *inglês*, outras soft skills mencionadas incluem:

- *Trabalho em Equipe*: Capacidade de colaborar efetivamente em projetos, essencial para a coordenação entre membros da equipe.

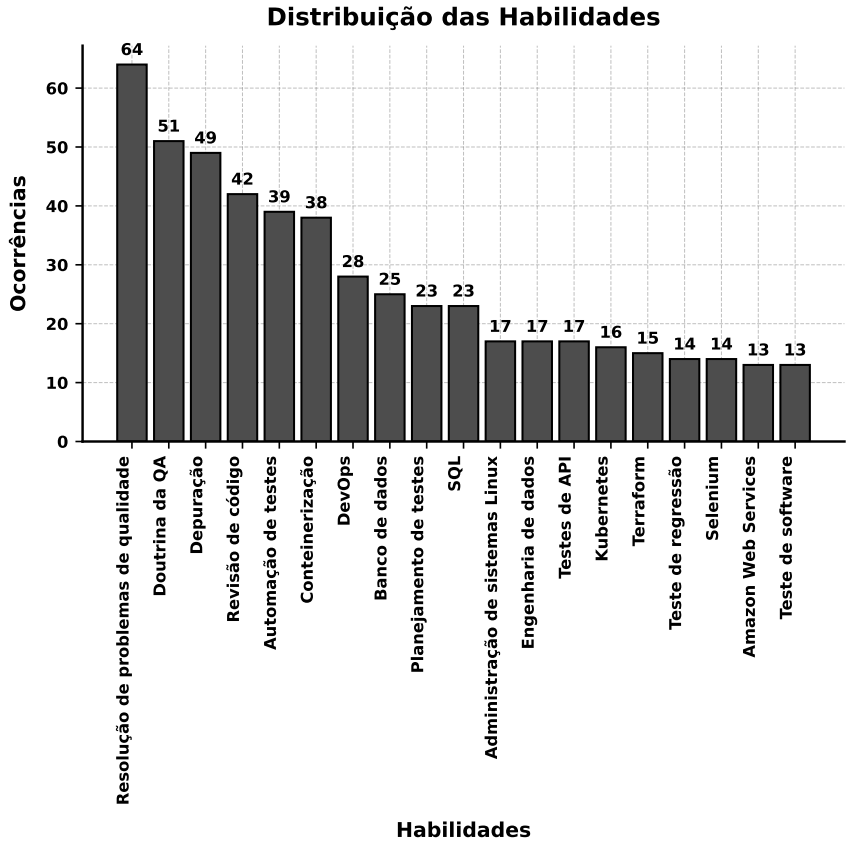


Figura 15.7: Gráfico de Garantia de Qualidade

- *Comunicação*: Habilidade de transmitir informações de forma clara, facilitando a interação com *stakeholders* e dentro da equipe.
- *Resolução de Problemas*: Habilidade para identificar e superar desafios técnicos.
- *Trabalho em Equipe*: Habilidade atuar como membro de equipe.
- *Aquisição de Conhecimento*: Capacidade de aprender e se adaptar a novas tecnologias e métodos.

Essas *soft skills* são destacadas não apenas por sua recorrência nas descrições de vagas, mas também pelo impacto significativo que exercem no desempenho profissional no desenvolvimento web. A Tabela 15.1 ilustra a relevância dessas habilidades, que foram frequentemente mencionadas nas oportunidades de emprego

Tabela 15.1: Soft Skills para Desenvolvimento Web

Habilidade	Descrição	Ocorrências Gerais
Inglês	Proficiência em comunicação técnica	214
Comunicação	Habilidade de comunicação eficaz	145
Resolução de Problemas	Capacidade de enfrentar desafios técnicos	81
Trabalho em Equipe	Capacidade de colaborar em projetos	30
Aquisição de Conhecimento	Aprendizado contínuo e adaptação	30

para os quatro cargos analisados neste estudo sobre desenvolvimento web, bem como o quantitativo de menções as essas *soft skill*.

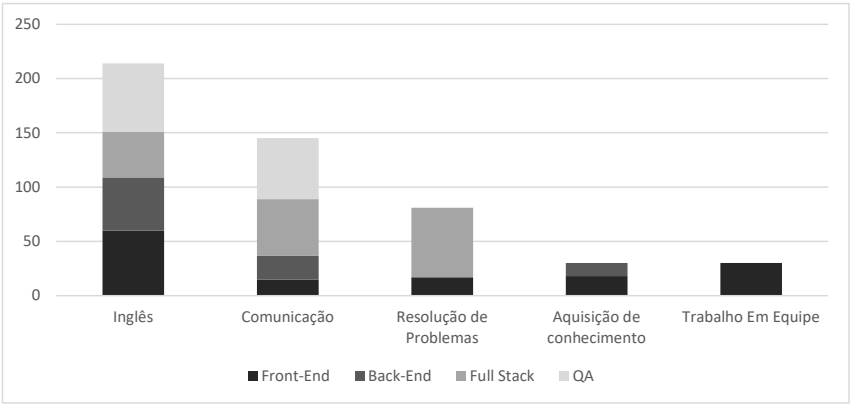


Figura 15.8: Gráfico de Soft Skills

15.7 Discussão dos Dados

Nesta seção serão abordadas as implicações dos dados analisados seguidas por recomendações de *hard skills* para desenvolvedores iniciantes e por fim uma reflexão sobre as lições aprendidas.

15.7.1 Implicações dos Dados

Os dados analisados revelam uma clara tendência nas demandas do mercado para desenvolvedores web, com ênfase significativa nas *hard skills* específicas para as áreas de *front-end*, *back-end*, *full stack* e QA. A presença recorrente de certas habilidades, como *Java inglês*, *HTML*, *JavaScript*, *CSS* e *Git*, indica a importância dessas competências no ambiente profissional atual. A habilidade em Inglês, por exemplo, é um pré-requisito indispensável, aparecendo consistentemente entre os

cargos analisados, o que reflete a globalização do mercado e a necessidade de comunicação em um idioma amplamente utilizado em documentação técnica e colaboração internacional.

Além disso, a alta demanda por conhecimentos em frameworks modernos e ferramentas de versionamento demonstra a rápida evolução tecnológica no desenvolvimento web, exigindo dos profissionais uma constante atualização para se manterem competitivos. As empresas buscam profissionais que não apenas possuam um conjunto sólido de habilidades técnicas, mas que também sejam capazes de se adaptar rapidamente às mudanças tecnológicas.

15.7.2 Recomendações de *Hard Skills* Essenciais para Iniciantes no Desenvolvimento Web

Para aqueles que estão iniciando na carreira de desenvolvimento web, é fundamental focar em habilidades que oferecem maior empregabilidade e que são comumente requisitadas. De acordo com os dados analisados, recomenda-se investir nas seguintes *hard skills*:

- *HTML* e *CSS*: Considerados os pilares do desenvolvimento web, são essenciais para qualquer desenvolvedor *front-end*. O *HTML* organiza o conteúdo, enquanto *CSS* define a sua aparência e estilo. Para um desenvolvedor front-end, dominar essas tecnologias é essencial para criar sites funcionalmente estruturados e visualmente atraentes.

- *Java*: Durante a análise, observou-se que *Java* foi a linguagem de programação com alta ocorrência em todos os quatro cargos analisados: *front End*, *back End*, *full Stack* e QA. Esta alta representatividade de *Java* destaca sua relevância e versatilidade no campo do desenvolvimento web. A importância de *Java* pode ser atribuída a várias razões:

  - **Versatilidade e Robustez**: *Java* é amplamente reconhecido por sua capacidade de criar aplicações robustas e escaláveis, o que é essencial para diversas áreas do desenvolvimento, incluindo *back End*, onde a construção de sistemas complexos e de alta performance é frequentemente necessária.
  - **Uso em Projetos Diversificados**: A presença de *Java* em *front End*, *full Stack* e QA indica que a linguagem não se limita a um único domínio, mas é empregada em múltiplos contextos, desde o desenvolvimento de interfaces de usuário até a automação de testes e o desenvolvimento completo de aplicações.
  - **Comunidade e Suporte**: A ampla adoção de *Java* é apoiada por uma comunidade ativa e um rico ecossistema de ferramentas e bibliotecas, o que facilita a resolução de problemas e a adaptação a novas necessidades tecnológicas.

- *JavaScript*: Com o crescimento de *frameworks* como *React.js* e *Node.js*, *JavaScript* se torna essencial tanto para *front-end* quanto para *back-end*. Por esse motivo, essa *hard skill* tem se destacado no mercado, sendo indispensável para quem deseja atuar como desenvolvedor *full stack* e dominar todas as etapas do desenvolvimento web.

- *Git*: A familiaridade com sistemas de versionamento, como o Git, mostrou-se importante para a colaboração em projetos e para o gerenciamento de código. O Git é amplamente utilizado devido à sua capacidade de gerenciar múltiplas versões de código simultaneamente, facilitando o rastreamento de mudanças e a coordenação entre equipes de desenvolvimento. Além disso, o uso do Git permite a implementação de práticas de integração contínua e o gerenciamento eficaz de ramificações, essenciais para manter a qualidade e a integridade do código em projetos complexos. Estudar e dominar o Git é, portanto, uma habilidade relevante para profissionais que desejam trabalhar em ambientes de desenvolvimento colaborativos.

- *Inglês*: Embora não seja uma habilidade técnica, o domínio do *inglês* é vital para acessar materiais educativos, documentação técnica e participar de comunidades globais de desenvolvedores.

Essas habilidades não só aumentarão as chances de emprego, mas também proporcionarão uma base sólida para o aprendizado de tecnologias mais avançadas no futuro.

15.7.3 Lições Aprendidas

Ao longo deste estudo, foram identificadas algumas lições importantes que podem ser aplicadas tanto por profissionais quanto por educadores na área de tecnologia:

- **A Importância da Atualização Contínua:** As tecnologias e ferramentas no desenvolvimento web estão em constante evolução. É fundamental que os profissionais continuem aprendendo e se adaptando às novas tendências para se manterem competitivos no mercado.

- **Valorização das Soft Skills:** Além das *hard skills*, as *soft skills* como comunicação e trabalho em equipe mostraram-se essenciais, especialmente em ambientes colaborativos onde a interação eficaz com colegas e clientes é crucial.

- **A Necessidade de uma Base Sólida:** Investir em fundamentos, como *HTML*, *CSS* e *JavaScript*, é essencial para a construção de uma carreira robusta em desenvolvimento web. Esses conhecimentos básicos são fundamentais para a compreensão e aplicação de tecnologias mais avançadas.

- **Importância da Experiência Prática:** A experiência prática em projetos reais é fundamental para a aplicação dos conhecimentos adquiridos. Participar de projetos, sejam pessoais ou colaborativos, ajuda a consolidar o aprendizado e a preparar melhor os profissionais para desafios reais do mercado.
- **Adaptabilidade e Flexibilidade:** O mercado de tecnologia está em constante mudança, e a capacidade de se adaptar rapidamente a novas ferramentas e metodologias é um diferencial importante. Profissionais que demonstram flexibilidade e disposição para aprender novas habilidades tendem a se destacar.
- **Valor da Especialização:** No desenvolvimento web, a especialização em áreas específicas, como *front-end*, *back-end* ou *full-stack*, pode proporcionar uma vantagem significativa. Profissionais especializados têm a oportunidade de se destacar em nichos específicos do mercado, tornando-se experts em suas áreas e atendendo a demandas mais complexas e desafiadoras. A especialização também pode abrir portas para oportunidades de liderança e projetos mais avançados, além de possibilitar um desenvolvimento mais profundo das habilidades técnicas e conceituais relacionadas à sua área de foco.
- **Valor da Graduação em Tecnologia:** A análise revelou que muitas vagas exigem uma graduação em *tecnologia* principalmente Ciência da Computação. Isso sublinha a importância de uma formação acadêmica sólida, que oferece uma base essencial para entender os principais conceitos e ferramentas da área. Além disso, a graduação proporciona um *networking* valioso com profissionais e colegas do setor, o que pode abrir portas para oportunidades de emprego e colaboração.

Lista de Siglas

SUT *System Under Test*. 550

AOT *Ahead-Of-Time*. 320

API *Application Programming Interface*. 240–242, 290, 291, 294, 295, 298–301, 305–308, 311–313, 315, 317–319, 323, 328–330, 332, 333, 336, 398, 412

API REST *Representational State Transfer Application Programming Interface*. 411

ARPANET *Advanced Research Projects Agency Network*. 240

BLANK *Blank Lines of Code*. 555

BSD *Berkeley Software Distribution*. 308

BSON *Binary JSON*. 246

CLI *Common Language Infrastructure*. 319, 321, 324, 390, 409

CLOC *Comment Lines of Code*. 555

CLR *Common Language Runtime*. 319

CPU Unidade Central de Processamento. 296, 297

CRUD *Create, Read, Update e Delete*. 288, 301–303, 311, 313, 315, 322, 323, 327, 332

CSS *Cascading Style Sheets*. 240, 309, 387–389, 406

CSV *Comma-separated values*. 270

DBA *Database Administrator*. 253

DCL *Data Control Language*. 267

DDL *Data Definition Language*. 266

DML *Transaction Manipulation Language*. 267

DOM *Document Object Mode*l. 299, 399, 400

DQL *Data Query Language*. 262

DRF *Django REST Framework*. 293, 308, 310–312, 315, 334–336, 736

DRY *Don't Repeat Yourself*. 309, 310

EF *Entity Framework*. 321, 324, 327

EFC *Entity Framework Core*. 321, 324

ER Entidade Relacionamento. 253

EUA Estados Unidos da América. 240

HTML *HyperText Markup Language*. 240, 290, 291, 309, 387–389, 391, 392, 394, 399, 406

HTTP *Hypertext Transfer Protocol*. 241, 290, 291, 293, 295, 300, 301, 303–307, 311, 315, 316, 323, 411, 412

HTTPS *Hypertext Transfer Protocol Secure*. 290

IDE *Integrated Development Environment*. 319, 321

IDL *Interface Definition Language*. 364

IL linguagem intermediária. 320

IOT *Internet of Things*. 246

IP *Internet Protocol*. 240

JIT *Just in Time*. 296, 320

JS *JavaScript*. 240, 309

JSON *JavaScript Object Notation*. 247, 269, 270, 284, 290, 291, 301, 304–306, 310, 311, 314, 316, 340, 346, 411, 412

JSX Extensão de sintaxe para JavaScript. 392

LINQ *Language Integrated Query*. 288, 321

MER *Modelo Entidade Relacionamento*. 243, 244

MVC *Model View Controller*. 323

MVT *Model View Template*. 308, 309

NoSQL *Not Only Structured Query Language*. 246

NPM *Node Package Manager*. 300

ORM *Object Relational Mapping*. 287, 288, 308, 321, 335

PLOC *Physical Lines of Code*. 555

POO *Object-oriented Programming*. 310, 320

REST *Representational State Transfer*. 290, 291, 300, 308, 310–312, 317, 318, 323, 324, 327, 332

RPC *Remote Procedure Call*. 363

SDK *Software Development Kit*. 321, 322

SDL *Schema Definition Language*. 361

SGBD Sistema de Gerenciamento de Banco de Dados. 246, 247, 249–252, 254–256, 262, 269, 280

SLOC *Source Lines of Code*. 555

SO Sistema Operacional. 255, 256, 270, 271, 281, 282, 320, 334, 546

SQL *Structured Query Language*. 246, 253, 262, 266–268, 270, 276, 291, 292, 301, 312, 321, 335

SSE *Server-sent events*. 351

TCL *Transaction Control Language*. 268

TCP *Transmission Control Protocol*. 240, 350

URL *Uniform Resource Locator*. 293, 295, 306–308, 313, 318, 342

VirtualDOM *Virtual Document Object Model*. 398–400

VM *Virtual Machine*. 250

WSL *Windows Subsystem for Linux*. 281, 282

XML *Extensible Markup Language*. 290, 291, 310

XSS *Cross Site Scripting*. 389

YAML *YAML Ain't Markup Language*. 310

APÊNDICE A

Tabela de comparação entre o *Selenium* e *Cypress*

A Tabela consiste em um resumo das características do *Selenium* e *Cypress*.

