# INFRAESTRUTURA PARA DESENVOLVIMENTO WEB

## UNIDADE I

### INTRODUÇÃO À INFRAESTRUTURA

## Elaboração

Miriã Falcão Freitas

## Produção

Equipe Técnica de Avaliação, Revisão Linguística e Editoração

# SUMÁRIO

# UNIDADE I

# INTRODUÇÃO À INFRAESTRUTURA

**Carta de um desenvolvedor desoftware:**

> Durante muito tempo, fui um desenvolvedor apaixonado por desenvolvimento *web* e código-fonte. Normalmente eu faria toda a integração do sistema e implementaria o aplicativo. Era emocionante e eu estava sendo bem pago por isso. No entanto, essa rotina envolvia escrever código que não estava em conformidade com os padrões de desenvolvimento *web*. Mas, desde que o aplicativo funcionasse tudo estava bom.
>
> Minhas rotinas como desenvolvedor continuaram até que um dia encontrei com colegas em um projeto conjunto com diferentes origens. Depois de um pequeno empurrão com prós e contras, decidimos (a equipe e eu) por uma estrutura e iniciamos o projeto. Um dos nossos maiores desafios era a profundidade do nosso trabalho. Consultei o engenheiro de *software* da equipe e ele sugeriu que separássemos as visualizações (*front-end*) da lógica (*back-end*). Então, pela primeira vez, e após muito estudar, escrevi minha primeira API. A equipe acompanhou o desenvolvimento e passou a consumir a API que criei. A integração foi tranquila. Depois disso, trabalhamos em paralelo e isso levou a boas práticas de desenvolvimento (FOWLER, 2014, p. 62).

Desenvolver um sistema *web* com separação em *front-end* e *back-end*, **é** muito melhor do que colocar tudo em um aplicativo monolítico gigante. Além do mais, há diversas vantagens nessa prática:

- O desenvolvedor *back-end* não precisa se preocupar em ter entidades de *front-end* em seu código.
- A separação impede que a infraestrutura de *back-end* prejudique a fluidez do *front-end*, por exemplo, Ajax e transições suaves.
- O fluxo e a visualização que os clientes estão acostumados não precisam mudar, caso o *back-end* seja reconstruído.
- Essa adoção introduz o paralelismo no ciclo de desenvolvimento. Assim, os desenvolvedores de *back-end* e *front-end* podem começar a trabalhar sem ter que esperar pela implementação do outro.
- O gerenciamento da base do código-fonte torna-se muito melhor de cada lado.

Nesta Unidade você aprender**á** isso e muito mais sobre o desenvolvimento *web*. Está pronto?

# CAPÍTULO 1

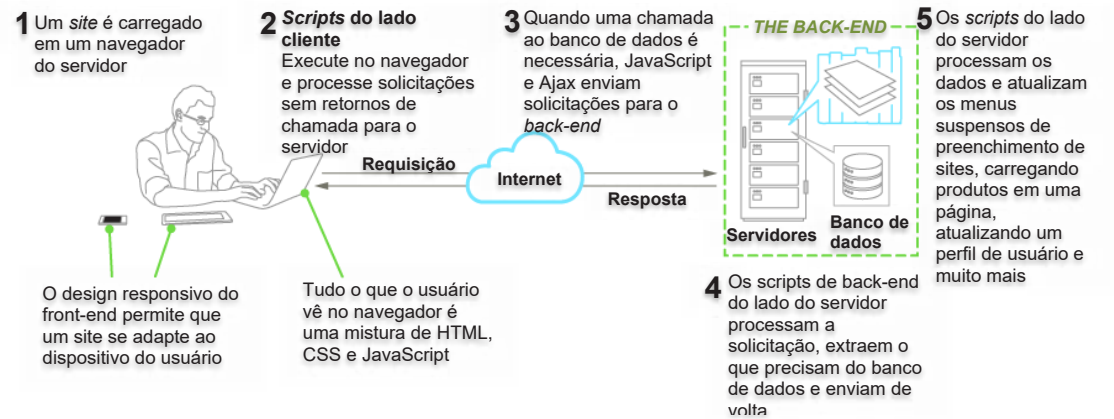
## FRONT-END

O desenvolvimento *web front-end*, também conhecido como desenvolvimento do lado do cliente, é a prática de produzir HTML, CSS e JavaScript para um *site* ou aplicativo *web*, para que um usuário possa ver e interagir diretamente com eles. O desafio associado ao desenvolvimento do *front-end* é que as ferramentas e técnicas usadas para criar o *front-end* de um *site* mudam constantemente e, portanto, o desenvolvedor precisa estar constantemente ciente de como o campo está se desenvolvendo.

O objetivo de criar um *site* é garantir que, quando os usuários o abram, vejam as informações em um formato fácil de ler e relevante. Isso é ainda mais complicado pelo fato de os usuários agora usarem uma grande variedade de dispositivos com tamanhos e resoluções de tela variados, forçando o *designer* a levar em consideração esses aspectos ao projetar o *site*. Eles precisam garantir que o *site* seja exibido corretamente em diferentes navegadores (entre navegadores), diferentes sistemas operacionais (entre plataformas) e diferentes dispositivos (entre dispositivos), o que requer um planejamento cuidadoso do lado do desenvolvedor.

Um desenvolvedor *front-end* projeta e desenvolve *sites* e aplicativos usando tecnologias da *web* (HTML, CSS, DOM e JavaScript), que são executadas na *Open Web Platform* ou atuam como entrada de compilação para ambientes de plataforma não *web* (como *React Native*). A Figura 1 ilustra o trabalho de um desenvolvedor *front-end*.

Figura 1. Desenvolvedor *Front-End*




Fonte: Adaptado de Wodehouse , (2015).

***Open Web Platform*** é a coleção de tecnologias abertas (livres de *royalties*) que permitem a *web*. Usando a *Open Web Platform*, todos têm o direito de implementar um componente de *software* da *web* sem exigir aprovações ou renúncia a taxas de licença.

> ***React Native*** é um *framework* baseado no já aclamado *React* (biblioteca JavaScript para criar interfaces de usuário), desenvolvido pela equipe do *Facebook*, que possibilita o desenvolvimento de aplicações mobile, tanto para Android, como para iOS, utilizando apenas JavaScript.