Tabela A.1: Comparação entre as ferramentas Selenium e Cypress.

CARACTERÍSTICA		**SELENIUM**	**CYPRESS**
Open source		SIM	SIM
Sistemas Operacionais Suportados	**Windows**	SIM	SIM
	macOS	SIM	SIM
	Linux	SIM	SIM
	Android	SIM	NÃO
	iOS	SIM	NÃO
	Outros	SIM	SIM
Linguagens de Programação Suportadas	**JavaScript**	SIM	SIM
	Python	SIM	NÃO
	Java	SIM	NÃO
	C#	SIM	NÃO
	PHP	SIM	NÃO
	Ruby	SIM	NÃO
	Outras	SIM *(Kotlin e Perl)*	NÃO
Navegadores Suportados por padrão	**Google Chrome**	SIM	SIM
	Mozila Firefox	SIM	SIM
	Microsoft Edge	SIM	SIM
	Opera	SIM	N/C
	Safari	SIM	N/C
	Brave	NÃO *(Mas configurável)*	SIM
	Outros	SIM *(Internet Explorer)*	SIM *(Electron e WebKit (Experimental))*
Dependências de Drivers		SIM	NÃO
Multi-abas		SIM	NÃO
Múltiplas instâncias do navegador		SIM	NÃO
Viagem no tempo		NÃO	SIM
Capturas de tela e vídeo disponíveis por padrão		NÃO *(Mas configurável)*	SIM
Documentação	**Completa**	NÃO	SIM
	Internacionalizada	SIM	SIM
Execução Remota		SIM	NÃO
Teste Paralelo		SIM	SIM
Uso em CI/CD		SIM	SIM
Relatório de Execução		SIM	SIM

APÊNDICE B

Especificação dos testes do *SUT*

A Tabela consiste na documentação para especificar todos os casos de testes que foram implementados e executados neste estudo. Nas colunas **TC** (do inglês, *Test Case*) e **Título**, são exibidos o identificador e o título dos casos de testes. Em seguida são detalhados os passos para a execução de cada caso de teste e os resultados esperados de cada passo, respectivamente, em **Passos** e **Resultado Esperado**. A coluna identificada como **UC** (do inglês *User Case*) exibe o caso de uso correspondente ao teste.

UC	TC	Título	Passos	Resultado Esperado
UC01 - Cadastro de Usuários	TC01	Cadastro de usuário com sucesso	1. Abrir a aplicação ParaBank. 2. Verificar a exibição do menu Register. 3. Clicar em Register. 4. Verificar a exibição da tela de cadastro com os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 5. Inserir dados válidos nos campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 6. Clicar no botão Register.	1. Exibir página inicial do ParaBank. 2. O menu Register deve ser exibido. 3. A aplicação exibe a tela de cadastro de clientes. 4. Os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm devem ser exibidos. 5. Os dados devem ser inseridos nos campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 6. O login deve ser efetuado e a aplicação exibe na tela a mensagem: “Welcome <nome-do-usuário> Your account was created successfully. You are now logged in.”
	TC02	Cadastro de usuário com todos os campos vazios	1. Abrir a aplicação ParaBank. 2. Verificar a exibição do menu Register. 3. Clicar em Register. 4. Verificar a exibição da tela de cadastro com os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 5. Deixar vazios os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 6. Clicar no botão Register.	1. Exibir página inicial do ParaBank. 2. O menu Register deve ser exibido. 3. A aplicação exibe a tela de cadastro de clientes. 4. Os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm devem ser exibidos. 5. Os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm devem permanecer vazios. 6. A aplicação exibe uma mensagem de “<nome-campo> is required” para os campos: First name, Last name, Address, City, State, Zip Code, Social Security Number, Username, Password e Password confirmation.
	TC03	Cadastro de usuário com um username já cadastrado	1. Abrir a aplicação ParaBank. 2. Verificar a exibição do menu Register. 3. Clicar em Register. 4. Verificar a exibição da tela de cadastro com os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 5. Inserir dados válidos nos campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 6. Clicar no botão Register. 7. Localizar o menu de Log Out. 8. Clicar no menu de Log Out. 9. Verificar a exibição do menu Register. 10. Clicar em Register. 11. Verificar a exibição da tela de cadastro com os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 12. Inserir os mesmos dados válidos do usuário cadastrado anteriormente nos campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 13. Clicar no botão Register.	1. Exibir página inicial do ParaBank. 2. O menu Register deve ser exibido. 3. A aplicação exibe a tela de cadastro de clientes. 4. Os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm devem ser exibidos. 5. Os dados devem ser inseridos nos campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 6. O login deve ser efetuado e a aplicação exibe na tela a mensagem: “Welcome <nome-do-usuário> Your account was created successfully. You are now logged in.” 7. O menu de Log Out deve ser exibido. 8. A aplicação encerra a sessão e retorna para a tela inicial de login do cliente. 9. O menu Register deve ser exibido. 10. A aplicação exibe a tela de cadastro de clientes. 11. Os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm devem ser exibidos. 12. Os dados devem ser inseridos nos campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 13. A aplicação exibe uma mensagem de alerta: “This username already exists.”
	TC04	Cadastro de usuário com campos não obrigatórios vazios	1. Abrir a aplicação ParaBank. 2. Verificar a exibição do menu Register. 3. Clicar em Register. 4. Verificar a exibição da tela de cadastro com os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 5. Inserir dados válidos nos campos: First Name, Last Name, Address, City, State, Zip Code, SSN, Username, Password e Confirm. 6. Deixar o campo Phone não obrigatório vazio. 7. Clicar no botão Register.	1. Exibir página inicial do ParaBank. 2. O menu Register deve ser exibido. 3. A aplicação exibe a tela de cadastro de clientes. 4. Os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm devem ser exibidos. 5. Os dados devem ser inseridos nos campos: First Name, Last Name, Address, City, State, Zip Code, SSN, Username, Password e Confirm. 6. O campo Phone deve permanecer vazio. 7. O login deve ser efetuado e a aplicação exibe na tela a mensagem: “Welcome <nome-do-usuário> Your account was created successfully. You are now logged in.”
	TC05	Cadastro de usuário com senhas diferentes	1. Abrir a aplicação ParaBank. 2. Verificar a exibição do menu Register. 3. Clicar em Register. 4. Verificar a exibição da tela de cadastro com os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 5. Inserir dados válidos nos campos: First Name, Last Name, Address, City, State, Zip Code, Phone e Username. 6. Inserir senhas diferentes nos campos Password e Confirm. 7. Clicar no botão Register.	1. Exibir página inicial do ParaBank. 2. O menu Register deve ser exibido. 3. A aplicação exibe a tela de cadastro de clientes. 4. Os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm devem ser exibidos. 5. Os dados devem ser inseridos nos campos: First Name, Last Name, Address, City, State, Zip Code, Phone e Username. 6. Os dados (diferentes) devem ser inseridos nos campos Password e Confirm. 7. A aplicação exibe uma mensagem de alerta “Passwords did not match.” ao lado do campo Confirm.
	TC06	Cadastro de usuário com campos obrigatórios vazios	1. Abrir a aplicação ParaBank. 2. Verificar a exibição do menu Register. 3. Clicar em Register. 4. Verificar a exibição da tela de cadastro com os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm. 5. Inserir dados válidos nos campos: First Name, Last Name, Address, City, State, Zip Code, Phone, Username, Password e Confirm. 6. Deixar o campo obrigatório vazio (SSN). 7. Clicar no botão Register.	1. Exibir página inicial do ParaBank. 2. O menu Register deve ser exibido. 3. A aplicação exibe a tela de cadastro de clientes. 4. Os campos: First Name, Last Name, Address, City, State, Zip Code, Phone, SSN, Username, Password e Confirm devem ser exibidos. 5. Os dados devem ser inseridos nos campos: First Name, Last Name, Address, City, State, Zip Code, Phone, Username, Password e Confirm. 6. O campo obrigatório SSN deve permanecer vazio. 7. A aplicação exibe uma mensagem de alerta “Social Security Number is required.” ao lado do campo SSN.

UC02 - Login de Usuários	TC07	Login com sucesso	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado e a página de Visão geral das contas deve ser exibida.
	TC08	Login com todos os campos vazios	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Não inserir nenhum dado nos campos UserName e Password. 4. Clicar no botão Login.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os campos UserName e Password permanecem vazios. 4. Na tela da aplicação deve ser exibida a mensagem: "Error! Please enter a username and password."
	TC09	Login com usuário não cadastrado	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário não cadastrado. 4. Inserir uma senha qualquer. 5. Clicar no botão Login.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. Na tela da aplicação deve ser exibida a mensagem: "Error! An internal error has occurred and has been logged."
	TC10	Login com senha errada	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha errada. 5. Clicar no botão Login.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. Na tela da aplicação deve ser exibida a mensagem: "Error! An internal error has occurred and has been logged."
	TC11	Login com campo de usuário vazio	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Deixar o campo UserName vazio. 4. Inserir a senha válida. 5. Clicar no botão Login.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. O campo UserName deve permanecer vazio. 4. Os dados de Password devem ser inseridos no campo. 5. Na tela da aplicação deve ser exibida a mensagem: "Error! Please enter a username and password."
	TC12	Login com campo de senha vazio	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Deixar o campo Password vazio. 5. Clicar no botão Login.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. O campo UserName deve permanecer vazio. 4. Os dados de Password devem ser inseridos no campo. 5. Na tela da aplicação deve ser exibida a mensagem: "Error! Please enter a username and password."
UC03 - Encontrar informações de login	TC13	Encontrar informações de login com sucesso	1. Abrir a aplicação ParaBank. 2. Verificar a exibição do menu Forgot login info. 3. Clicar em Forgot login info. 4. Verificar a exibição da tela de pesquisa de cliente com os campos: First Name, Last Name, Address, City, State, Zip Code e SSN. 5. Inserir dados válidos de um usuário já cadastrado nos campos: First Name, Last Name, Address, City, State, Zip Code e SSN. 6. Clicar no botão "Find My Login Info".	1. Exibir página inicial do ParaBank. 2. O menu Forgot login info deve ser exibido. 3. A aplicação exibe a tela de pesquisa de clientes. 4. Os campos: First Name, Last Name, Address, City, State, Zip Code e SSN devem ser exibidos. 5. Os dados devem ser inseridos nos campos: First Name, Last Name, Address, City, State, Zip Code e SSN. 6. A pesquisa deve ser efetuada e a aplicação exibe na tela a mensagem de “Your login information was located successfully. You are now logged in.i” e os dados de Username e Password.
	TC14	Encontrar informações de login com todos os campos vazios	1. Abrir a aplicação ParaBank. 2. Verificar a exibição do menu Forgot login info. 3. Clicar em Forgot login info. 4. Verificar a exibição da tela de pesquisa de cliente com os campos: First Name, Last Name, Address, City, State, Zip Code e SSN. 5. Deixar vazios os campos: First Name, Last Name, Address, City, State, Zip Code e SSN. 6. Clicar no botão "Find My Login Info".	1. Exibir página inicial do ParaBank. 2. O menu Forgot login info deve ser exibido. 3. A aplicação exibe a tela de pesquisa de clientes. 4. Os campos: First Name, Last Name, Address, City, State, Zip Code e SSN devem ser exibidos. 5. Os campos: First Name, Last Name, Address, City, State, Zip Code e SSN devem permanecer vazios. 6. A aplicação exibe uma mensagem de “<nome-campo> is required” para todos os campos: First Name, Last Name, Address, City, State, Zip Code e SSN.
	TC15	Encontrar informações de login com dados inconsistentes	1. Abrir a aplicação ParaBank. 2. Verificar a exibição do menu Forgot login info. 3. Clicar em Forgot login info. 4. Verificar a exibição da tela de pesquisa de cliente com os campos: First Name, Last Name, Address, City, State, Zip Code e SSN. 5. Inserir dados inválidos (dados errados de um cliente cadastrado) nos campos: First Name, Last Name, Address, City, State, Zip Code e SSN. 6. Clicar no botão "Find My Login Info".	1. Exibir página inicial do ParaBank. 2. O menu Forgot login info deve ser exibido. 3. A aplicação exibe a tela de pesquisa de clientes. 4. Os campos: First Name, Last Name, Address, City, State, Zip Code e SSN devem ser exibidos. 5. Os dados devem ser inseridos nos campos: First Name, Last Name, Address, City, State, Zip Code e SSN. 6. A aplicação exibe uma mensagem de alerta na tela: “Error! The customer information provided could not be found.”
	TC16	Encontrar informações de login com usuário não cadastrado	1. Abrir a aplicação ParaBank. 2. Verificar a exibição do menu Forgot login info. 3. Clicar em Forgot login info. 4. Verificar a exibição da tela de pesquisa de cliente com os campos: First Name, Last Name, Address, City, State, Zip Code e SSN. 5. Inserir dados inválidos (de um cliente não cadastrado) nos campos: First Name, Last Name, Address, City, State, Zip Code e SSN. 6. Clicar no botão "Find My Login Info".	1. Exibir página inicial do ParaBank. 2. O menu Forgot login info deve ser exibido. 3. A aplicação exibe a tela de pesquisa de clientes. 4. Os campos: First Name, Last Name, Address, City, State, Zip Code e SSN devem ser exibidos. 5. Os dados devem ser inseridos nos campos: First Name, Last Name, Address, City, State, Zip Code e SSN. 6. A aplicação exibe uma mensagem de alerta na tela: “Error! The customer information provided could not be found.
	TC17	Encontrar informações de login com campos obrigatórios vazios	1. Abrir a aplicação ParaBank. 2. Verificar a exibição do menu Forgot login info. 3. Clicar em Forgot login info. 4. Verificar a exibição da tela de pesquisa de cliente com os campos: First Name, Last Name, Address, City, State, Zip Code e SSN. 5. Inserir dados válidos nos campos: First Name, Last Name, Address, City, State, Zip Code. 6. Deixar o campo obrigatório vazio (SSN). 7. Clicar no botão "Find My Login Info".	1. Exibir página inicial do ParaBank. 2. O menu Forgot login info deve ser exibido. 3. A aplicação exibe a tela de pesquisa de clientes. 4. Os campos: First Name, Last Name, Address, City, State, Zip Code e SSN devem ser exibidos. 5. Os dados devem ser inseridos nos campos: First Name, Last Name, Address, City, State, Zip Code. 6. O campo obrigatório SSN deve permanecer vazio. 7. A aplicação exibe uma mensagem de alerta “Social Security Number is required.” ao lado do campo SSN.

UC04 - Atualizar dados	TC18	Atualizar dados com sucesso	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login. 6. Verificar a exibição do menu “Update Contact Info”. 7. Clicar no menu “Update Contact Info”. 8. Verificar a exibição dos campos: First Name, Last Name, Address, City, State, Zip Code e Phone. 9. Atualizar os dados de endereço e telefone do usuário nos campos correspondentes (Address, City, State, Zip Code e Phone). 10. Clicar no botão de “Update Profile”.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado e a página de Visão geral das contas deve ser exibida. 6. O menu "Update Contact Info" deve ser exibido. 7. A aplicação exibe a tela de atualizar perfil. 8. Os campos: First Name, Last Name, Address, City, State, Zip Code e Phone devem ser exibidos. 9. Os novos dados de endereço e telefone devem ser exibidos nos campos correspondentes. 10. A aplicação atualiza os dados e exibe na tela “Profile Updated”, como também uma mensagem: “Your updated address and phone number have been added to the system.”
	TC19	Atualizar dados com todos os campos vazios	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login. 6. Verificar a exibição do menu “Update Contact Info”. 7. Clicar no menu “Update Contact Info”. 8. Verificar a exibição dos campos: First Name, Last Name, Address, City, State, Zip Code e Phone. 9. Apagar todos os dados de todos os campos (First Name, Last Name, Address, City, State, Zip Code e Phone). 10. Clicar no botão de “Update Profile”.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado e a página de Visão geral das contas deve ser exibida. 6. O menu "Update Contact Info" deve ser exibido. 7. A aplicação exibe a tela de atualizar perfil. 8. Os campos: First Name, Last Name, Address, City, State, Zip Code e Phone devem ser exibidos. 9. Os campos ficam vazios (First Name, Last Name, Address, City, State, Zip Code e Phone). 10. A aplicação exibe uma mensagem de “<nome-campo> is required” para todos os campos: First Name, Last Name, Address, City, State, Zip Code e Phone.
	TC20	Atualizar dados com campos obrigatórios vazios	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login. 6. Verificar a exibição do menu “Update Contact Info”. 7. Clicar no menu “Update Contact Info”. 8. Verificar a exibição dos campos: First Name, Last Name, Address, City, State, Zip Code e Phone. 9. Atualizar os dados de endereço nos campos: Address, City, State e deixar o campo Zip Code (obrigatório) vazio. 10. Clicar no botão de “Update Profile”.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado e a página de Visão geral das contas deve ser exibida. 6. O menu "Update Contact Info" deve ser exibido. 7. A aplicação exibe a tela de atualizar perfil. 8. Os campos: First Name, Last Name, Address, City, State, Zip Code e Phone devem ser exibidos. 9. Os novos dados de endereço devem ser exibidos nos campos correspondentes, e o campo Zip Code deve estar vazio. 10. A aplicação exibe uma mensagem de “Zip Code is required.” ao lado do campo obrigatório Zip Code.
	TC21	Atualizar dados com o campo telefone (não obrigatório) vazio	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login. 6. Verificar a exibição do menu “Update Contact Info”. 7. Clicar no menu “Update Contact Info”. 8. Verificar a exibição dos campos: First Name, Last Name, Address, City, State, Zip Code e Phone. 9. Atualizar os dados de endereço nos campos: Address, City, State, Zip Code e deixar o campo Phone (não obrigatório) vazio. 10. Clicar no botão de “Update Profile”.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado e a página de Visão geral das contas deve ser exibida. 6. O menu "Update Contact Info" deve ser exibido. 7. A aplicação exibe a tela de atualizar perfil. 8. Os campos: First Name, Last Name, Address, City, State, Zip Code e Phone devem ser exibidos. 9. Os novos dados devem ser exibidos nos campos correspondentes. 10. A aplicação atualiza os dados e exibe na tela “Profile Updated”, como também uma mensagem: “Your updated address and phone number have been added to the system.”
UC05 - Criar uma conta	TC22	Criar uma conta com sucesso	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login. 6. Verificar a exibição do menu “Open New Account”. 7. Clicar no menu “Open New Account”. 8. Verificar a exibição dos campos para selecionar o tipo de conta e uma conta existente para transferir fundos para a nova conta. 9. Selecionar o tipo de conta. 10. Selecionar uma conta existente para transferir fundos para a nova conta. 11. Clicar no botão de “Open New Account”.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado e a página de Visão geral das contas deve ser exibida. 6. O menu “Open New Account” deve ser exibido. 7. A aplicação exibe a tela de abrir uma nova conta. 8. Os campos para selecionar o tipo de conta e uma conta existente para transferir fundos para a nova conta devem ser exibidos. 9. O tipo de conta selecionado é exibido no menu de seleção. 10. A conta existente selecionada para transferir fundos para a nova conta é exibida no menu. 11. A aplicação abre a conta e exibe na tela “Account Opened!”, assim como uma mensagem: “Opened! Congratulations, your account is now open ” seguida dos dados da nova conta (número da conta).

UC05 - Criar uma conta	TC23	Visualizar detalhes de uma nova conta	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login. 6. Verificar a exibição do menu “Open New Account”. 7. Clicar no menu “Open New Account”. 8. Verificar a exibição dos campos para selecionar o tipo de conta e uma conta existente para transferir fundos para a nova conta. 9. Selecionar o tipo de conta. 10. Selecionar uma conta existente para transferir fundos para a nova conta. 11. Clicar no botão de “Open New Account”. 12. Clicar no número da conta criada. 13. Verificar os detalhes da conta criada.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado e a página de Visão geral das contas deve ser exibida. 6. O menu “Open New Account” deve ser exibido. 7. A aplicação exibe a tela de abrir uma nova conta. 8. Os campos para selecionar o tipo de conta e uma conta existente para transferir fundos para a nova conta devem ser exibidos. 9. O tipo de conta selecionado é exibido no menu de seleção. 10. A conta existente selecionada para transferir fundos para a nova conta é exibida no menu. 11. A aplicação abre a conta e exibe na tela “Account Opened!”, assim como uma mensagem: “Opened! Congratulations, your account is now open ” seguida dos dados da nova conta (número da conta). 12. A aplicação redireciona para a tela de detalhes da conta (Account Details). 13. A aplicação deve exibir os dados da conta nos campos: Account Number, Account Type, Balance e Available.
UC06 - Visualizar painel geral das contas	TC24	Visualizar painel geral das contas com sucesso	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login. 6. Verificar a exibição do menu “Accounts Overview”. 7. Clicar no menu “Accounts Overview”.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado. 6. O menu “Accounts Overview” deve ser exibido. 7. A aplicação exibe a tela de visão geral das contas.
	TC25	Exibir lista de contas com sucesso	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login. 6. Verificar a exibição da tela geral das contas (Accounts Overview). 7. Verificar a exibição da tabela com os dados das contas do usuário (Account, Balance e Available Amount).	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado e a página de Visão geral das contas deve ser exibida. 6. A página de Visão geral das contas deve ser exibida. 7. A aplicação exibe uma tabela com os dados das contas do usuário (Account, Balance e Available Amount).
	TC26	Visualizar detalhes de uma conta com sucesso	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login. 6. Verificar a exibição da tela geral das contas (Accounts Overview). 7. Verificar a exibição da tabela com os dados das contas do usuário (Account, Balance e Available Amount). 8. Clicar na linha correspondente à uma conta.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado e a página de Visão geral das contas deve ser exibida. 6. A página de Visão geral das contas deve ser exibida. 7. A aplicação exibe uma tabela com os dados das contas do usuário (Account, Balance e Available Amount). 8. A aplicação redireciona para a tela de detalhes da conta (Account Details).
	TC27	Verificar dados da conta	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login. 6. Verificar a exibição da tela geral das contas (Accounts Overview). 7. Verificar a exibição da tabela com os dados das contas do usuário (Account, Balance e Available Amount). 8. Clicar na linha correspondente à uma conta. 9. Verificar os detalhes da conta exibidos.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado e a página de Visão geral das contas deve ser exibida. 6. A página de Visão geral das contas deve ser exibida. 7. A aplicação exibe uma tabela com os dados das contas do usuário (Account, Balance e Available Amount). 8. A aplicação redireciona para a tela de detalhes da conta (Account Details). 9. A aplicação deve exibir os dados da conta nos campos: Account Number, Account Type, Balance e Available.
	TC28	Visualizar atividades da conta	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login. 6. Verificar a exibição da tela geral das contas (Accounts Overview). 7. Verificar a exibição da tabela com os dados das contas do usuário (Account, Balance e Available Amount). 8. Clicar na linha correspondente à uma conta. 9. Selecionar o período da atividade da conta no menu dropdown. 10. Selecionar o tipo de atividade (Credit ou Debit). 11. Clicar no botão “Go”.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado e a página de Visão geral das contas deve ser exibida. 6. A página de Visão geral das contas deve ser exibida. 7. A aplicação exibe uma tabela com os dados das contas do usuário (Account, Balance e Available Amount). 8. A aplicação redireciona para a tela de detalhes da conta (Account Details). 9. A aplicação deve exibir no menu dropdown a opção selecionada para o período da atividade. 10. A aplicação deve exibir no menu dropdown a opção selecionada para o tipo de atividade (Credit ou Debit). 11. A aplicação exibe uma tabela com as informações de atividade da conta (com os campos: Date, Transaction, Debit e Credit).

UC06 - Visualizar painel geral das contas	TC29	Visualizar lista de transações da conta	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login. 6. Verificar a exibição da tela geral das contas (Accounts Overview). 7. Verificar a exibição da tabela com os dados das contas do usuário (Account, Balance e Available Amount). 8. Clicar na linha correspondente à uma conta. 9. Selecionar o período da atividade da conta no menu dropdown. 10. Selecionar o tipo de atividade (Credit ou Debit). 11. Clicar no botão “Go”. 12. Clicar na linha clicável “Funds Transfer Sent” na atividade da conta.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado e a página de Visão geral das contas deve ser exibida. 6. A página de Visão geral das contas deve ser exibida. 7. A aplicação exibe uma tabela com os dados das contas do usuário (Account, Balance e Available Amount). 8. A aplicação redireciona para a tela de detalhes da conta (Account Details). 9. A aplicação deve exibir no menu dropdown a opção selecionada para o período da atividade. 10. A aplicação deve exibir no menu dropdown a opção selecionada para o tipo de atividade (Credit ou Debit). 11. A aplicação exibe uma tabela com as informações de atividade da conta (com os campos: Date, Transaction, Debit e Credit). 12. A aplicação redireciona para a tela de detalhes da transação (Transaction Details) como os campos: Transaction ID, Date, Description, Type e Amount.
UC07 - Sair do sistema	TC30	Realizar logout	1. Abrir a aplicação ParaBank. 2. Verificar a exibição dos campos UserName e Password. 3. Inserir um nome de usuário válido. 4. Inserir a senha válida. 5. Clicar no botão Login. 6. Localizar o menu de Log Out. 7. Clicar no menu de Log Out.	1. Exibir página inicial do ParaBank. 2. Os campos UserName e Password devem ser exibidos. 3. Os dados de UserName devem ser inseridos no campo. 4. Os dados de Password devem ser inseridos no campo. 5. O login deve ser efetuado e a página de Visão geral das contas deve ser exibida. 6. O menu de Log Out deve ser exibido. 7. A aplicação encerra a sessão e retorna para a tela inicial de login do cliente.

APÊNDICE C

Repositórios remotos dos testes

Os *scripts* de testes implementados com as ferramentas em estudo, `Selenium` e `Cypress`, foram devidamente armazenados nos repositórios remotos do GitHub.

- ***Scripts* no Selenium:**⟨https://github.com/sousaellen/Selenium-Test-Automation⟩
- ***Scripts* no Cypress:** ⟨https://github.com/sousaellen/Cypress-Test-Automation⟩

Referências Bibliográficas

ACATE. *Santa Catarina avança na formação de profissionais na área da tecnologia*. 2022. Acesso em: 28 nov. 2023. Disponível em: ⟨https://www.acate.com.br/blog-da-acate/santa-catarina-avanca-na-formacao-de-profissionais-na-area-da-tecnologia/⟩.

AGARWAL, V. et al. "way back then": A data-driven view of 25+ years of web evolution. In: *Proceedings of the ACM Web Conference 2022*. [S.l.: s.n.], 2022. p. 3471–3479.

AGUILAR, J. H. S. et al. Publication of linked open data–a systematic literature review for identifying problems and technical tools supporting the process. *Journal of Computer Science and Technology*, v. 23, n. 2, p. e16–e16, 2023.

AGULAR. *Angular Material*. 2025. Acessado em 22 de abril de 2025: ⟨https://material.angular.io/⟩.

AGULAR. *Angular Universal*. 2025. Acessado em 22 de abril de 2025: ⟨https://v17.angular.io/guide/ssr⟩.

AGULAR. *Angular's Github Changelog Archive*. 2025. Acessado em 22 de abril de 2025: ⟨https://raw.githubusercontent.com/angular/angular/main/CHANGELOG_ARCHIVE.md⟩.

AHMED, F. et al. Evaluating the demand for soft skills in software development. *IT Professiona*, v. 14, n. 12487439, p. 44 – 49, janeiro 2012. ISSN 1520-9202.

ALMEIDA, M. B. Uma introdução ao xml, sua utilização na internet e alguns conceitos complementares. *Ciência da informação*, SciELO Brasil, v. 31, p. 5–13, 2002.

ALRAZAK. *GRPC stubs and Skeletons*. Medium, 2023. Acesso em: 06 junho 2025. Disponível em: ⟨https://medium.com/%40alrazak/grpc-stubs-and-skeletons-f0203bfedf11⟩.

ALURA. *Django e Django Rest: Diferenças e semelhanças*. 2023. Acessado em 09-junho-2023, disponível em ⟨https://www.alura.com.br/artigos/django-django-rest-diferencas⟩.

ALURA. *Apostila HTML, CSS e JavaScript - Seção 03: A Especificação HTML*. 2024. Disponível em: ⟨https://www.alura.com.br/apostila-html-css-javascript/03CA-a-spec-html⟩.

ALURA. O que é html: suas tags e atributos - parte 5. 2024. Disponível em: ⟨https://www.alura.com.br/artigos/o-que-e-html-suas-tags-parte-5-atributos-elementos⟩.

AMAZON. *O que é API RESTful?* 2023. Acessado em 09-junho-2023, disponível em ⟨https://aws.amazon.com/pt/what-is/restful-api/⟩.

AMAZON. *O que é o Django?* 2023. Acessado em 09-junho-2023, disponível em ⟨https://aws.amazon.com/pt/what-is/django/⟩.

AMAZON. *O que é uma CLI (Interface de linha de comando)?* 2025. Acessado em 25 de fevereiro de 2025: ⟨https://aws.amazon.com/pt/what-is/cli/⟩.

AMAZON, A. *Qual é a diferença entre RPC e REST?* 2024. Acesso em: 27 ago. 2024. Disponível em: ⟨https://aws.amazon.com/compare/the-difference-between-rpc-and-rest/⟩.

AMUNDSEN, M. *RESTful Web API Patterns and Practices Cookbook: Connecting and Orchestrating Microservices and Distributed Data*. 1. ed. [S.l.]: O'Reilly Media, 2023.

ANALYTICS, M. *What is an API Request Body?* 2024. Acesso em: 13 maio 2025. Disponível em: ⟨https://mixedanalytics.com/knowledge-base/request-bodies-explained/⟩.

ANGMO, R. et al. Selenium tool: A web based automation testing framework. *International Journal of Emerging Technologies in Computational and Applied Science, Chandigarh*, 2014.

Angular Team. *Angular CLI Overview*. [S.l.], 2025. Acessado em 22 de abril de 2025. Disponível em: ⟨https://angular.io/cli⟩.