O desenvolvedor *front-end* cria interação e experiência do usuário com *scripts* incorporados no HTML de um *site*. Tudo o que um visitante do seu *site* vê, clica ou usa para inserir ou recuperar informações é o trabalho do desenvolvedor *front-end* que cria um *software* do lado do cliente que dá vida ao *design* do *site*. Os *scripts* são baixados pelo navegador, processados e depois executados separadamente do servidor.

Isso requer partes iguais de tecnologia e visão. Os desenvolvedores de *front-end* são a ponte entre o *designer* e o programador de *back-end*, o que significa que eles precisam ser criativos e conhecedores de tecnologia.

Tudo no *front-end* é construído com uma mistura de *scripts* HTML, CSS e do lado do cliente, como JavaScript — os elementos principais do desenvolvimento do *front-end*. O desenvolvedor *front-end* une o mundo do design e da tecnologia, dando vida ao *design* e empacotando a utilidade do *back-end* de maneira moderna e convidativa para os usuários interagirem.

## 1.1. Tecnologias empregadas no *Front-End*

As seguintes tecnologias são empregadas por desenvolvedores *front-end*:

- Linguagem de marcação de hipertexto (HTML).
- Folhas de estilos em cascata (CSS).
- Localizadores uniformes de recursos (URLs).
- Protocolo de transferência de hipertexto (HTTP).
- Linguagem de programação JavaScript.
- Notação de Objeto JavaScript (JSON).
- Modelo de Objeto de documento (DOM).
- APIs da *Web* (HTML5 ou APIs do navegador).
- Diretrizes de acessibilidade para conteúdo (WCAG) e aplicativos *Accessible Rich Internet* (ARIA).

A Figura 2 apresenta as três principais tecnologias da *web* empregadas por desenvolvedores *front-end*.

Figura 2. Principais tecnologias *Front-End*


Fonte: Adaptado de Lindley (2018).

### 1.1.1. HTML

A *HyperText Markup Language* (HTML) é a linguagem de marcação padrão usada para criar páginas da *web*. Os navegadores da *web* podem ler arquivos HTML e transformá-los em páginas da *web* visíveis ou audíveis. O HTML descreve a estrutura de um site semântica, juntamente com sugestões para apresentação, tornando-o uma linguagem de marcação, em vez de uma linguagem de programação.

### 1.1.2. CSS

*Cascading Style Sheets* (CSS) é uma linguagem de folha de estilos usada para descrever a aparência e a formatação de um documento escrito em uma linguagem de marcação. Embora seja usado com mais frequência para alterar o estilo de páginas da *web* e interfaces de usuário escritas em HTML e XHTML, a linguagem pode ser aplicada a qualquer tipo de documento XML, incluindo XML simples, SVG e XUL. Juntamente com HTML e JavaScript, o CSS é uma tecnologia fundamental usada pela maioria dos sites para criar páginas da *web* visualmente atraentes, interfaces de usuário para aplicativos da *web* e interfaces de usuário para muitos aplicativos móveis.

### 1.1.3. HTTP

O *Hypertext Transfer Protocol* (HTTP) é um protocolo de aplicativo para sistemas de informações hipermídia distribuídos, colaborativos. O HTTP é a base da comunicação de dados para a *world wide web*.

### 1.1.4. URL

Um localizador uniforme de recursos (URL) (também chamado de endereço da *web*) é uma referência a um recurso que especifica o local do recurso em uma rede de computadores e um mecanismo para recuperá-lo. Um URL é um tipo específico de identificador uniforme de recursos (URI), embora muitas pessoas usem os dois termos de forma intercambiável. Uma URL implica os meios para acessar um recurso indicado, o que não é verdadeiro para todos os URI. Os URLs ocorrem com mais frequência para referenciar páginas da web (HTTP), mas, também, são usados para transferência de arquivos (FTP), *e-mail*, acesso ao banco de dados (JDBC) e muitos outros aplicativos.

### 1.1.5. DOM

O *Document Object Model* (DOM) é uma convenção de plataforma cruzada e independente de idioma para representar e interagir com objetos em documentos HTML, XHTML e XML. Os nós de cada documento são organizados em uma estrutura em árvore, chamada de árvore DOM. Objetos na árvore DOM podem ser endereçados e manipulados usando métodos nos objetos. A interface pública de um DOM está especificada em sua interface de programação de aplicativos (API).

### 1.1.6. JavaScript

JavaScript é uma linguagem de programação de alto nível, dinâmica, sem tipo e interpretada. Foi padronizado na especificação da linguagem ECMAScript. Juntamente com HTML e CSS, é uma das três tecnologias essenciais da produção de conteúdo da *world wide web*; a maioria dos *sites* o utiliza e é suportado por todos os navegadores modernos sem *plug-ins*. O JavaScript é baseado em protótipo com funções de primeira classe, tornando-o uma linguagem de vários paradigmas, suportando estilos de programação orientados a objetos, imperativos e funcionais. Tem uma API para trabalhar com texto, matrizes, datas e expressões regulares, mas não inclui qualquer I/O, tais como redes, instalações de armazenamento ou gráficos, contando, para isso, no ambiente de acolhimento em que ela está inserida.

### 1.1.7. HTML5

HTML5 é uma linguagem de programação cujo acrônimo significa *Hyper Text Markup Language*. É um sistema que permite modificar a aparência das páginas da *web*, além de fazer ajustes na aparência. Também era usado para estruturar e apresentar conteúdo para a *web*.

Com o HTML5, navegadores como Firefox, Chrome, Explorer, Safari e outros podem saber como exibir uma página da *web* específica, saber onde estão os elementos, onde colocar as imagens e onde colocar o texto.

Além do HTML5, existem outros idiomas necessários para fornecer formato e interatividade a um *site*, mas a estrutura básica de qualquer página é definida pela primeira vez na linguagem HTML5.

As principais virtudes do HTML5 sobre seu antecessor (HTML4) é que você pode adicionar conteúdo multimídia sem usar o *Flash* ou outro *media player*. Graças ao HTML5, os usuários podem acessar *sites* sem estarem conectados à internet. Além disso, há a funcionalidade de arrastar e soltar, bem como a edição *on-line* de documentos que foram popularizados pelo Google Docs.

### 1.1.8. JSON

JSON: **J**ava **S**cript **O**bject **N**otation.