Angular Team. *Angular Components Overview*. [S.l.], 2025. Disponível em: ⟨https://angular.io/guide/architecture-components⟩.

Angular Team. *Angular Routing*. 2025. Acesso em: 26 Junho 2025. Disponível em: ⟨https://angular.dev/guide/routing⟩.

Angular Team. *Angular: The modern web developer's platform*. 2025. ⟨https://angular.io⟩. Accessed: 2025-04-22.

Angular Team. *Application Architecture*. 2025. Acesso em 04 de maio de 2025. Disponível em: ⟨https://angular.io/guide/architecture⟩.

Angular Team. *Composing Components*. 2025. Acesso em: 04 maio 2025. Disponível em: ⟨https://angular.dev/essentials/components⟩.

Angular Team. *Conditional Rendering*. 2025. Acesso em 14 de maio de 2025. Disponível em: ⟨https://angular.io/guide/structural-directives#ngif⟩.

Angular Team. *Conditionals and Loops*. 2025. Acesso em 14 de maio de 2025. Disponível em: ⟨https://angular.dev/api/common/NgIf⟩.

Angular Team. *Conditionals and Loops*. 2025. Acesso em 14 de maio de 2025. Disponível em: ⟨https://v18.angular.dev/essentials/conditionals-and-loops⟩.

Angular Team. *Conditionals and Loops*. 2025. Acesso em 14 de maio de 2025. Disponível em: ⟨https://angular.dev/api/core/@switch⟩.

Angular Team. *Deferred loading with @defer*. 2025. Acesso em: 24 Junho 2025. Disponível em: ⟨https://angular.dev/guide/templates/defer⟩.

Angular Team. *Event binding*. 2025. Acessado em 22 de Maio de 2025. Disponível em: ⟨https://v17.angular.io/guide/event-binding⟩.

Angular Team. *Migration to lazy-loaded routes*. 2025. Acesso em: 26 Junho 2025. Disponível em: ⟨https://angular.dev/reference/migrations/route-lazy-loading⟩.

Angular Team. *Navigate to routes*. 2025. Acesso em: 26 Junho 2025. Disponível em: ⟨https://angular.dev/guide/routing/navigate-to-routes⟩.

Angular Team. *ng new*. [S.l.], 2025. Acessado em 22 de abril de 2025. Disponível em: ⟨https://v17.angular.io/cli/new⟩.

Angular Team. *Property Binding*. 2025. Acessado em 22 de Maio de 2025. Disponível em: ⟨https://angular.dev/essentials/templates#setting-dynamic-properties-and-attributes⟩.

Angular Team. *Read route state*. 2025. Acesso em: 26 Junho 2025. Disponível em: ⟨https://angular.dev/guide/routing/read-route-state⟩.

Angular Team. *RouterLink*. 2025. Acesso em: 26 Junho 2025. Disponível em: ⟨https://angular.dev/api/router/RouterLink⟩.

Angular Team. *RouterModule*. 2025. Acesso em: 26 Junho 2025. Disponível em: ⟨https://angular.dev/api/router/RouterModule⟩.

Angular Team. *Template Syntax – Data Binding*. 2025. Acesso em 04 de maio de 2025. Disponível em: ⟨https://angular.io/guide/binding-syntax⟩.

Angular Team. *Template Syntax – Interpolation*. 2025. Acesso em 04 de maio de 2025. Disponível em: ⟨https://angular.io/guide/interpolation⟩.

ANKUSH_953. *Views In Django — Python*. 2016. Acessado em 19 de Maio de 2023: ⟨https://www.geeksforgeeks.org/views-in-django-python/⟩.

ANTONIO. Requisitos não funcionais e arquitetura de software. *Revista Engenharia de Software*, n. 3, 2008. Acesso em 14 set. 2024. Disponível em: ⟨https://www.devmedia.com.br/artigo-engenharia-de-software-3-requisitos-nao-funcionais/9525⟩.

ANTONIOU, G. et al. Web ontology language: Owl. *Handbook on ontologies*, Springer, p. 91–110, 2009.

ANYWHERE, A. *Robotic Process Automation*. 2024. Acesso em: 8 ago. 2024. Disponível em: ⟨https://www.automationanywhere.com/br/rpa/robotic-process-automation⟩.

ANZBÖCK, R. et al. Modeling and implementing medical web services. *Data & Knowledge Engineering*, Elsevier, v. 55, n. 2, p. 203–236, 2005.

ARCANGELI, J.-P. et al. Automatic deployment of distributed software systems: Definitions and state of the art. *Journal of Systems and Software*, v. 103, p. 198–218, 2015. ISSN 0164-1212. Disponível em: ⟨https://www.sciencedirect.com/science/article/pii/S0164121215000308⟩.

ATLASSIAN. *BitButcket Tutorial -- git pull*. 2023. Disponível em: ⟨https://www.atlassian.com/git/tutorials/syncing/git-pull⟩. Acesso em: 09 jun. 2023.

AUTH0. *Auth0 Documentation -- Introduction to Identity and Access Management (IAM)*. 2023. Disponível em: ⟨https://auth0.com/docs/get-started/identity-fundamentals/identity-and-access-management⟩. Acesso em: 28 mai. 2023.

AXIOS. *AXIOS doc*. 2023. Acessado em 17 de Novembro de 2023: ⟨https://axios-http.com/ptbr/docs/intro⟩.

BAAREN, E. v. *Comparing 3 popular Python rest apis: Which One to pick?* 2024. Acesso em: 02 maio 2025. Disponível em: ⟨https://python.land/create-a-python-rest-api⟩.

BACHIEGA, N. G. et al. Container-based performance evaluation: A survey and challenges. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. [S.l.: s.n.], 2018. p. 398–403.

BARBOSA, E. F. et al. Introdução ao teste de software. *Minicurso apresentado no XIV Simpósio Brasileiro de Engenharia de Software (SBES 2000)*, 2000.

BARBOSA, F. et al. O teste de software no mercado de trabalho. *Tecnologias em Projeção*, v. 2, n. 1, 2011.

BARTIÉ, A. Garantia da qualidade de software: as melhores práticas de engenharia de software aplicadas à sua empresa. *Rio de Janeiro: Campus*, 2002.

BECHHOFER, S. et al. Owl web ontology language reference. *W3C recommendation*, World Wide Web Consortium.[WWW document] http://www.w3. org/TR/owl-ref, v. 10, n. 2, p. 1–53, 2004.

BECK, K. Embracing change with extreme programming. *Computer*, v. 32, n. 10, p. 70–77, 1999.

BEER, B. *Introducing GitHub: A Non-Technical Guide*. [S.l.]: O'Reilly Media, 2018.

BELLO, S. A. et al. Cloud computing in construction industry: Use cases, benefits and challenges. *Automation in Construction*, v. 122, p. 103441, 2021. ISSN 0926-5805. Disponível em: ⟨https://www.sciencedirect.com/science/article/pii/S0926580520310219⟩.

BENNETT, J. *Practical Django Projects*. Apress, 2009. ISBN 9781430219385. Disponível em: ⟨https://books.google.com.br/books?id=4KArzQEACAAJ⟩.

BERNERS-LEE, T. *Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor*. [S.l.]: Harper San Francisco, 1999.

BERNERS-LEE, T. et al. World-wide web: the information universe. *Internet Research*, MCB UP Ltd, v. 2, n. 1, p. 52–58, 1992.

BERNERS-LEE, T. et al. Tabulator: Exploring and analyzing linked data on the semantic web. In: ATHENS, GEORGIA. *Proceedings of the 3rd international semantic web user interaction workshop*. [S.l.], 2006. v. 2006, p. 159.

BERNERS-LEE, T. et al. *Hypertext transfer protocol–HTTP/1.0*. [S.l.], 1996.

BERNERS-LEE, T. et al. The semantic web. *Scientific american*, JSTOR, v. 284, n. 5, p. 34–43, 2001.

BERNERS-LEE, T. J. The world-wide web. *Computer networks and ISDN systems*, Elsevier, v. 25, n. 4-5, p. 454–459, 1992.

BIBI, N. et al. Web semantics and ontologies-based framework for software component selection from online repositories. *International Journal of Web and Grid Services*, Inderscience Publishers (IEL), v. 19, n. 3, p. 318–349, 2023.

BIEHL, M. *API Architecture*. CreateSpace Independent Publishing Platform, 2015. (API-University Series). ISBN 9781508676645. Disponível em: ⟨https://books.google.com.br/books?id=6D64DwAAQBAJ⟩.

BILAL, A. *What is an API marketplace?* 2023. Acesso em: 09 julho 2025. Disponível em: ⟨https://rapidapi.com/guides/api-marketplace⟩.

BIZER, C. et al. Linked data-the story so far. In: *Linking the World's Information: Essays on Tim Berners-Lee's Invention of the World Wide Web*. [S.l.: s.n.], 2023. p. 115–143.

BIZER, C. et al. Linked data on the web (ldow2008). In: *Proceedings of the 17th international conference on World Wide Web*. [S.l.: s.n.], 2008. p. 1265–1266.

BOARD, Q. *Standard glossary of terms used in Software Testing*. [S.l.]: Homepage, 2014.

BOUCHER, J. *Django Middleware: An overview of middleware*. 2016. Acessado em 19 de Maio de 2023: ⟨https://jodyboucher.com/blog/django-middleware-overview⟩.

BRADBURY, D. Data mining with linkedin. *Information and Software Technology*, Elsevier, v. 2011, p. 5–8, outubro 2011.

BRAGA, A. *Front-end e Back-end: o que você precisa saber para ter sucesso nessas áreas*. 2022. Acesso em: 28 nov. 2023. Disponível em: ⟨https://ateliware.com/blog/tipos-de-desenvolvedores?gad_source=1&gclid=CjwKCAiAvJarBhA1EiwAGgZl0B03Wyevb4ebpOKEs0OxywFqZ81IK7QRGlfAka6gYMU2OxoC0NoQAvD_BwE⟩.

BRASIL, S. E. de Comunicação Social Governo Federal do. *Manual de SEO: Otimização para Mecanismos de Busca*. [S.l.], 2020. Disponível em: ⟨https://www.gov.br/governodigital/pt-br/acesso-a-informacao/legislacao-e-normativos/guia-seo.pdf⟩.

BREITMAN, K. K. *Web Semântica - A Internet do Futuro*. [S.l.]: Grupo GEN, 2005.

BRITO, G. et al. Rest vs graphql: A controlled experiment. In: IEEE. *2020 IEEE international conference on software architecture (ICSA)*. [S.l.], 2020. p. 81–91.

BUGL, D. *Learn React Hooks: Build and refactor modern React. js applications using Hooks*. [S.l.]: Packt Publishing Ltd, 2019.

CABEZAS, I. et al. Using software engineering design principles as tools for freshman students learning. In: *2020 IEEE Frontiers in Education Conference (FIE)*. [S.l.: s.n.], 2020. p. 1–5.

Capterra. *Helix RM*. 2024. Acesso em: 13 de set. 2024. Disponível em: ⟨https://www.capterra.com.br/software/93101/seapine-alm⟩.

CARDOSO, L. d. C. *Frameworks Back End*. [S.l.]: Editora Saraiva, 2021.

CELESTI, A. et al. A study on join operations in mongodb preserving collections data models for future internet applications. *Future Internet*, v. 11, n. 4, 2019. ISSN 1999-5903. Disponível em: ⟨https://www.mdpi.com/1999-5903/11/4/83⟩.

CERIOLI, M. et al. What 5 million job advertisements tell us about testing: a preliminary empirical investigation. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. [S.l.: s.n.], 2020. p. 1586–1594.

CHACON, S. *Why Git is Better than X*. 2009. Disponível em: ⟨http://web.archive.org/web/20090210020404id_/http://whygitisbetterthanx.com/#the-staging-area⟩. Acesso em: 04 jun. 2023.

CHACON, S. et al. *Pro git*. [S.l.]: Apress, 2014. Acesso em: 03 jun. 2023.

CHAQFEH, M. et al. Towards a world wide web without digital inequality. *Proceedings of the National Academy of Sciences*, National Academy of Sciences, v. 120, n. 3, p. e2212649120, 2023.

CHEN, T.-H. Um estudo empírico sobre a prática de manutenção de código de mapeamento objeto-relacional em sistemas java. In: . [S.l.: s.n.], 2016. Acessado em 05-maio-2023, disponível em ⟨https://dl.acm.org/doi/abs/10.1145/2901739.2901758⟩.

CHERNY, B. *Programming TypeScript: Making Your JavaScript Applications Scale*. [S.l.]: O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2019. v. 2019.

CLOUD, G. *O que é uma máquina virtual?* 2023. ⟨https://cloud.google.com/learn/what-is-a-virtual-machine?hl=pt-br⟩. Online; acessado em Jun 2023.

CLOUD, G. *PaaS x IaaS xSaaS x CaaS: qual a diferença entre elas?* 2023. ⟨https://cloud.google.com/learn/paas-vs-iaas-vs-saas?hl=pt-br⟩. Online; acessado em Jun 2023.

CODD, E. F. *A Relational Model of Data for Large Shared Data Banks*. 1969. Acessado em 07-setembro-2023, disponível em ⟨https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf⟩.

CODING, H. *Aninhando tags HTML - Tutorial*. 2024. Disponível em: ⟨https://happycoding-io.translate.goog/tutorials/html/nesting-html-tags?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=wa⟩.

CONSORTIUM, W. W. W. *JSON-LD 1.1*. 2020. Disponível em: ⟨https://www.w3.org/TR/json-ld11/⟩. Acesso em: 04 set. 2024.

CONSORTIUM, W. W. W. *W3C History*. 2023. Disponível em: ⟨https://www.w3.org/about/history/⟩. Acesso em: 06 dez. 2023.

CONSORTIUM, W. W. W. et al. Extensible markup language (xml) 1.1. World Wide Web Consortium, 2006.

CONTENT, R. *JavaScript: o que é, como funciona e por que usá-lo no seu site*. 2019. ⟨https://rockcontent.com/br/blog/javascript/⟩.

CONTRIBUTORS, M. Javascript. *MDN Web Docs*, 2021. Acessado em 08 de maio de 2023. Disponível em: ⟨https://developer.mozilla.org/pt-BR/docs/Web/JavaScript⟩.

CORMAN, J. et al. Semantics and validation of recursive shacl. In: SPRINGER. *The Semantic Web–ISWC 2018: 17th International Semantic Web Conference, Monterey, CA, USA, October 8–12, 2018, Proceedings, Part I 17*. [S.l.], 2018. p. 318–336.

COYIER, C. *What Are The Benefits of Using a CSS Framework?* 2008. Acessado em 26 de fevereiro de 2025: ⟨https://css-tricks.com/what-are-the-benefits-of-using-a-css-framework/⟩.

CROZAT, B. Are css property names and values case sensitive? 2024. Disponível em: ⟨https://benjamincrozat.com/css-property-names-values-case-sensitive⟩.

Cubos Academy. *TypeScript x JavaScript: qual a diferença?* 2022. ⟨https://blog.cubos.academy/typescript-x-javascript-qual-a-diferenca/⟩. Online; acessado em 5 de junho de 2026.

CYGANIAK, R. et al. *RDF 1.1 Concepts and Abstract Syntax*. 2014. ⟨https://www.w3.org/TR/rdf11-concepts/⟩. W3C Recommendation.

CYPRESS. *Why Cypress?* 2023. Disponível em: ⟨https://docs.cypress.io/guides/overview/why-cypress⟩.

DARGHAM, J. et al. A navigational web requirements validation through animation. In: *Proceedings of The Third International Conference on Internet and Web Applications and Services (ICIW)*. [S.l.]: IEEE, 2008. p. 56–61. ISBN 978-0-7695-3163-2.

DATTA, D. *What is SQLAlchemy used for? an overview*. CData Software, 2024. Acesso em: 13 maio 2025. Disponível em: ⟨https://www.cdata.com/blog/what-is-sqlalchemy⟩.

DAVIS, M. M. et al. *Fundamentos da administração da produção*. [S.l.]: Bookman, 2001.

DBENGINES. *DB-Engines Rankin*. 2023. Acessado em 25-abril-2023, disponível em ⟨https://db-engines.com/en/ranking⟩.

DEBOIS, P. Agile infrastructure and operations: How infra-gile are you? In: *Agile 2008 Conference*. [S.l.: s.n.], 2008. p. 202–207.

DEBOIS, P. *Devopsdays-Organizing Guide*. 2009.

DECKER, S. et al. Framework for the semantic web: an rdf tutorial. *IEEE Internet Computing*, IEEE, v. 4, n. 6, p. 68–73, 2000.

DELAMARO, M. et al. *Introdução ao teste de software*. [S.l.]: Elsevier Brasil, 2017.

DESENVOLVEDOR casa do. *Qualidade do software: por que é tão importante?* 2021. Acesso em: 28 nov. 2023. Disponível em: ⟨https://blog.casadodesenvolvedor.com.br/qualidade-do-software/⟩.

DevMedia. *Tipos primitivos e variáveis em Java*. 2006. ⟨https://www.devmedia.com.br/tipos-primitivos-e-variaveis-em-java/3149⟩. Online; acessado em 5 de junho de 2026.

DEVPLENO. *Quais são as vantagens de usar JavaScript em todas as camadas de uma aplicação?* 2017. ⟨https://devpleno.com/quais-sao-as-vantagens-de-usar-javascript-em-todas-as-camadas-de-uma-aplicacao⟩. Online; acessado em 5 de junho de 2026.

DIAS, T. D. et al. Web semântica: conceitos básicos e tecnologias associadas. *Cadernos do IME-Série Informática*, v. 14, p. 80–92, 2003.

DICODING. *O que são serviços da Web? Junto com definições e exemplos*. 2023. Acessado em 13-junho-2023, disponível em ⟨https://www.dicoding.com/blog/apa-itu-web-service/⟩.

DIGITALOCEAN. How to nest html elements. 2024. Disponível em: ⟨https://www.digitalocean.com/community/tutorials/how-to-nest-html-elements⟩.

DIO. Qual a estrutura básica do html? 2024. Disponível em: ⟨https://www.dio.me/articles/qual-a-estrutura-basica-do-html⟩.

DIO.ME. *Quality Assurance (QA)*. 2023. Acesso em: 28 nov. 2023. Disponível em: ⟨https://www.dio.me/technologies/quality-assurance-qa⟩.

DJANGO. *Documentação Django*. 2022. Acessado em 20-abril-2023, disponível em ⟨https://www.djangoproject.com⟩.

DJANGO. *Django*. 2023. Acessado em 28-maio-2023, disponível em: ⟨https://www.djangoproject.com/⟩.

DOCKER. *Guides -- Overview*. 2020. Disponível em: ⟨https://docs.docker.com/get-started/docker-overview⟩. Acesso em: 30 mai. 2023.

Docker, Inc. *Docker Overview*. 2024. Accessed: 2025-06-06. Disponível em: ⟨https://docs.docker.com/get-started/docker-overview/⟩.

Docker, Inc. *Docker Hub Overview*. 2025. Captura de tela da interface do Docker Hub. Acesso em: 9 jul. 2025. Disponível em: ⟨https://hub.docker.com/⟩.

DOCS, M. W. *Tutorial Django Parte 8: Autenticação de usuário e permissões*. 2024. Acessado em 06-setembro-2024, disponível em: ⟨https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Authentication⟩.

DODDS, K. C. *How to use React Context effectively*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://kentcdodds.com/blog/how-to-use-react-context-effectively⟩.

DOVER, D. et al. *Search Engine Optimization (SEO) Secrets*. [S.l.]: Wiley, 2011. ISBN 9780470554180.

DRIESSEN, V. *A successful Git branching model*. 2010. ⟨https://nvie.com/posts/a-successful-git-branching-model/⟩. Acesso em: 19 maio 2025.

DUARTE, A. N. et al. Uma abordagem baseada em testes automáticos de software para diagnóstico de faltas em grades computacionais. Universidade Federal de Campina Grande, 2010.

Dublin Core Metadata Initiative (DCMI). *Dublin Core Metadata Element Set, Version 1.1*. 2020. ⟨https://www.dublincore.org/specifications/dublin-core/dces/⟩. Acesso em: 31 jul. 2025.

DYCK, A. et al. Towards definitions for release engineering and devops. In: *2015 IEEE/ACM 3rd International Workshop on Release Engineering*. [S.l.: s.n.], 2015. p. 3–3.

DZIEKANIAK, G. V. et al. Web semântica. 2004.

EBERT, C. et al. Devops. *Ieee Software*, IEEE, v. 33, n. 3, p. 94–100, 2016.

ECCHER, C. *Professional Web Design: Techniques and Templates*. [S.l.]: Cengage Learning, 2015.

EIS, D. *Introdução à Web Semântica: A inteligência da informação*. [S.l.]: Editora Casa do Código, 2017.

ELEMENTOR. Como fazer indentação em html. 2024. Disponível em: ⟨https://elementor.com/blog/pt-br/como-fazer-indentacao-em-html/⟩.

Embarcados. *Tipos de dados para uso em algoritmos*. 2022. ⟨https://embarcados.com.br/tipos-de-dados/⟩. Online; acessado em 5 de junho de 2026.

ENGE, E. et al. *The Art of SEO: Mastering Search Engine Optimization*. 4. ed. [S.l.]: O'Reilly Media, 2022. ISBN 9781098102616.

FACEBOOK. *Announcing Facebook Connect*. 2008. Disponível em: ⟨https://developers.facebook.com/blog/post/2008/05/09/announcing-facebook-connect/⟩.

FENSEL, D. et al. *Knowledge graphs*. [S.l.]: Springer, 2020.

FIELDING, R. et al. *Hypertext transfer protocol–HTTP/1.1*. [S.l.], 1999.

FIELDING, R. et al. *Hypertext transfer protocol (HTTP/1.1): range requests*. [S.l.], 2014.

FIELDING, R. et al. *Hypertext transfer protocol (HTTP/1.1): Caching*. [S.l.], 2014.

FIELDING, R. et al. *Hypertext transfer protocol (http/1.1): Authentication*. [S.l.], 2014.

FIELDING, R. et al. *Hypertext transfer protocol (HTTP/1.1): Conditional requests*. [S.l.], 2014.

FIELDING, R. et al. *Hypertext transfer protocol (HTTP/1.1): Message syntax and routing*. [S.l.], 2014.

FIELDING, R. et al. *Hypertext transfer protocol (HTTP/1.1): Semantics and content*. [S.l.], 2014.

FIELDING, R. T. *Architectural styles and the design of network-based software architectures*. [S.l.]: University of California, Irvine, 2000.

FLEURY, M. T. L. et al. Construindo o conceito de competência. *Revista de Administração Contemporânea*, v. 5, n. spe, p. 183–196, setembro 2001. ISSN 1415-6555.

FOWLER, M. *Continuous Delivery*. 2010. ⟨https://martinfowler.com/bliki/ContinuousDelivery.html⟩.

FOWLER, M. *Blue/Green Deployment*. 2014. Disponível em: ⟨https://martinfowler.com/bliki/BlueGreenDeployment.html⟩.

FOWLER, M. *DevOps Culture*. 2016. Disponível em: ⟨https://martinfowler.com/bliki/DevOpsCulture.html⟩.

FOWLER, M. The practical test pyramid. *martinfowler. com.[Online]. Available: https://martinfowler. com/articles/practical-test-pyramid. html*, 2018.

FOWLER, M. *Continuous Integration*. 2023. ⟨https://martinfowler.com/articles/continuousIntegration.html⟩.

FOWLER, M. et al. *Microservices: a definition of this new architectural term*. 2014. ⟨https://martinfowler.com/articles/microservices.html⟩. Accessed: 2024-11-27.

FREECODECAMP. How to indent in html and why it is important. 2024. Disponível em: ⟨https://www.freecodecamp.org/news/how-to-indent-in-html-and-why-it-is-important/⟩.

FREEMAN, A. et al. *Pro Angular 16*. [S.l.]: Manning Publications Co., 2024. ISBN 9781633436695; 1633436691.

FREEMAN, E. *DevOps Para Leigos*. E-book. Rio de Janeiro: Editora Alta Books, 2021. ISBN 9788550816661. Disponível em: ⟨https://app.minhabiblioteca.com.br/#/books/9788550816661/⟩.

GABARDO, A. C. *Laravel para ninjas*. [S.l.]: Novatec Editora, 2017.

Gaea. *Entenda o que são variáveis na programação e saiba como lidar com elas*. 2022. ⟨https://gaea.com.br/variaveis-programacao/⟩. Online; acessado em 5 de junho de 2026.

GALVÃO, P. Html, css e javascript: o que são e quais as diferenças. *PG Tech*, 2024. Disponível em: ⟨https://tech.pedrogalvao.com/dev/html-css-javascript/⟩.

GARCIA, J. E. et al. A requirements-to-implementation mapping tool for requirements traceability. *J. Softw.*, v. 11, n. 2, p. 193–200, 2016.

GARCIA-MOLINA, H. et al. *Database Systems: The Complete Book*. 2nd. ed. [S.l.]: Pearson, 2008. ISBN 9780131873254.

GARCíA, B. et al. A survey of the selenium ecosystem. *Electronics*, v. 9, n. 7, 2020. ISSN 2079-9292. Disponível em: ⟨https://www.mdpi.com/2079-9292/9/7/1067⟩.

GARTNER. *Magic Quadrant for Cloud Infrastructure and Platform Services*. 2022. ⟨https://www.gartner.com/doc/reprints?id=1-2AOZQAQL&ct=220728&st=sb&refid=d655fed1-d452-405a-8dc6-682ee5154f83⟩. Online; acessado em Jun 2023.

GEEKSFORGEEKS. Are html5 tags case sensitive? *GeeksforGeeks*, 2022. Disponível em: ⟨https://www.geeksforgeeks.org/are-html5-tags-case-sensitive/⟩.

GEEKSFORGEEKS. *Django Project MVT Structure*. 2023. Acessado em 20-abril-2023, disponível em ⟨https://www.geeksforgeeks.org/django-project-mvt-structure/⟩.

GeeksforGeeks. *Explain the Architecture Overview of Angular*. 2023. ⟨https://www.geeksforgeeks.org/explain-the-architecture-overview-of-angular/⟩. Last updated: 03 Nov 2023.

GILLIS, A. S. et al. *What is remote procedure call (RPC)?: Definition from TechTarget*. TechTarget, 2024. Acesso em: 27 ago. 2024. Disponível em: ⟨https://www.techtarget.com/searchapparchitecture/definition/Remote-Procedure-Call-RPC⟩.

GITHUB. *GitHub Importer*. 2023. Disponível em: ⟨https://docs.github.com/en/migrations/importing-source-code/using-github-importer/about-github-importer⟩. Acesso em: 18 jun. 2023.

GITHUB. *GitHub Flow*. 2025. Acesso em: 19 maio 2025. Disponível em: ⟨https://githubflow.github.io/⟩.

GitHub Docs. *Compreendendo o GitHub Actions*. 2024. Acessado em: 14 abr. 2025. Disponível em: ⟨https://docs.github.com/pt/actions/about-github-actions/understanding-github-actions⟩.

GITKRAKEN. *Best Practices -- Writing a Good Git Commit Message*. 2023. Disponível em: ⟨https://www.gitkraken.com/learn/git/best-practices/git-commit-message⟩. Acesso em: 05 jun. 2023.

DRF. *Documentação do Django REST Framewok*. 2023. Acessado em 09-junho-2023, disponível em ⟨https://www.django-rest-framework.org/⟩.

GONZALEZ, R. S. S. et al. Automation tools for web testing. beyond selenium. *Memorias de las JAIIO*, v. 8, n. 3, 2022.

GOOGLE. *gRPC overview*. Google, 2024. Acesso em: 28 ago. 2024. Disponível em: ⟨https://cloud.google.com/api-gateway/docs/grpc-overview⟩.

GOTARDO, R. Linguagem de programação. *Rio de Janeiro: Seses*, 2015.

GOURLEY, D. et al. *HTTP: the definitive guide*. [S.l.]: "O'Reilly Media, Inc.", 2002.

GOYAL, S. et al. Evolution and advancements of the world wide web: From concept to global phenomenon. In: IEEE. *2023 International Conference on Research Methodologies in Knowledge Management, Artificial Intelligence and Telecommunication Engineering (RMKMATE)*. [S.l.], 2023. p. 1–8.

GRPC. *Introduction to gRPC*. 2024. Acesso em: 28 ago. 2024. Disponível em: ⟨https://grpc.io/docs/what-is-grpc/introduction/⟩.

GRUBER, T. R. A translation approach to portable ontology specifications. *Knowledge acquisition*, Elsevier, v. 5, n. 2, p. 199–220, 1993.

GUARINO, N. et al. What is an ontology? *Handbook on ontologies*, Springer, p. 1–17, 2009.

GUERRA, P. *Modelagem de Requisitos de Software*. 2009. Acessado em: 13 set. 2024. Disponível em: ⟨https://www.mindmeister.com/pt/28238820/modelagem-de-requisitos-de-software⟩.

GUTIERREZ, C. et al. Introducing time into rdf. *IEEE transactions on Knowledge and Data Engineering*, IEEE, v. 19, n. 2, p. 207–218, 2006.

HAFENSCHERER, M. et al. Robotic process automation of calculating investments in a business project. *ICSCA: Software and Computer Applications*, ACM, p. 309–314, Junho 2023.

HAOYU, W. et al. Basic design principles in software engineering. In: *2012 Fourth International Conference on Computational and Information Sciences*. [S.l.: s.n.], 2012. p. 1251–1254.

HAT, R. *O que é função como serviço (FaaS)?* 2020. ⟨https://www.redhat.com/pt-br/topics/cloud-native-apps/what-is-faas⟩. Online; acessado em Jun 2023.

HAT, R. *O que é CI/CD?* 2025. Acessado em: 18 mar. 2025. Disponível em: ⟨https://www.redhat.com/pt-br/topics/devops/what-is-ci-cd⟩.

HEATH, F. *Managing Software Requirements the Agile Way: Bridge the gap between software requirements and executable specifications to deliver successful projects*. [S.l.]: Packt Publishing Ltd, 2020.

HEATH, T. Linked data: Evolving the web into a global data space. *Morgan & Claypool*, 2011.

HENDLER, J. Is there an intelligent agent in your future? *Nature*, Springer Science and Business Media LLC, v. 11, 1999.

HENDLER, J. Agents and the semantic web. *IEEE Intelligent systems*, IEEE, v. 16, n. 2, p. 30–37, 2001.

HEUMANN, J. Generating test cases from use cases. *The rational edge*, v. 6, n. 01, 2001.

HITZLER, P. A review of the semantic web field. *Communications of the ACM*, ACM New York, NY, USA, v. 64, n. 2, p. 76–83, 2021.