JSON é uma sintaxe para armazenar e trocar dados.

JSON é um texto, escrito com notação de objeto JavaScript.

Ao trocar dados entre um navegador e um servidor, os dados podem ser apenas texto. JSON é texto, e podemos converter qualquer objeto JavaScript em JSON e enviar JSON para o servidor. Também podemos converter qualquer JSON recebido do servidor em objetos JavaScript. Dessa forma, podemos trabalhar com os dados como objetos JavaScript, sem análises e traduções complicadas.

### 1.1.9. WCAG

As Diretrizes de Acessibilidade para Conteúdo Web (WCAG) abrangem diversas recomendações com a finalidade de tornar o conteúdo da *web* mais acessível. Seguir estas diretrizes irá tornar o conteúdo acessível a um maior número de pessoas com deficiência, incluindo acomodações para cegueira e baixa visão, surdez e baixa audição, limitações de movimentos, incapacidade de fala, fotossensibilidade e combinações destas características, e alguma acomodação para dificuldades de aprendizagem e limitações cognitivas; mas não abordará todas as necessidades de usuários com essas deficiências. Seu conteúdo da *web* também ficará mais acessível aos usuários em geral ao seguir estas diretrizes.

### 1.1.10. ARIA

Aplicações Ricas para tornar a Internet Acessível — *Accessible Rich Internet Applications* (ARIA) definem as formas de tornar o conteúdo e as aplicações da Rede Mundial — *Web* — (especialmente aqueles desenvolvidos com Ajax e *JavaScript*) mais acessíveis às pessoas com deficiência. Por exemplo, a ARIA permite a marcação de regiões importantes na página (como uma caixa de busca, um cabeçalho, chamadas "pontos de referência"), para facilitar a navegação (agilizam a utilização de leitores de tela, por exemplo), JavaScript para *widgets*, sugestões de preenchimento de formulário e mensagens de erro, atualizações de conteúdo em tempo real e muito mais.

**HTML**. Disponível em: https://html.spec.whatwg.org/multipage/introduction.html#introduction

**CSS**. Disponível em: https://developer.mozilla.org/en-US/docs/Web/CSS/Reference

**HTTP**. Disponível em: https://httpwg.org/specs/rfc7540.html

**URL**. Disponível em: https://url.spec.whatwg.org/

**DOM**. Disponível em: https://dom.spec.whatwg.org/

**JavaScript**. Disponível em: http://ecma-international.org/ecma-262/9.0/index.html#Title

**HTML5**. Disponível em: https://www.devmedia.com.br/o-que-e-o-html5/25820

**JSON**. Disponível em: https://www.json.org/json-en.html

**WCAG**. Disponível em: https://ceweb.br/cartilhas/cartilha-w3cbr-acessibilidade-web-fasciculo-I.html

**ARIA.** Disponível em: https://tableless.com.br/documentos-acessiveis-com-aria-em-html5/

## 1.2. Infraestrutura *front-end*

Muitos desenvolvedores são hábeis em codificação, mas não se aprofundam na construção e implantação de aplicativos. Entender a infraestrutura do *front-end* é fundamental para que o desenvolvedor saiba como definir o processo de desenvolvimento como um todo.

Podemos dividir a infraestrutura *front-end* em quatro áreas principais:

» Desenvolvimento de aplicações.

» Desenvolvimento local.

» Implantação.

» Monitoramento e rastreamento de eventos.

### 1.2.1. Desenvolvimento de aplicações

O desenvolvimento de aplicações é dividido em cinco áreas, são elas:

» Componentes.

» Guia de estilo de componentes.

» Componente de versão.

» Agrupamento.

» Configurações compartilhadas.

#### 1.2.2. Componentes

São os componentes de todos os aplicativos *front-end*. Os componentes incentivam a reutilização e composição, impedindo que os desenvolvedores reinventem a roda.

Mesmo se você planeja usar uma estrutura de código aberto, ainda assim, deve agrupar os componentes. Definir sua própria API permite alterar os detalhes da implementação sem violar os contratos.

#### 1.2.3. Guia de estilo de componentes

Após adquirir sua biblioteca de componentes, hospede-a para que as equipes possam observar e aprovar alterações. Para isso, você pode usar um guia de estilo como o *Storybook*.

O *Storybook* é uma ferramenta de código aberto para o desenvolvimento de componentes de interface do usuário isolados para *React*, *Vue* e Angular. Torna a criação de UIs organizada e eficiente.

Para mais informações consulte: https://storybook.js.org/ (STORYBOOK, 2020).

Um guia de estilo é uma boa maneira de formalizar um contrato entre Produto, *Design* e Engenharia. Ele permite que os membros da equipe detectem problemas de implementação antes de entrar em desenvolvimento ativo.

### 1.2.4. Componente de versão

Se você for usar seus componentes em mais de um projeto ou dispositivo, os componentes de controle de versão são obrigatórios.

No início, o controle de versão parece que o deixará mais lento. Ter atualizações de botão de pressão (monólito) para uma plataforma inteira parece mais rápido. Realmente é um pesadelo de controle de qualidade. Também exige que todas as equipes estejam prontas para atualizar de uma só vez, o que pode ser desafiador.

O controle de versão tem muitos desafios. Você precisará resolver "quem pode" e "quando" publicar. Para isso, considere o uso de **confirmações convencionais** e configure o servidor para captar as alterações.

Saiba mais sobre **Confirmações convencionais**. Disponível em: https://www.conventionalcommits.org/en/v1.0.0-beta.2/ (CONVENTIONAL COMMITS, 2020)

### 1.2.5. Agrupamento

Agrupar é o processo em que seu código-fonte se transforma em um aplicativo que você pode distribuir para seus usuários. Ele resolve muitos problemas: permitindo que diferentes tipos de módulos conversem entre si, compilação de módulos CSS e muito mais. Muitos desses processos podem ser tratados isoladamente, mas os empacotadores facilitam a execução de todas essas tarefas em conjunto.

Os empacotadores não são a única maneira de realizar algumas tarefas. Algumas pessoas preferem usar soluções orientadas a tarefas como Gulp ou Grunt. Mesmo que sejam vistos como "antiquados", sua simplicidade não deve ser negada.

De várias maneiras, o agrupamento pode ser a desgraça do desenvolvimento do *front-end,* caso fique muito complexo e provavelmente poderíamos ter que dar um passo atrás e fazer uma reavaliação.

Os empacotadores resolvem muitos problemas que você pode evitar. Deseja evitar referenciar manualmente os arquivos que precisam de concatenação? Você precisará de um empacotador. Deseja usar a substituição de módulo quente? Você precisará de um empacotador. Deseja comunicar-se perfeitamente entre os módulos CJS e ES6? Você definitivamente precisa de um empacotador!

**Substituição do módulo quente** ou *Hot Module Replacement* viabiliza a troca, adição ou remoção de módulos enquanto um aplicativo está em execução, sem uma recarga completa. Isso pode acelerar significativamente o desenvolvimento de algumas maneiras:

- Retém o estado do aplicativo perdido durante uma recarga completa.
- Economiza um tempo valioso de desenvolvimento atualizando apenas o que mudou.
- Atualiza instantaneamente o navegador quando forem feitas modificações em CSS / JS no código-fonte, que é quase comparável à mudança de estilos diretamente nas ferramentas de desenvolvimento do navegador.

#### 1.2.6. Configurações compartilhadas

A experiência do desenvolvedor é a chave para escrever um bom *software*. Fornecer uma experiência consistente ao alternar projetos ajuda a aliviar a carga cognitiva da alternância de contexto.

Compartilhar configurações entre projetos pode ajudar a fornecer consistência. As configurações compartilhadas respondem a perguntas como “Em qual versão do *JavaScript* escrevemos”, “Com quais erros de conexão nos importamos” ou “Em quais versões de navegador estamos direcionados”.

As configurações compartilhadas também são uma boa maneira de otimizar as correções para gargalos de desenvolvimento. Eles eliminam a redundância de pessoas que resolvem os mesmos problemas de construção em diferentes projetos.

### 1.2.7. local

O desenvolvimento local é dividido em três áreas, são elas:

- Configuração de ambiente.
- Servidor *web*.
- Recarregamento de módulos quentes.

#### 1.2.8. Configuração de ambiente

Seu primeiro objetivo deve ser criar um processo de desenvolvimento local que exija o mínimo de instalação possível. Toda a configuração deve levar de 10 a 15 minutos.

### 1.2.9. Servidor *web*

A inicialização do servidor *web* deve ser um *script NPM* de uma palavra: *yarn start* ou *npm start*. O *script* em si pode ter uma cadeia de comandos como: *yarn webpack-dev-server && yarn,* etc.

Saiba mais sobre **Scripts NPM.** Disponível em: https://www.freecodecamp.org/news/introduction-to-npm-scripts-1dbb2ae01633/.

Os aplicativos podem ser servidos por algo tão simples quanto o *webpack-dev-server*. No entanto, a menos que você esteja escrevendo uma biblioteca de interface do usuário sem estado, precisará de alguns dados. Para isso, você precisará de algum tipo de proxy. A regra de ouro é que atingir esse *proxy* deve ser contínuo. Os desenvolvedores não precisam editar arquivos de configuração para fazê-lo funcionar.

O que você absolutamente deseja evitar é forçar os desenvolvedores a criar ambientes que eles não possuem. Isso só causará dores de cabeça à sua equipe *Front-End* e aumentará a aceleração.

### 1.2.10. Recarregamento de módulo quente

Além de a configuração e a inicialização serem rápidas, você deseja que o desenvolvimento real seja rápido também. É aqui que entra o *Hot Module Reloading*.

Recarga de módulo quente permite recarregar partes do aplicativo sem precisar atualizar a página. Em algum momento, é necessário um esforço extra para configurar, dependendo da complexidade do seu aplicativo. Esse esforço extra renderá dividendos, aumentando o prazer e a velocidade geral do desenvolvimento.

## 1.2.11. Implantação

A implantação é dividida em duas áreas, são elas:

» Integração contínua.

» Remessa.

### 1.2.12. Integração contínua

Depois que um desenvolvedor conclui a codificação, ele precisa levar seu código aos usuários. Não é tão simples fazer o *upload* de tudo para um *bucket* S3. Você precisará

criar e verificar suas alterações antes da implantação. É aí que entra um servidor de integração contínua (CI).

Os servidores de CI são de todos os tipos. Alguns exigem muita configuração de desenvolvedor como o Jenkins. Alguns serão mais manuais, como Travis CI ou Codeship.

Independentemente do que você decidir usar, você precisará saber como eles funcionam e como definir planos para os desenvolvedores. Para o *Front-End*, isso geralmente requer código de agregação, execução de testes de unidade e integração, localização de ativos, implantação na preparação e promoção na produção.

### 1.2.13. Remessa

Em seguida, você precisa colocar seu aplicativo nas mãos dos usuários. Existem inúmeras maneiras de disponibilizar seu aplicativo, mas uma configuração típica será mais ou menos assim:

- *Dockerize* seu aplicativo.
- Hospedado no AWS ECS, geralmente executando em uma instância do EC2.
- Nginx ou um AWS *Elastic Load Balancer*.
- CDN para servir ativos estáticos.

***Dockerize*** é um utilitário para simplificar aplicativos em execução em *containers* de encaixe. O caso típico de uso para *dockerize* é quando você possui um aplicativo com um ou mais arquivos de configuração e deseja controlar alguns dos valores usando variáveis de ambiente.

Infelizmente, é aqui que muitos desenvolvedores do *front-end* desistem e chutam tudo por cima dos muros para *devops*. Se você está fazendo isso, está fazendo errado.

Mesmo que o *Front-end* não seja o proprietário direto da infraestrutura da *web*, a equipe deve entendê-la intimamente. Eles devem garantir que os ativos tenham os cabeçalhos corretos ou que o conteúdo seja compactado com gzip.

Eles devem prestar atenção ao tamanho do pacote e medir seu impacto no usuário final. Você deve alinhar sua estratégia de armazenamento em cache do pacote para que a maneira como seus ativos sejam implantados tenha o menor impacto possível sobre como eles são baixados pelo usuário final.

### 1.2.14. Monitoramento e rastreamento de eventos

O monitoramento e rastreamento de eventos são divididos em duas áreas, são elas:

» Monitoramento de erros.

» Principais indicadores de desempenho.