HOLOVATY, A. et al. *The definitive guide to Django: Web development done right*. [S.l.]: Apress, 2009.

HOSTINGER. O que é css: Guia básico de css. 2024. Disponível em: ⟨https://www.hostinger.com.br/tutoriais/o-que-e-css-guia-basico-de-css⟩.

HUMBLE, J. *Continuous Delivery vs Continuous Deployment*. 2010. ⟨https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/⟩.

HUMBLE, J. et al. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. [S.l.]: Addison-Wesley, 2010. ISBN 9780321601919.

HUMBLE, J. et al. The deployment production line. In: *Proceedings of AGILE 2006 Conference (AGILE'06)*. [S.l.]: IEEE, 2006. p. 6–14.

IANNELLA, R. et al. *Metadata: enabling the Internet*. [S.l.]: DSTC Pty Limited, 1997.

IBM. *SQL Documentation*. 2023. Acessado em 05-maio-2023, disponível em ⟨https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview⟩.

IBM. *Understanding threads and processes*. 2023. Disponível em: ⟨https://www.ibm.com/docs/en/aix/7.2?topic=programming-understanding-threads-processes⟩. Acesso em: 26 jun. 2023.

IBM. *Remote Procedure Call*. 2024. Acesso em: 20 ago. 2024. Disponível em: ⟨https://www.ibm.com/docs/en/aix/7.3?topic=concepts-remote-procedure-call⟩.

IMASTER. *Confira quais foram as linguagens de programação mais usadas em 2022*. 2023. Acessado em 25-maio-2023, disponível em ⟨https://imasters.com.br/noticia/confira-quais-foram-as-linguagens-de-programacao-mais-usadas-em-2022⟩.

iMasters. *Confira quais foram as linguagens de programação mais usadas em 2022*. 2023. ⟨https://imasters.com.br/noticia/confira-quais-foram-as-linguagens-de-programacao-mais-usadas-em-2022⟩.

INC., D. *Use containers to Build, Share and Run your applications*. 2023. ⟨https://www.docker.com/resources/what-container/⟩. Online; acessado em Jun 2023.

INGALLS, S. *Microservices vs. monolithic architecture*. 2021. Acessado em 25 de Junho de 2023: ⟨https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith⟩.

INGALLS, S. *What Is a Client-Server Model? A Guide to Client-Server Architecture*. 2021. Acessado em 31 de Maio de 2023: ⟨https://www.serverwatch.com/guides/client-server-model/⟩.

INSOMNIA. *Insomnia*. 2023. Acessado em 05 de julho de 2023: ⟨https://insomnia.rest/⟩.

INSOMNIA. *Introduction to Insomnia*. 2025. Acesso em: 09 julho 2025. Disponível em: ⟨https://docs.insomnia.rest/insomnia/get-started⟩.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS (IEEE). *Systems and software engineering – Life cycle processes – Requirements engineering*. New York, NY, 2011.

IONOS, D. G. *SQL JOIN: consultas a varias tablas de datos*. 2018. Acessado em 07-setembro-2023, disponível em ⟨https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/sql-join/⟩.

ISO/IEC/IEEE:24765. Iso/iec/ieee international standard - systems and software engineering – vocabulary. In: . [S.l.: s.n.], 2017. p. 1–541.

ISOTANI, S. et al. Estado da arte em web semântica e web 2.0: potencialidades e tendências da nova geração de ambientes de ensino na internet. *Revista brasileira de informática na educação*, v. 17, n. 01, p. 30, 2009.

IYENGAR, J. et al. *QUIC: A UDP-Based Multiplexed and Secure Transport. draft-ietf-quic-transport-13. IETF*. 2018.

JABBARI, R. et al. What is devops? a systematic mapping study on definitions and practices. In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. [S.l.: s.n.], 2016. p. 1–11.

JACOBSON, D. et al. *APIs: A Strategy Guide*. 1. ed. [S.l.]: O'Reilly Media, 2011.

JIN, B. et al. *Designing Web APIs: Building APIs that Developers Love*. O'Reilly Media, 2018. ISBN 9781492026921. Disponível em: ⟨https://books.google.com.br/books?id=8jUrtAEACAAJ⟩.

JIRAPANTHONG, W. Requirements traceability on web applications. In: IEEE. *2015 7th International Conference on Information Technology and Electrical Engineering (ICITEE)*. [S.l.], 2015. p. 18–23.

JOARDER, Y. A. et al. A survey on the security issues of quic. In: *2022 6th Cyber Security in Networking Conference (CSNet)*. [S.l.: s.n.], 2022. p. 1–8.

JOHNSON, H. A. Trello. *Journal of the Medical Library Association*, v. 105, n. 2, p. 118–121, April 2017. Disponível em: ⟨https://jmla.pitt.edu/ojs/jmla/article/view/49⟩.

JOSE, B. et al. Performance analysis of nosql and relational databases with mongodb and mysql. *Materials Today: Proceedings*, v. 24, p. 2036–2043, 2020. ISSN 2214-7853. International Multi-conference on Computing, Communication, Electrical & Nanotechnology, I2CN-2K19, 25th & 26th April 2019. Disponível em: ⟨https://www.sciencedirect.com/science/article/pii/S2214785320324159⟩.

KARUNARATHNA, I. et al. *Biblioteca -- O que é o WSO2 Identity Server?* 2017. Disponível em: ⟨https://wso2.com/library/articles/2017/08/o-que-e-o-wso2-identity-server⟩. Acesso em: 25 mai. 2023.

KEPNER, C. et al. O administrador racional: Uma abordagem sistemática para solução de problemas e tomada de decisão. 2 ª edição. *São Paulo*, 1972.

KHASAWNEH, T. N. et al. *2020 11th International Conference on Information and Communication Systems (ICICS)*. [S.l.: s.n.], 2020. Acessado em 04-maio-2023, disponível em ⟨https://ieeexplore.ieee.org/abstract/document/9078970?casa_token=Vx-Q4r5mySkAAAAA:HmOm018kf_Ff2SCdNEtDu3Y-WQsBpiPg1q7442uALW7NAUVjk8p8YmI5YAhXwEjm1Xf4wV45EA⟩.

KHETARPAL, A. *What is Cypress? Cypress Architecture, Features and Introduction*. TOOLSQA, 2021. Disponível em: ⟨https://www.toolsqa.com/cypress/what-is-cypress/⟩.

KHOLMATOV, A. WIDELY USED LIBRARIES IN THE JAVASCRIPT PROGRAMMING LANGUAGE AND THEIR CAPABILITIES. *Intent Research Scientific Journal*, 2023.

KIM, G. et al. *Manual de DevOps: Como obter agilidade, confiabilidade e segurança em organizações tecnológicas*. [S.l.]: Alta Books, 2018. ISBN 978-8550802695.

KINSTA. Melhores práticas de html. 2024. Disponível em: ⟨https://kinsta.com/pt/blog/melhores-praticas-html/⟩.

KODALI, N. The evolution of angular cli and schematics: Enhancing developer productivity in modern web applications. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, v. 10, n. 5, p. 805–812, 2024.

KONNORATE, G. C. et al. A importancia do controle de versões no desenvolvimento de software. *Seminário de Tecnologia Gestão e Educação*, v. 1, n. 2, p. 1–4, 2019.

KOPPALA, J. *ERP Solution with ReactJS*. [S.l.]: TBD Publishing, 2021. Chapter 3: The evolution from XHP to ReactJS.

KOSINDRDECHA, N. et al. A test case generation technique and process. In: *International Workshop on Design, Evaluation and Refinement of Intelligent Systems (DERIS2010)*. [S.l.: s.n.], 2010. p. 59.

KRYLOV, G. et al. The evolution of garbage collection in v8: Google's javascript engine. In: *2020 9th Mediterranean Conference on Embedded Computing (MECO)*. [S.l.: s.n.], 2020. p. 1–6.

KUMAR, S. A review on client-server based applications and research opportunity. *International Journal of Recent Scientific Research*, v. 10, n. 7, p. 33857–3386, 2019.

KUROSE, J. et al. *Computer networks: A top down approach featuring the internet*. [S.l.]: Pearson Addison Wesley, 2010.

KYAW, N. N. Analysis and simulation of hyper text transfer protocol at the application layer of the internet. *International Journal of Scientific and Research Publications (IJSRP)*, v. 9, n. 1, p. 10–29322, 2019.

LADAS, C. *Scrumban - Essays on Kanban Systems for Lean Software Development*. Modus Cooperandi Press, 2009. (Modus Cooperandi lean series). ISBN 9780578002149. Disponível em: ⟨https://books.google.com.br/books?id=SQFdAgAAQBAJ⟩.

LAIRD, L. M. et al. *Software measurement and estimation: a practical approach*. [S.l.]: John Wiley & Sons, 2006.

LANGLEY, A. et al. The quic transport protocol: Design and internet-scale deployment. In: *Proceedings of the conference of the ACM special interest group on data communication*. [S.l.: s.n.], 2017. p. 183–196.

LARAVEL. *Laravel*. 2023. Acessado em 15-junho-2023, disponível em: ⟨https://laravel.com/⟩.

LAWSON, B. et al. *Introducing html5*. [S.l.]: New Riders, 2011.

LAZUARDY, M. F. S. et al. Modern front end web architectures with react. js and next. js. *Research Journal of Advanced Engineering and Science*, v. 7, n. 1, p. 132–141, 2022.

LEFFINGWELL, D. *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. [S.l.]: Addison-Wesley Professional, 2010.

LEITE, L. et al. A survey of devops concepts7 and challenges. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 52, n. 6, p. 1–35, 2019.

LEVINE, K. J. et al. Introducing the sixth source of vocational anticipatory socialization: Using the internet to search for career information. *Journal of Career Development*, v. 49, n. 2, p. 443–456, julho 2020.

LEYENDECKER, D. *Os três níveis da modelagem de dados: conceitual, lógico e físico*. 2023. Acessado em 07-setembro-2023, disponível em ⟨https://www.dio.me/articles/os-tres-niveis-da-modelagem-de-dados-conceitual-logico-e-fisico⟩.

LI, J. et al. How does web service api evolution affect clients? In: IEEE. *2013 IEEE 20th International Conference on Web Services*. [S.l.], 2013. p. 300–307.

LI, Z. et al. Performance overhead comparison between hypervisor and container based virtualization. In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. [S.l.: s.n.], 2017. p. 955–962.

LIANG, L. et al. Express supervision system based on nodejs and mongodb. In: *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*. [S.l.: s.n.], 2017. p. 607–612.

LIBUV. *Documentação Libuv*. 2014. Acessado em 11-junho-2023, disponível em ⟨https://docs.libuv.org/en/v1.x/design.html#the-i-o-loop⟩.

LINKEDIN. *LinkedIn Newsroom Retrieved*. 2023. Acesso em: 25 nov. 2023. Disponível em: ⟨https://news.linkedin.com/about-us#Statistics⟩.

LINKEDIN. *Procurar vagas no LinkedIn*. 2023. Acesso em: 28 nov. 2023. Disponível em: ⟨https://www.linkedin.com/help/linkedin/answer/a512245?hcppcid=search⟩.

LOMBARDO, E. *Especificação e Modelagem de Requisitos*. Acessado em: 13 set. 2024. Disponível em: ⟨https://slideplayer.com.br/slide/353858/⟩.

MA, S. Q. et al. The impacts of personality traits, use intensity and features use of linkedin on bridging social capital. v. 14, p. 1059–1078, setembro 2019.

MACHADO, O. *Ferramentas de testes de aceitação: Uma Odisséia.* Medium, 2017. Disponível em: ⟨https://medium.com/@otavio/ferramentas-de-testes-de-aceita%C3%A7%C3%A3o-uma-odiss%C3%A9ia-8eff4c96da9e⟩.

BARROS MACIEL, F. de. *Python e Django: desenvolvimento web moderno e ágil.* Alta Books, 2020. ISBN 9786555200973. Disponível em: ⟨https://books.google.com.br/books?id=Td_aDwAAQBAJ⟩.

MACIEL, R. S. P. et al. *Uma solução para o desenvolvimento de aplicações distribuídas.* [S.l.]: CienteFico, 2004.

MACORATTI. *Conceitos Básicos de modelagem de dados.* 2023. Acessado em 11-maio-2023, disponível em ⟨https://www.macoratti.net/cbmd1.htm⟩.

MADURAPPERUMA, I. et al. State-of-art frameworks for front-end and back-end web development. Faculty of Technology, South Eastern University of Sri Lanka, Sri Lanka, 2022.

MAHAJAN, P. et al. Automation testing in software organization. *International Journal of Computer Applications Technology and Research*, v. 5, n. 4, p. 198–201, 2016.

MALDONADO, J. C. et al. Critérios potenciais usos: Uma contribuição ao teste estrutural de software. *PhD thesis, DCA/FEE/UNICAMP, Campinas, SP*, July 1991.

MANOLA, F. et al. Rdf primer. *W3C recommendation*, Citeseer, v. 10, n. 1-107, p. 6, 2004.

MARCOTTE, E. *Responsive Web Design.* [S.l.]: A Book Apart, 2011.

MARTIN, R. C. *Código Limpo.* São Paulo: Alta Books, 2008.

MARTIN, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* 1st. ed. USA: Prentice Hall Press, 2017. ISBN 0134494164.

MARTIN, R. C. *Arquitetura Limpa.* [*S. l.*]: Alta Books, 2019. ISBN 9788550808161.

MASSE, M. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces.* O'Reilly Media, 2011. ISBN 9781449319908. Disponível em: ⟨https://books.google.com.br/books?id=eABpzyTcJNIC⟩.

AL MASUD, S. M. R. et al. Devops enabled agile: Combining agile and devops methodologies for software development. *International Journal of Advanced Computer Science and Applications*, Science and Information (SAI) Organization Limited, v. 13, n. 11, 2022.

MATEUS, L. *Quem é melhor? ORM ou SQL puro?* 2023. Acessado em 07-setembro-2023, disponível em ⟨https://dev.to/lucas_jdev/quem-e-melhor-orm-ou-sql-puro-kjb⟩.

MCBRIDE, B. The resource description framework (rdf) and its vocabulary description language rdfs. In: *Handbook on ontologies*. [S.l.]: Springer, 2004. p. 51–65.

MCGUINNESS, D. L. et al. Owl web ontology language overview. *W3C recommendation*, v. 10, n. 10, p. 2004, 2004.

MDN. *Manipulating documents*. 2024. Acessado em 1 de maio de 2024: ⟨https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Manipulating_documents⟩.

MDN. *Rendering engine*. 2024. Acessado em 1 de maio de 2024: ⟨https://developer.mozilla.org/en-US/docs/Glossary/Rendering_engine⟩.

MELL, P. et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National ..., 2011.

MENDEZ, S. *The Test Process and Important Testing Techniques*. 2011.

META. *Legacy React Documentation*. Meta, 2023. Disponível em: ⟨https://legacy.reactjs.org/⟩.

META. *Component State*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://legacy.reactjs.org/docs/faq-state.html⟩.

META. *Composition vs Inheritance*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://legacy.reactjs.org/docs/composition-vs-inheritance.html⟩.

META. *Conditional Rendering*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://react.dev/learn/conditional-rendering⟩.

META. *Getting Started Create React App*. 2025. Acessado em 25 de fevereiro de 2025: ⟨https://create-react-app.dev/docs/getting-started/⟩.

META. *Hook useRef*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://react.dev/reference/react/useRef⟩.