#### 1.2.15. Monitoramento de erros

Quando seu aplicativo estiver disponível para milhões de pessoas, você quer garantir que nada dê errado. É por isso que você precisará de monitoramento para saber o que está acontecendo. Existem muitas soluções e, muitas delas, como *Sentry*, *Rollbar* ou *TrackJS*, exigem pouca configuração.

#### 1.2.16. Principais indicadores de desempenho

O monitoramento de erros captura apenas o negativo e o positivo? Você deseja mostrar todas as tendências sofisticadas que demonstram como o aplicativo está sendo usado. Resolve problemas do mundo real?

Para isso, você precisará de algo que possa triturar e agregar eventos em milhões de registros. Algumas ferramentas a serem consideradas são o *New Relic Insights* ou até o *Google Analytics*.

# CAPÍTULO 2

## BACK-END

O desenvolvimento *web back-end*, também conhecido como desenvolvimento do lado do servidor. O *back-end* é o código executado no servidor, que recebe solicitações dos clientes e contém a lógica para enviar os dados apropriados de volta ao cliente. O *back-end* também inclui o banco de dados, que armazenará persistentemente todos os dados do aplicativo.

E quem são os clientes que enviam solicitações para o *back-end*? Geralmente, são navegadores que solicitam o código HTML e JavaScript que serão executados para exibir *sites* para o usuário final. No entanto, existem muitos tipos diferentes de clientes: eles podem ser um aplicativo móvel, um aplicativo em execução em outro servidor ou mesmo um dispositivo inteligente habilitado para a *web*.

Logo, o desenvolvedor *back-end* é responsável pela programação das ações que o *site* executa no servidor. O trabalho de um desenvolvedor *back-end* gira em torno de:

- **Autenticação de usuário**: garantir que os detalhes da conta do usuário estejam corretos, que ele tenha a permissão necessária para ver o conteúdo do *site*/sistema *web*.
- **Manuseio de pedidos**: garantir que não haja erros nos pedidos processados no *site*.
- **Otimização**: garantir que todas as funcionalidades do *site* funcionem não apenas da maneira mais rápida possível.
- **Automatizar tarefas**: responsável por escrever o código-fonte que automatiza tarefas que são executadas repetidamente.
- **Proteger os dados**: responsável por criar tarefas que garantem que os dados inseridos no sistema/*site* sejam válidos antes de fazer ajustes no servidor.
- **Acessar o banco de dados**: otimizar o processo de acesso aos bancos de dados de forma que o *site* carregue o mais rápido possível e que sua funcionalidade seja executada o mais rápido possível.
- **Criar APIs**: responsável por trabalhar e criar APIs necessárias à economia de tempo no desenvolvimento do sistema/site.

## 2.1. Desenvolvimento do *back-end*

A programação de *back-end* pode ser orientada a objeto (OO) ou funcional.

A primeira é a técnica que se concentra na criação de objetos. Com a programação orientada a objetos, as instruções devem ser executadas em uma ordem específica. As linguagens mais populares de Programação Orientada a Objetos (POO) são Java, .NET e Python.

Esta última é uma técnica mais baseada em "ação". A programação funcional usa linguagem declarativa, o que significa que as instruções podem ser executadas em qualquer ordem. É comumente usada para ciência de dados, e as linguagens mais populares são SQL, F# e R.

Vamos ver mais detalhadamente algumas linguagens de desenvolvimento de *back-end* e para que elas são usadas.

### 2.1.1. Java

Java é a linguagem de programação mais popular do mundo — e por boas razões. O Java não é apenas extremamente versátil (seu uso se estende de *smartphones* a cartões inteligentes); também tem sido usado por desenvolvedores há mais de 20 anos.

O que torna o Java tão versátil é a *Java Virtual Machine* (JVM). Com muitas linguagens de desenvolvimento, a compilação de um programa cria um código que pode ser executado de maneira diferente se os computadores em que são executados forem diferentes. Este não é um problema para Java devido à JVM. A *Java Virtual Machine* atua como uma camada intermediária que pode executar código em qualquer computador, independentemente de onde o código foi compilado.

Embora o Java seja extremamente popular entre os desenvolvedores de *software* de *desktop* e de negócios, é menos amigável para iniciantes do que uma linguagem como *Python*. Isso ocorre porque é detalhado e requer mais código para criar recursos; como resultado, é menos gratificante para quem está começando.

Dito isto, a longa popularidade e o uso variado de Java significam que sua comunidade é grande. Consequentemente, qualquer dúvida que você possa ter sobre o idioma provavelmente terá tópicos em fóruns e tutoriais *on-line* dedicados a respondê-lo.

O que pode ser feito com Java?

» Desenvolvimento de aplicativos móveis.

- Desenvolvimento de *sites*.
- Conectividade de bancos de dados.
- Processamento de imagens.
- Programas baseados em GUI, redes e muito mais.

### 2.1.2. PHP

O PHP alimenta 78,9% de todos os *sites* (W3TECHS, 2020) cuja linguagem de programação do servidor conhecemos. O idioma foi lançado pela primeira vez em 1995, quando havia poucas opções para a criação de *sites* dinâmicos.

Como o PHP é digitado dinamicamente, significa que você pode criar uma variedade de soluções alternativas para um problema. Isso também significa que o mesmo trecho de código pode significar algo diferente, dependendo do contexto, o que torna os programas escritos em PHP difíceis de dimensionar e às vezes demorados para executar.

PHP é uma ótima linguagem para aprender para quem está começando por vários motivos:

- É mais tolerante a erros, o que significa que você é capaz de compilar e executar um programa até chegar a uma parte problemática.
- Há uma abundância de recursos dedicados ao idioma como resultado do grande apoio da comunidade e das ferramentas. O idioma passa por atualizações. Portanto, garanta que você está aprendendo com um tutorial atualizado.
- A configuração é relativamente fácil se comparada a uma linguagem como *Ruby on Rails*. Você pode fazer o *download* do MAMP (para Macs) ou WAMP (para Windows) e deve estar pronto em cinco minutos.

#### MAMP

É um ambiente de servidor local gratuito que pode ser instalado no macOS e Windows com apenas alguns cliques. O MAMP fornece todas as ferramentas necessárias para executar o WordPress em seu PC para fins de teste ou desenvolvimento, por exemplo. Você pode testar facilmente seus projetos em dispositivos móveis. Não importa se você prefere o Apache ou o Nginx ou se deseja trabalhar com PHP, Python, Perl ou Ruby.