META. *Passing Props to a Component*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://react.dev/learn/passing-props-to-a-component⟩.

META. *State: A Component's Memory*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://react.dev/learn/state-a-components-memory⟩.

META. *Virtual DOM and Internals*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://legacy.reactjs.org/docs/faq-internals.html⟩.

META. *Writing Markup with JSX*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://react.dev/learn/writing-markup-with-jsx⟩.

META. *Your First Component*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://react.dev/learn/your-first-component⟩.

MICHAEL, B. et al. Ruby on rails. *IEEE software*, v. 6, n. 24, p. 105–108, 2007.

MICROSOFT. *Arquitetura do .NET*. 2023. Acessado em 27-abril-2023, disponível em ⟨https://learn.microsoft.com/pt-br/dotnet/csharp/tour-of-csharp/#net-architecture⟩.

MICROSOFT. *Documentação do Subsistema do Windows para Linux*. 2023. Acessado em 25-maio-2023, disponível em ⟨https://aws.amazon.com/pt/what-is/database/⟩.

MICROSOFT. *Entity Framework Documentation*. 2023. Acessado em 05-maio-2023, disponível em ⟨https://learn.microsoft.com/en-us/ef/⟩.

MICROSOFT. *O que são nuvens públicas, privadas e híbridas?* 2023. ⟨https://azure.microsoft.com/pt-br/resources/cloud-computing-dictionary/what-are-private-public-hybrid-clouds/#overview⟩. Online; acessado em Jun 2023.

MICROSOFT. *O que é uma VM (máquina virtual)?* 2023. ⟨https://azure.microsoft.com/pt-br/resources/cloud-computing-dictionary/what-is-a-virtual-machine⟩. Online; acessado em Jun 2023.

MICROSOFT. *Power Automate Tour Guiado*. 2023. Acesso em: 28 nov. 2023. Disponível em: ⟨https://powerautomate.microsoft.com/pt-br/guidedtour/power-platform/power-automate/2/5⟩.

Microsoft Docs. *Razor Syntax Reference*. 2025. Acesso em 14 de maio de 2025. Disponível em: ⟨https://docs.microsoft.com/aspnet/core/mvc/views/razor⟩.

MILES, A. et al. *SKOS Simple Knowledge Organization System Reference*. United States: World Wide Web Consortium, 2009. (W3C Recommendation).

MILES, A. et al. Skos core: simple knowledge organisation for the web. In: DUBLIN CORE METADATA INITIATIVE. *Proceedings of the International Conference on Dublin Core and Metadata Applications*. [S.l.], 2005.

MILETTO, E. M. et al. *Desenvolvimento de Software II: Introdução ao Desenvolvimento Web com HTML, CSS, JavaScript e PHP-Eixo: Informação e Comunicação-Série Tekne*. [S.l.]: Bookman Editora, 2014.

MOLINARI, L. *Testes de Software: Produzindo Sistemas melhores e mais confiáveis*. [S.l.: s.n.], 2008. v. 4.

MONGODB. *MongoDB Documentation*. 2023. Acessado em 28-maio-2023, disponível em ⟨https://www.mongodb.com/docs/⟩.

MONTANDON, J. E. et al. What skills do IT companies look for in new developers? A study with stack overflow jobs. *Information and Software Technology*, Elsevier, v. 129, Janeiro 2021.

MOZDEVNET. *Authorization - http: MDN*. 2023. Acesso em: 09 julho 2025. Disponível em: ⟨https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization⟩.

MOZDEVNET. *Cache-Control - http: MDN*. 2023. Acesso em: 09 julho 2025. Disponível em: ⟨https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Headers/Cache-Control⟩.

MOZDEVNET. *Identificando recursos na web - http: MDN*. 2023. Acesso em: 09 julho 2025. Disponível em: ⟨https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web⟩.

MOZDEVNET. *Métodos de requisição HTTP - http: MDN*. 2023. Acesso em: 09 julho 2025. Disponível em: ⟨https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods⟩.

MOZILLA. *The web and web standards*. 2021. Disponível em: ⟨https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/The_web_and_web_standards⟩. Acesso em: 27 abr. 2023.

MOZILLA. *AJAX*. 2023. Acessado em 11 de Novembro de 2023: ⟨https://developer.mozilla.org/pt-BR/docs/conflicting/Web/Guide/AJAX⟩.

MOZILLA. *Introdução a WEB*. 2023. Acessado em 25-maio-2023, disponível em ⟨https://developer.mozilla.org/pt-BR/docs/Learn/Getting_started_with_the_web⟩.

MOZILLA. *Uma visão Geral do HTTP*. 2023. Acessado em 05-maio-2023, disponível em ⟨https://www.ibm.com/docs/en/i/7.2?topic=programming-sql⟩.

MOZILLA. *CSS*. 2025. Acessado em 25 de fevereiro de 2025: ⟨https://developer.mozilla.org/pt-BR/docs/Web/CSS⟩.

MOZILLA. *Document Object Model (DOM)*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model⟩.

MOZILLA. *HTML: Linguagem de Marcação de Hipertexto*. 2025. Acessado em 25 de fevereiro de 2025: ⟨https://developer.mozilla.org/pt-BR/docs/Web/HTML⟩.

MOZILLA. *import*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/export⟩.

MOZILLA. *import*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import⟩.

MOZILLA. *Javascript*. 2025. Acessado em 25 de fevereiro de 2025: ⟨https://developer.mozilla.org/pt-BR/docs/Web/JavaScript⟩.

MOZILLA. *MVC*. 2025. Acessado em 25 de fevereiro de 2025: ⟨https://developer.mozilla.org/en-US/docs/Glossary/MVC⟩.

MOZILLA. *Package management basics*. 2025. Acessado em 25 de fevereiro de 2025: ⟨https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Client-side_tools/Package_management⟩.

MOZILLA. *SPA (Single-page application)*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://developer.mozilla.org/en-US/docs/Glossary/SPA⟩.

Mozilla Developer Network. *JavaScript Guide: Numbers and dates*. 2023. ⟨https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide/Numbers_and_dates⟩.

Mozilla Developer Network. *let - JavaScript*. 2023. ⟨https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/let⟩.

Mozilla Developer Network. *var - JavaScript*. 2023. ⟨https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/var⟩.

MWAURA, W. *End-to-End Web Testing with Cypress: Explore techniques for automated frontend web testing with Cypress and JavaScript*. [S.l.]: Packt Publishing Ltd, 2021.

MYERS, G. J. et al. *The art of software testing*. [S.l.]: John Wiley & Sons, 2011.

MYSQL. *MySQL Documentation*. 2023. Acessado em 28-maio-2023, disponível em ⟨https://dev.mysql.com/doc/⟩.

NDUGU, A. Are html tags case sensitive? *DevPractical*, 2020. Disponível em: ⟨https://devpractical.com/are-html-tags-case-sensitive/⟩.

NECULA, S.-C. Semantic web applications: Current trends in datasets, tools and technologies' development for linked open data. *Informatica Economica*, v. 24, n. 4, 2020.

NETO, A. et al. Introdução a teste de software. *Engenharia de Software Magazine*, v. 1, p. 22, 2007.

NETWORK, M. D. *Basic JavaScript concepts*. 2023. ⟨https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#basics⟩. Acesso em 06/05/2023.

NETWORK, M. D. *JavaScript data types and data structures*. 2023. ⟨https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#dynamic_and_weak_typing⟩. Acesso em 06/05/2023.

NETWORK, M. D. *JavaScript data types and data structures*. 2023. ⟨https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures⟩. Acesso em 06/05/2023.

NETWORK, M. D. *Acessibilidade HTML — MDN*. 2024. Disponível em: ⟨https://developer.mozilla.org/pt-BR/docs/Learn_web_development/Core/Accessibility/HTML⟩.

NETWORK, M. D. *Atributos HTML — MDN*. 2024. Disponível em: ⟨https://developer.mozilla.org/pt-BR/docs/Web/HTML/Attributes⟩.

NETWORK, M. D. *CSS — MDN*. 2024. Disponível em: ⟨https://developer.mozilla.org/pt-BR/docs/Web/CSS⟩.

NETWORK, M. D. *HTML — MDN*. 2024. Disponível em: ⟨https://developer.mozilla.org/en-US/docs/Web/HTML⟩.

NEWMARCH, J. et al. Http. In: *Network Programming with Go Language: Essential Skills for Programming, Using and Securing Networks with Open Source Google Golang*. [S.l.]: Springer, 2022. p. 167–197.

NODE. *The V8 JavaScript Engine*. 2024. Acessado em 1 de maio de 2024: ⟨https://nodejs.org/en/learn/getting-started/the-v8-javascript-engine⟩.

NOSSOMEIO. *Linkedin: a rede social utilizada por 90% dos recrutadores.* 2022. Acesso em: 25 nov. 2023. Disponível em: ⟨https://nossomeio.com.br/linkedin-a-rede-social-utilizada-por-90-dos-recrutadores/⟩.

npm, Inc. *Downloading and installing Node.js and npm*. [S.l.], 2023. Acessado em 22 de abril de 2025. Disponível em: ⟨https://docs.npmjs.com/downloading-and-installing-node-js-and-npm⟩.

NPMJS. *About npm*. 2025. Acessado em 25 de fevereiro de 2025: ⟨https://docs.npmjs.com/about-npm⟩.

OLIVEIRA, J. et al. Evaluating a method to select software developers from source code analysis. Março 2023.

OLIVEIRA, M. *O que É um Servidor Api?* 2024. Disponível em: ⟨https://apidog.com/pt/blog/what-is-an-api-server/⟩.

OpenJS Foundation. *Download Node.js*. [S.l.], 2025. Acessado em 22 de abril de 2025. Disponível em: ⟨https://nodejs.org/en/download⟩.

O'REILLY. *Velocity 09: John Allspaw and Paul Hammond, "10+ Deploys Pe*. 2009. Vídeo online. Disponível em: ⟨https://www.youtube.com/watch?v=LdOe18KhtT4⟩.

PARKER, H. *Ferramentas de gerenciamento de requisitos*. 2024. Acesso em: 13 de set. 2024. Disponível em: ⟨https://clickup.com/pt-BR/blog/41671/ferramentas-de-gerenciamento-de-requisitos⟩.

PATEL, J. *10 Reasons Why Laravel is Better Than Other PHP Frameworks?* 2023. Acessado em 24-junho-2023, disponível em: ⟨https://www.monocubed.com/blog/why-laravel-php-framework/⟩.

PATEL, M. *Ruby on Rails Architecture — Everything You Need To Know*. 2023. Acessado em 07 de Abril de 2024: ⟨https://www.rorbits.com/ruby-on-rails-architecture/⟩.

PATHAK, J. et al. Modeling web services by iterative reformulation of functional and non-functional requirements. In: SPRINGER. *Service-Oriented Computing–ICSOC 2006: 4th International Conference, Chicago, IL, USA, December 4-7, 2006. Proceedings 4*. [S.l.], 2006. p. 314–326.

PATUCI, G. d. O. *Ferramentas de teste de software - Revista Engenharia de Software Magazine 37*. 2011. ⟨https://www.devmedia.com.br/ferramentas-de-teste-de-software-revista-engenharia-de-software-magazine-37/21424⟩.

PEARSON, M. et al. *Pro Microsoft power platform: solution building for the citizen developer*. [S.l.]: Springer, 2020.

PENA, L. et al. The contribution of linkedin use to career outcome expectations. *Journal of Business Research*, Elsevier, v. 144, p. 788–796, maio 2022.

PÉREZ, J. et al. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, ACM New York, NY, USA, v. 34, n. 3, p. 1–45, 2009.

PIKA. *Introduction to Pika*. 2017. Disponível em: ⟨https://pika.readthedocs.io/en/stable/⟩. Acesso em: 27 jun. 2023.

PILGRIM, M. *HTML5: up and running: dive into the future of web development*. [S.l.]: "O'Reilly Media, Inc.", 2010.

PINHEIRO, F. A. Requirements traceability. *Perspectives on software requirements*, Springer, p. 91–113, 2004.

PORTALPOS. *O que são hard skills: qual o significado e 10 exemplos*. 2022. Acesso em: 28 nov. 2023. Disponível em: ⟨https://blog.portalpos.com.br/o-que-sao-hard-skills/⟩.

POSTMAN. *Intro to apis: History of apis*. 2019. Disponível em: ⟨https://blog.postman.com/intro-to-apis-history-of-apis/⟩.

POSTMAN. *What are the components of an API?* 2023. Disponível em: ⟨https://blog.postman.com/what-are-the-components-of-an-api/⟩.

POSTMAN. *What is an API endpoint?* 2023. Disponível em: ⟨https://blog.postman.com/what-is-an-api-endpoint/⟩.

POSTMAN. 2024. Disponível em: ⟨https://www.postman.com/api-platform/api-client/⟩.

POTDAR, A. M. et al. Performance evaluation of docker container and virtual machine. *Procedia Computer Science*, v. 171, p. 1419–1428, 2020. ISSN 1877-0509. Third International Conference on Computing and Network Communications (CoCoNet'19). Disponível em: ⟨https://www.sciencedirect.com/science/article/pii/S1877050920311315⟩.

PRESSMAN, R. S. et al. *Web Engineering: A Practitioner's Approach*. [S.l.]: McGraw-Hill, 2009.

PRESSMAN, R. S. et al. *Engenharia de software-9*. [S.l.]: McGraw Hill Brasil, 2021.

PYTHON. *socket — Low-level networking interface*. 2023. Disponível em: ⟨https://docs.python.org/3/library/socket.html⟩. Acesso em: 26 jun. 2023.

QAMADNESS. *Skills QA Engineers Should Have Now and in the Future*. 2021. Acesso em: 28 nov. 2023. Disponível em: ⟨https://www.qamadness.com/skills-qa-engineers-should-have-now-and-in-the-future/⟩.

RABBITMQ. *RabbitMQ is the most widely deployed open source message broker*. 2023. Disponível em: ⟨https://www.rabbitmq.com/⟩. Acesso em: 27 jun. 2023.

RAILS, R. on. *Ruby on Rails*. 2024. Acessado em 29-fevereiro-2024, disponível em: ⟨https://rubyonrails.org/⟩.

RAMAKRISHNAN, R. et al. *Database Management Systems*. [S.l.]: McGraw-Hill Education, 2008. v. 3.

RAMOS, E. d. S. *Lidando com Stakeholders: boas práticas em Análise de Negócios e Engenharia de Requisitos*. Amazon.com: Kindle Direct Publishing (KDP) by Amazon.com, Inc., 2019. Edição do Kindle.

RAMOS, V. *DESENVOLVIMENTO WEB COM PYTHON E DJANGO: INTRODUÇÃO*. 2023. Acessado em 20 de Abril de 2023: ⟨https://pythonacademy.com.br/blog/desenvolvimento-web-com-python-e-django-introducao⟩.

RAMYA, P. et al. Testing using selenium web driver. *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, p. 1–7, 2017.

RAPID. 2021. Disponível em: ⟨https://rapidapi.com/blog/api-glossary/client/⟩.

RASHID, A. et al. Cloud computing characteristics and services: a brief review. *International Journal of Computer Sciences and Engineering*, v. 7, n. 2, p. 421–426, 2019.

REACT. *React Documentation*. 2025. Acessado em 19 de Março de 2025: ⟨https://react.dev/learn⟩.

REDHAT. *O que é API REST?* 2023. Acessado em 11 de Novembro de 2023: ⟨https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api⟩.