Disponível em: https://www.mamp.info/en/windows/ (MAMP, 2020).

### WAMP

O *WampServer* é um ambiente de desenvolvimento *web* para Windows. Permite criar aplicativos da *web* com Apache2, PHP e um banco de dados MySQL.

Disponível em: http://www.wampserver.com/en/. (WAMPSERVER, 2020).

O que pode ser feito com PHP?

- Coleta de dados de formulário.
- Geração de páginas dinâmicas.
- Envio e recebimento de *cookies*.
- Escrita de *scripts* de linha de comando.
- Gravação de scripts do servidor.

### 2.1.3. .NET (C#, VB)

O ASP.NET é a resposta da Microsoft ao Java da *Sun Microsystem* (agora Oracle). A estrutura de aplicativos da *web* é usada para criar *sites* usando linguagens como *Visual Basic* (VB), C#, F# e muito mais.

Seu padrão de arquitetura *Model-View-Controller* (MVC) permite que as tarefas de *back-end* sejam tratadas por um controlador, que interage com um modelo para processar dados. O resultado é, então, apresentado como uma página da *web* de *front-end*.

Criado em 2016 e com código-fonte aberto, o .NET pode se integrar ao iOS, Linux e Android por meio do .NET Core. O código é altamente estável e confiável, tornando-o uma escolha popular para as empresas. Por ser um produto da Microsoft, há um ótimo suporte disponível.

### 2.1.4. C#

C# é uma linguagem de computador de alto nível, o que significa que permite aos desenvolvedores escrever programas independentes de um tipo específico de computador. Idiomas como esses escrevem e leem mais como linguagem humana em vez de linguagem de máquina.

O C# é popular entre os desenvolvedores porque possui o poder do C++ (outra linguagem de *back-end*), mas é mais fácil de usar, porque elimina os erros de comandos que tendem a atrapalhar os usuários do C++.

### 2.1.5. VB

VB é uma linguagem de programação que usa uma interface gráfica do usuário (GUI) para modificar o código escrito na linguagem de programação Basic. É uma linguagem fácil para começar por causa de sua sintaxe direta e seu amplo uso. Como resultado, é frequentemente usado para prototipagem.

A desvantagem da codificação com VB é a grande quantidade de memória necessária para instalar e executar ferramentas de desenvolvimento baseadas em GUI.

O que pode ser feito com .NET?

- Aplicativos *desktop*.
- Aplicativos móveis.
- Jogos.
- Trabalhar com *Big Data* e muito mais.

### 2.1.6. Ruby

*Ruby on Rails* (ou *Ruby*) é uma linguagem de desenvolvimento *web* construída sobre a linguagem de programação *Ruby*. O *Ruby on Rails* possui um conjunto de ferramentas que permitem criar tarefas básicas (por exemplo, você pode criar um *blog* básico com uma linha de código).

*Ruby* envolve pouco trabalho de *back-end*, permitindo que os desenvolvedores criem e iniciem aplicativos rapidamente. É semelhante ao *Python*, pois é fantástico para prototipagem. Como resultado, *Ruby* cresceu em popularidade no início dos anos 2000, mas diminuiu desde então.

*Ruby* é de código aberto, o que significa que pode ser modificado e desenvolvido.

O que pode ser feito com *Ruby*?

- Automatizar tarefas repetidas.
- Construir aplicativos e jogos para dispositivos móveis.
- Criar protótipos e muito mais.

### 2.1.7. Python

Há uma razão pela qual o *Python* é a linguagem de programação que mais cresce. A linguagem versátil é usada para o desenvolvimento na *web* e na área de trabalho. Como no Java, há muitos tutoriais e guias *on-line*, tornando-o uma linguagem acessível para quem está começando. Além disso, sua sintaxe é simples e fácil de entender em relação a outros idiomas.

Como mencionado acima, *Ruby* e *Python* compartilham semelhanças. São linguagens dinamicamente tipadas, de código aberto e orientadas a objetos. As principais diferenças entre os dois são as diferenças de popularidade (*Ruby* está em declínio, enquanto *Python* dispara rapidamente) e as ferramentas na linguagem de *Ruby*.

O que pode ser feito com *Python*?

» Script de *shell* de plataforma cruzada.

» Automação rápida.

» Desenvolvimento simples da web e muito mais.

### 2.1.8. SQL

Linguagem de consulta estruturada ou SQL é a linguagem de consulta mais comum. SQL é usado para interagir com bancos de dados.

O SQL é ótimo para iniciantes, pois é uma linguagem declarativa. Uma linguagem declarativa permite que os codificadores "declarem" os resultados que gostariam de ver, sem especificar as etapas ou o processo para esse resultado.

Uma linguagem de consulta como o SQL é ótima para quem deseja tirar proveito da riqueza de conhecimentos armazenados nos bancos de dados.

O que pode ser feito com SQL?

» Acessar, manipular e criar banco de dados.

### 2.1.9. JavaScript

Para não ser confundido com Java, o *JavaScript* é uma linguagem que pode ser usada para o *front-end* e o *back-end*.

É um ótimo idioma para iniciantes, porque é um idioma de nível superior e há pouca configuração envolvida (você pode começar a codificar no seu navegador).

Como a linguagem é muito flexível, os objetos criados nessa linguagem são lentos. Também é difícil manter e escalar, como na maioria das linguagens de tipo dinâmico.

Dito isto, a linguagem é onipresente, o que significa que a comunidade é grande — proporcionando a você uma enorme quantidade de recursos e muitas oportunidades de emprego.

O que pode ser feito com *JavaScript*?

- Criação de *sites*.
- Aplicativos móveis e de *desktop*.
- Jogos.
- Servidores *web* e muito mais

## 2.2. Ferramentas de *back-end*

Desenvolvimento *back-end* é o lado místico do desenvolvimento *web*. Os desenvolvedores de *back-end* estão envolvidos na construção da lógica real na qual um aplicativo ou site funciona. Algumas das principais habilidades que eles devem ter incluem: um conhecimento profundo da linguagem / estrutura de programação de *back-end*, conformidade de acessibilidade e segurança, um entendimento básico de tecnologias da *web* de *front-end* como HTML, CSS e a capacidade de gerenciar uma hospedagem.