REDIS. *Getting started with Redis*. 2023. Disponível em: ⟨https://redis.io/docs/getting-started/⟩. Acesso em: 27 jun. 2023.

REDIS. *Redis Documentation*. 2023. Acessado em 28-maio-2023, disponível em ⟨https://redis.io/docs/⟩.

REDUX. *Getting Started with Redux*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://redux.js.org/introduction/getting-started⟩.

SILVA REGO, A. da. *Técnicas de Elicitação de Requisitos*. 2023. Accessed on September 13, 2024. Disponível em: ⟨https://www.estrategiaconcursos.com.br/blog/tecnicas-elicitacao-requisitos-ebserh/⟩.

REST, D. *Django REST*. 2024. Acessado em 26-agosto-2024, disponível em: ⟨https://www.django-rest-framework.org/⟩.

ŘEZNÍK, T. et al. Improving the documentation and findability of data services and repositories: A review of (meta) data management approaches. *Computers & Geosciences*, Elsevier, v. 169, p. 105194, 2022.

ROBBESTAD, S. A. *ReactJS blueprints*. [S.l.]: Packt Publishing, 2016.

ROBERTS, P. *What the heck is the event loop anyway?* 2014. Acessado em 11-junho-2023, disponível em ⟨https://2014.jsconf.eu/speakers/philip-roberts-what-the-heck-is-the-event-loop-anyway.html⟩.

ROUTER, R. *React Router*. 2023. Acessado em 26 de fevereiro de 2025: ⟨https://reactrouter.com/home⟩.

RUBY. *Ruby*. 2024. Acessado em 12-junho-2024, disponível em: ⟨https://www.ruby-lang.org/pt/⟩.

RUBY, S. et al. *Agile Web Development with Rails 7*. [S.l.]: Pragmatic Bookshelf, 2023.

SANTOS, A. Introdução ao desenvolvimento frontend: Html, css e javascript. *Código em Ação*, 2024. Disponível em: ⟨https://codigoemacao.com/introducao-ao-desenvolvimento-frontend-html-css-e-javascript/⟩.

SANTOS, A. B. *Which web browser should you be using 2023*. 2023. ⟨https://softwarekeep.com/blog/which-web-browser-should-you-be-using-in-2021-updated⟩.

SANTOS, L. N. *O teorema de CAP – Utilizado para auxiliar um servidor de sistema distribuídos*. 2017. Acessado em 25-abril-2023, disponível em ⟨https://medium.com/@lucasnsantos/o-teorema-de-cap-utilizado-para-auxiliar-um-servidor-de-sistema-distribu%C3%ADdos-f73d6a4c5d9⟩.

Schema.org. *Schema.org – Official Documentation*. 2024. ⟨https://schema.org⟩. Site oficial com descrição dos tipos, propriedades e guias de uso em JSON-LD, RDFa e Microdata. Acesso em: 31 jul. 2025.

SEGARAN, T. et al. *Programming the semantic web: Build flexible applications with graph data*. [S.l.]: "O'Reilly Media, Inc.", 2009.

SEGUNDO, J. E. S. Web semântica, dados ligados e dados abertos: uma visão dos desafios do brasil frente às iniciativas internacionais. *Tendências da Pesquisa Brasileira em Ciência da Informação*, v. 8, n. 2, 2015.

Selenium. *The selenium browser automation project*. 2023. ⟨https://www.selenium.dev/documentation/⟩.

SENDBIRD. *What is an API response?* Sendbird, 2025. Acesso em: 13 maio 2025. Disponível em: ⟨https://sendbird.com/learn/what-is-an-api-response⟩.

SERVICES, A. W. *Qual a diferença entre contêineres e máquinas virtuais?* 2023. ⟨https://aws.amazon.com/pt/compare/the-difference-between-containers-and-virtual-machines/⟩. Online; acessado em Jul 2023.

SHEPARD, A. *What are the different types of apis and protocols?* Kong Inc., 2023. Acesso em: 15 maio 2025. Disponível em: ⟨https://konghq.com/blog/learning-center/different-api-types-and-use-cases⟩.

SILVA, F. D. F. Uma nova abordagem de mineracaoo de repositÓrios de software utilizando ferramentas da web semÂntica. PUC-Rio, n. 4916, 2015.

SILVA, G. *Pessoa QA: o que faz e qual a sua importância*. 2023. Acesso em: 06 Dezembro. 2023. Disponível em: ⟨https://coodesh.com/blog/carreiras/pessoa-qa-o-que-faz-e-qual-a-sua-importancia/⟩.

SILVA, G. *Soft e hard skills de um desenvolvedor Front-end: conheça as mais valorizadas*. 2023. Acesso em: 06 Dezembro. 2023. Disponível em: ⟨https://coodesh.com/blog/carreiras/soft-e-hard-skills-de-um-front-end/⟩.

SOMMERVILLE, I. *Software Engineering*. 10th. ed. [S.l.]: Pearson, 2015. ISBN 0133943038.

SOMMERVILLE, I. *Engenharia de software*. [S.l.]: Editora Pearson Universidades, 2019.

SOMMERVILLE, I. *Engineering software products*. [S.l.]: Pearson London, UK, 2020. v. 355.

SOUSA, F. R. et al. Computação em nuvem: Conceitos, tecnologias, aplicações e desafios. 2009.

SOUZA, K. P. d. et al. A importância da atividade de teste no desenvolvimento de software. *XXXIII Encontro Nacional de Engenharia de Produção A Gestão dos Processos de Produção e as Parcerias Globais para o Desenvolvimento Sustentável dos Sistemas Produtivos [Internet].[citado em 2019 mar. 29]*, 2013.

SOUZA, R. R. et al. A web semântica e suas contribuições para a ciência da informação. *Ciência da Informação*, SciELO Brasil, v. 33, p. 132–141, 2004.

STACK, V. F. *Conhecendo o TypeScript*. 2021. ⟨https://vidafullstack.com.br/typescript/conhecendo-o-typescript/⟩. Online; Acessado em 5 de junho de 2026.

STĂNESCU, G. et al. Recent trends and insights in semantic web and ontology-driven knowledge representation across disciplines using topic modeling. *Electronics*, MDPI AG, v. 14, n. 7, p. 1313, 2025.

STAUFFER, M. *Laravel: Up & Running: A Framework for Building Modern PHP Apps*. O'Reilly Media, 2019. ISBN 9781492041184. Disponível em: ⟨https://books.google.com.br/books?id=HcqPDwAAQBAJ⟩.

STX Next. *What Is TypeScript? Pros and Cons of TypeScript vs. JavaScript*. 2022. ⟨https://www.stxnext.com/blog/typescript-pros-cons-javascript/⟩. Online; acessado em 5 de junho de 2026.

SUBIH, M. A. et al. Comparison of agile method and scrum method with software quality affecting factors. *International Journal of Advanced Computer Science and Applications*, Science and Information (SAI) Organization Limited, v. 10, n. 5, 2019.

SUKUMARAN, S. et al. Uma abordagem rigorosa para validação de requisitos. In: *Anais da 4ª Conferência Internacional de Engenharia de Software (SEFM 06)*. Washington, DC, USA: IEEE, 2006.

TAKY, M. T. Automated testing with cypress. 2021.

TAYE, M. M. Understanding semantic web and ontologies: Theory and applications. *arXiv preprint arXiv:1006.4567*, 2010.

The Git Development Team. *Git - Documentation*. 2024. Acessado em: 14 abr. 2025. Disponível em: ⟨https://git-scm.com/docs/git⟩.

THOMAS, D. et al. *The Pragmatic Programmer: your journey to mastery*. [S.l.]: Addison-Wesley Professional, 2019.

TIWARI, N. et al. A survey on selenium automation testing tool. *International Journal of Creative Research Thoughts (IJCRT)*, v. 9, 2021. ISSN 2320-2882.

TOOLKIT, R. *Comparison with Other Tools*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://redux-toolkit.js.org/rtk-query/comparison⟩.

TOOMIM, M. et al. Managing duplicated code with linked editing. In: IEEE. *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. [S.l.], 2004. p. 173–180.

TypeScript. *TypeScript*. 2023. ⟨https://www.typescriptlang.org/⟩. Online; acessado em 5 de junho de 2026.

UDDIN, M. S. et al. A comparative analysis of the rise of git-based distributed collaborative hosting platforms: Survey, performance test, and comparison. *Asian Journal of Engineering and Applied Technology*, v. 12, n. 2, p. 29–33, November 2023. Disponível em: ⟨https://ajeat.com/index.php/ajeat/article/view/3969⟩.

UFRN, I. M. D. *História dos bancos de dados*. 2023. Acessado em 05-maio-2023, disponível em ⟨https://materialpublic.imd.ufrn.br/curso/disciplina/4/56/1/3⟩.

V8. *Documentação V8 engine*. 2022. Acessado em 20-abril-2023, disponível em ⟨https://v8.dev/⟩.

VALENTE, M. T. Engenharia de software moderna. *Available on*, 2020.

VALENTE, M. T. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. [S.l.]: Editora: Independente, 2020.

VALENTE, M. T. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. [S.l.]: Editora: Independente, 2020.

VALES, Z. et al. Service api modeling and comparison: A technology-independent approach. In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. [S.l.: s.n.], 2020. p. 158–161.

VAZQUEZ, C. E. et al. *Engenharia de Requisitos: software orientado ao negócio*. [S.l.]: BRASPORT, 2020. 92 p. Edição do Kindle.

SOUSA VELOSO, J. de et al. Avaliação de ferramentas de apoio ao teste de sistemas de informação. *iSys-Brazilian Journal of Information Systems*, v. 3, n. 1, 2010.

VERMA, A. Mvc architecture: A comparative study between ruby on rails and laravel. *Indian Journal of Computer Science and Engineering (IJCSE)*, v. 5, n. 5, p. 196–198, 2014.

Visure Solutions. *O Guia Mais Completo para Gerenciamento de Requisitos e Rastreabilidade*. 2024. Acesso em: 13 de set. 2024. Disponível em: ⟨https://visuresolutions.com/pt/requirements-management-traceability-guide/requirements-gathering/⟩.

VISWANATHAN, V. Rapid web application development: a ruby on rails tutorial. *IEEE software*, IEEE, v. 25, n. 6, p. 98–106, 2008.

VITE. *Vite Documentation*. 2023. Acessado em 6 de Junho de 2023: ⟨https://vitejs.dev/⟩.

VOLPATO, B. *Ranking: as redes sociais mais usadas no Brasil e no mundo em 2023, com insights, ferramentas e materiais*. 2023. Acessado em 25-maio-2023, disponível em ⟨https://resultadosdigitais.com.br/marketing/redes-sociais-mais-usadas-no-brasil/⟩.

W3C. *Shapes Constraint Language (SHACL)*: W3c recommendation. 2017. Disponível em: ⟨https://www.w3.org/TR/shacl/#introduction⟩.

W3SCHOOLS. *JavaScript History*. 2023. ⟨https://www.w3schools.com/js/js_history.asp⟩. Acesso em 06/05/2023.

W3SCHOOLS. *JavaScript Use Strict*. 2023. ⟨https://www.w3schools.com/js/js_strict.asp⟩.

W3SCHOOLS. *w3schools react hooks*. 2023. Acessado em 8 de setembro de 2023: ⟨https://www.w3schools.com/react/react_hooks.asp⟩.

W3SCHOOLS. *CSS Tutorial*. 2024. Disponível em: ⟨https://www.w3schools.com/css/css_intro.asp⟩.

W3SCHOOLS. *How TO - Make a Static Website*. 2024. Acessado em 1 de maio de 2024: ⟨https://www.w3schools.com/howto/howto_website_static.asp⟩.

W3SCHOOLS. *CSS Tutorial*. 2025. Acessado em 25 de fevereiro de 2025: ⟨https://www.w3schools.com/css/⟩.

W3SCHOOLS. *HTML Tutorial*. 2025. Acessado em 25 de fevereiro de 2025: ⟨https://www.w3schools.com/html/⟩.

W3SCHOOLS. *JavaScript Modules*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://www.w3schools.com/js/js_modules.asp⟩.

W3SCHOOLS. *JavaScript Tutorial*. 2025. Acessado em 25 de fevereiro de 2025: ⟨https://www.w3schools.com/js/⟩.

W3SCHOOLS. *React Components*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://www.w3schools.com/react/react_components.asp⟩.

W3SCHOOLS. *React Conditional Rendering*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://www.w3schools.com/react/react_conditional_rendering.asp⟩.

W3SCHOOLS. *React JSX*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://www.w3schools.com/react/react_jsx.asp⟩.

W3SCHOOLS. *Styling React Using CSS*. 2025. Acessado em 26 de fevereiro de 2025: ⟨https://www.w3schools.com/react/react_css.asp⟩.

WARDHAN, H. et al. Study on functioning of selenium testing tool. *International Research Journal of Modernization in Engineering Technology and Science Www. Irjmets. Com@ International Research Journal of Modernization in Engineering*, p. 2582–5208, 2021.

WAZLAWICK, R. S. Análise e projeto de sistemas de informação orientados a objetos. Elsevier Editora, 2011.

WGU. *5 most popular operating systems*. 2021. ⟨https://www.wgu.edu/blog/5-most-popular-operating-systems1910.html⟩.

Wikipédia. *Tipo de dado*. 2022. ⟨https://pt.wikipedia.org/wiki/Tipo_de_dado⟩. Online; acessado em 5 de junho de 2026.

Wikipédia. *Variável (programação)*. 2022. ⟨https://pt.wikipedia.org/wiki/Vari%C3%A1vel_(programa%C3%A7%C3%A3o)⟩. Online; acessado em 5 de junho de 2026.

WIKIPéDIA. *Modelo Cliente-Servidor*. 2023. Acessado em 07-setembro-2023, disponível em ⟨https://pt.wikipedia.org/wiki/Modelo_cliente%E2%80%93servidor⟩.

Wikipédia. *Primitive data type*. 2023. ⟨https://en.wikipedia.org/wiki/Primitive_data_type⟩. Online; acessado em 5 de junho de 2026.

WSO2. *API Manager Documentation -- Authorization Code Grant*. 2023. Disponível em: ⟨https://apim.docs.wso2.com/en/latest/design/api-security/oauth2/grant-types/authorization-code-grant⟩. Acesso em: 25 mai. 2023.

WSO2. *Identity Server Documentation -- Architecture*. 2023. Disponível em: ⟨https://is.docs.wso2.com/en/latest/get-started/architecture/#service-provider-section⟩. Acesso em: 25 mai. 2023.

WSO2. *Identity Server Documentation -- OpenID Connect Scopes and Claims*. 2023. Disponível em: ⟨https://is.docs.wso2.com/en/latest/references/concepts/authentication/scopes-claims⟩. Acesso em: 28 mai. 2023.

WSO2. *Identity Server Documentation -- Refresh Token Grant*. 2023. Disponível em: ⟨https://is.docs.wso2.com/en/latest/learn/refresh-token-grant⟩. Acesso em: 28 mai. 2023.

WSO2. *Identity Server Documentation -- Single Sign-On*. 2023. Disponível em: ⟨https://is.docs.wso2.com/en/latest/references/concepts/single-sign-on⟩. Acesso em: 25 mai. 2023.

WU, X. Fastapi as a backend framework. 2024.

ZOLKIFLI, N. N. et al. Version control system: A review. *Procedia Computer Science*, Elsevier, v. 135, p. 408–415, 2018.

ISBN 978-854341982-4

editora
Virtual Books
www.virtualbooks.com.br