A seguir uma lista das seis principais ferramentas essenciais de desenvolvimento de *back-end*:

- Linguagens e frameworks.
- Servidores da *web*.
- Sistemas de gerenciamento de banco de dados.
- Ambientes de desenvolvimento local.
- Serviços de colaboração.
- Testadores de desempenho de *sites*.

### 2.2.1. Linguagens e *frameworks*

Uma variedade de linguagens de programação e estruturas está envolvida na criação do *software*, também conhecido como *back-end*. Estruturas são bibliotecas de código

pré-escrito com uma estrutura pré-imposta que um desenvolvedor de *back-end* pode usar de acordo com os requisitos e necessidades. Enquanto que uma linguagem de programação é um superconjunto de linguagens de *script* como *Ruby*, *Java*, *Python*, PHP, Perl, *Erlang* e *Node.js*, que podem ser usadas para escrever instruções de execução.

### 2.2.2. Servidores da *web*

Servidores da *web* são programas de computador que armazenam, processam e entregam páginas da *Web* para os usuários. Os mais populares incluem o *Apache* (um servidor da *web* de código aberto atualmente usado por 50% de todos os *sites*) e o NGINX, que é bom para processos de *proxy* reverso, cache, balanceamento de carga e *streaming* de mídia.

### 2.2.3. Sistemas de gerenciamento de banco de dados

Um Sistema de Gerenciamento de Banco de Dados (DBMS) é uma coleção de programas que permite que seus usuários acessem um banco de dados, manipulem, interpretem e representem dados. O MySQL é o banco de dados relacional de código aberto mais popular do mundo. Não é apenas acessível, mas, também, gratuito. Sua facilidade de configuração e performances rápidas o tornam um favorito entre muitos desenvolvedores de *back-end*. Por outro lado, o MongoDB é um sistema de banco de dados NoSQL de código aberto que está intimamente associado a um conjunto de tecnologias baseadas em *JavaScript* como *ExpressJS*, *AngularJS* e *NodeJS*.

### 2.2.4. Ambientes de desenvolvimento local

Todos os desenvolvedores de *back-end* juram a importância de um ambiente de teste local. A vantagem de usar um *site* local visível apenas para você oferece a liberdade de experimentar códigos e experiências antes que o *site* seja publicado. XAMPP e *WampServer* são exemplos de ambientes de desenvolvimento de código aberto para Windows que permitem aos usuários usar aplicativos da *web* com *Apache*, PHP e um banco de dados MySQL.

Ajustar códigos em um ambiente local até que você o aperfeiçoe evita que o site ao vivo falhe devido a um código incorreto.

### 2.2.5. Serviços de colaboração

Como as operações de *back-end* e *front-end* andam de mãos dadas, é ideal que os desenvolvedores de ambos os lados permaneçam conectados. Plataformas como *Slack*, *Asana*, *Jira* e *Trello* ajudam as equipes a trabalhar efetivamente com uma melhor colaboração.

### 2.2.6. Testadores de desempenho do *site*

Na era digital, o tempo médio de atenção de um ser humano é de apenas oito segundos. Portanto, a velocidade na qual um *site* é carregado influencia bastante o tráfego dele. Mesmo um atraso de um segundo pode fazer um cliente se despedir de seu *site*. É por isso que os desenvolvedores de *back-end* consideram os testadores de velocidade como o *Google PageSpeed Insights* e o *Full-Page Load Tester* como importantes ferramentas de desenvolvimento de *back-end*. Eles não apenas fornecem relatórios da velocidade do site, mas, também, recomendam ajustes que você pode fazer para tornar sua página mais rápida.

## 2.3. Arquitetura *back-end*

O *back-end* é toda a tecnologia necessária para processar a solicitação recebida e gerar e enviar a resposta ao cliente. Isso geralmente inclui três partes principais:

- **O servidor**: este é o computador que recebe solicitações.
- **A aplicação**: este é o aplicativo em execução no servidor que escuta solicitações, recupera informações do banco de dados e envia uma resposta.
- **O banco de dados**: os bancos de dados são usados para organizar e manter dados.

Um servidor é simplesmente um computador que executa solicitações recebidas. Embora existam máquinas fabricadas e otimizadas para esse fim específico, qualquer computador conectado a uma rede pode atuar como servidor. De fato, você costuma usar seu próprio computador como servidor ao desenvolver aplicativos.

O servidor executa um aplicativo que contém lógica sobre como responder a várias solicitações com base no verbo HTTP e no *Uniform Resource Identifier* (URI). O par de um verbo HTTP e um URI é chamado de rota e correspondê-los com base em uma solicitação é chamado de roteamento.

#### Verbo HTTP

O HTTP define um conjunto de métodos de solicitação para indicar a ação desejada a ser executada para determinado recurso. Embora também possam ser substantivos, esses métodos de solicitação às vezes são chamados de **verbos HTTP**. Cada um deles implementa uma semântica diferente, mas alguns recursos comuns são compartilhados por um grupo deles: por exemplo, um método de solicitação pode ser seguro, idempotente ou armazenável em cache.

Algumas dessas funções de manipulador serão *middleware*. Nesse contexto, *middleware* é qualquer código que é executado entre o servidor que recebe uma solicitação e envia uma resposta. Essas funções de *middleware* podem modificar o objeto de solicitação, consultar o banco de dados ou processar a solicitação de entrada. As funções de *middleware* geralmente terminam passando o controle para a próxima função de *middleware*, em vez de enviar uma resposta.

Eventualmente, uma função de *middleware* quando chamada encerrará o ciclo de solicitação-resposta enviando uma resposta HTTP de volta ao cliente.

Frequentemente, os programadores usam uma estrutura como *Express* ou *Ruby on Rails* para simplificar a lógica do roteamento. Por enquanto, pense que cada rota pode ter uma ou várias funções de manipulador executadas sempre que uma solicitação para essa rota (verbo HTTP e URI) for correspondida.

Os dados que o servidor envia de volta podem vir de diferentes formas. Por exemplo, um servidor pode servir um arquivo HTML, enviar dados como JSON ou enviar apenas um código de *status* HTTP. Você provavelmente já viu o código de *status* "404 – Não encontrado" sempre que tentou navegar para um URI que não existe, mas existem muitos outros códigos de *status* que indicam o que aconteceu quando o servidor recebeu a solicitação.

Já os bancos de dados são comumente usados no *back-end* de aplicativos da *web*. Esses bancos de dados fornecem uma interface para salvar dados de maneira persistente na memória. Armazenar os dados em um banco de dados reduz a carga na memória principal da CPU do servidor e permite que os dados sejam recuperados se o servidor travar ou perder energia.

Muitas solicitações enviadas ao servidor podem exigir uma consulta ao banco de dados. Um cliente pode solicitar informações armazenadas no banco de dados ou pode enviar dados com sua solicitação para serem adicionados ao banco de dados.

Vamos tornar tudo isso um pouco mais concreto, seguindo um exemplo das principais etapas que acontecem quando um cliente faz uma solicitação ao servidor.

1. Maria está comprando no SuperCoolShop.com. Ela clica na imagem de uma capa do *smartphone* e esse evento de clique faz uma solicitação GET para http://www.SuperCoolShop.com/products/66432.

   Lembre-se, GET descreve o tipo de solicitação (o cliente está apenas solicitando dados, não alterando nada). O URI (identificador uniforme de recursos) /products/66432

especifica que o cliente está procurando mais informações sobre um produto e esse produto tem um ID 66432.

O SuperCoolShop possui um grande número de produtos e muitas categorias diferentes para filtrá-los; portanto, o URI real seria mais complicado do que isso. Mas esse é o princípio geral de como as solicitações e os identificadores de recursos funcionam.

2. O pedido de Maria viaja pela internet para um dos servidores da SuperCoolShop. Essa é uma das etapas mais lentas do processo, porque a solicitação não pode ir mais rápido que a velocidade da luz e pode ter uma longa distância a percorrer. Por esse motivo, os principais *sites* com usuários em todo o mundo terão muitos servidores diferentes e direcionarão os usuários para o servidor mais próximo deles!

3. O servidor, que está atendendo a solicitações de todos os usuários, recebe a solicitação de Maria!

4. Os ouvintes de eventos que correspondem a essa solicitação (o verbo HTTP: GET e o URI:) /products/66432 são acionados. O código que é executado no servidor entre a solicitação e a resposta é chamado de *middleware*.

5. Ao processar a solicitação, o código do servidor faz uma consulta ao banco de dados para obter mais informações sobre este estojo para *smartphone*. O banco de dados contém todas as outras informações que Maria deseja saber sobre este estojo para *smartphone*: o nome do produto, o preço do produto, algumas análises de produtos e uma sequência que fornecerá um caminho para a imagem do produto.

6. A consulta ao banco de dados é executada e o banco de dados envia os dados solicitados de volta ao servidor. Vale a pena notar que as consultas ao banco de dados são uma das etapas mais lentas desse processo. A leitura e gravação da memória estática é bastante lenta e o banco de dados pode estar em uma máquina diferente do servidor original. Essa consulta em si pode ter que passar pela internet!

7. O servidor recebe do banco de dados os dados necessários e agora está pronto para construir e enviar sua resposta de volta ao cliente. Esse corpo de resposta tem todas as informações necessárias pelo navegador para mostrar a Maria mais detalhes (preço, revisões, tamanho, etc.) sobre a capa de celular em que ela está interessada. O cabeçalho da resposta conterá um código de *status* HTTP 200 para indicar que a solicitação foi conseguiu.

8. A resposta viaja pela internet, de volta ao computador de Maria.

9. O navegador de Maria recebe a resposta e usa essas informações para criar e renderizar a visão que Maria finalmente vê!

## 2.4. Estudo de caso

Nesta seção vamos conhecer em detalhes como funciona a infraestrutura de *back-end* do *Spotify*, com um texto adaptado de (FORSUM, 2013).

### 2.4.1. Background

No *Spotify*, as coisas crescem o tempo todo. O número de usuários diários, o número de nós de *back-end* que alimentam o serviço, o número de plataformas de *hardware* em que os clientes executam, o número de equipes de desenvolvimento que trabalham com os produtos, o número de aplicativos externos que é hospedado na plataforma, o número de músicas que possui no catálogo.

À medida que o *Spotify* cresce, é preciso navegar com cuidado em torno de muitas coisas que podem reduzir a velocidade do desenvolvimento. Grandes esforços são realizados para eliminar dependências entre equipes e remover complexidade desnecessária da arquitetura.

Um conceito-chave no *Spotify* é que cada equipe de desenvolvimento deve ser autônoma. Uma equipe de desenvolvimento (ou "esquadrão" no jargão do *Spotify*) sempre deve poder se mover independentemente de outros esquadrões. Mesmo se houver uma dependência entre dois esquadrões, sempre existe uma maneira de o esquadrão dependente avançar. Para permitir que todos os esquadrões progridam, mesmo tendo dependências de outro esquadrão, algumas ideias estratégicas são usadas para que seja possível aplicar em qualquer lugar: no modelo de código transparente e na infraestrutura de autoatendimento.

Todo o código do *Spotify* está disponível para todos os desenvolvedores em um modelo de código transparente. Isso significa que todo o código no cliente *Spotify*, no *back-end* do *Spotify* e na infraestrutura do *Spotify* está disponível para todos os desenvolvedores do *Spotify* para ler ou alterar. Se um esquadrão está bloqueando outro esquadrão para alterar algum código, ele sempre tem a opção de prosseguir e fazer a alteração sozinho.

Na prática, o modelo de código transparente do *Spotify* funciona por todas as equipes que compartilham o mesmo servidor *git* centralizado. Cada repositório *git* tem um proprietário de sistema dedicado que cuida do código e garante que ele não se perca. O modelo de código transparente garante que todos possam progredir o tempo todo e que todos tenham acesso ao código de todos. Isso mantém o *Spotify* sempre avançando e oferece um ambiente de trabalho positivo e aberto.

Toda a infraestrutura necessária deve estar disponível como uma entidade de autoatendimento. Dessa forma, não há necessidade de esperar que outra equipe obtenha *hardware*, configure um cluster de armazenamento ou faça alterações na configuração. A infraestrutura de *back-end* do *Spotify* é formada por várias camadas de *hardware* e *software*, que variam de máquinas físicas a soluções de mensagens e armazenamento.

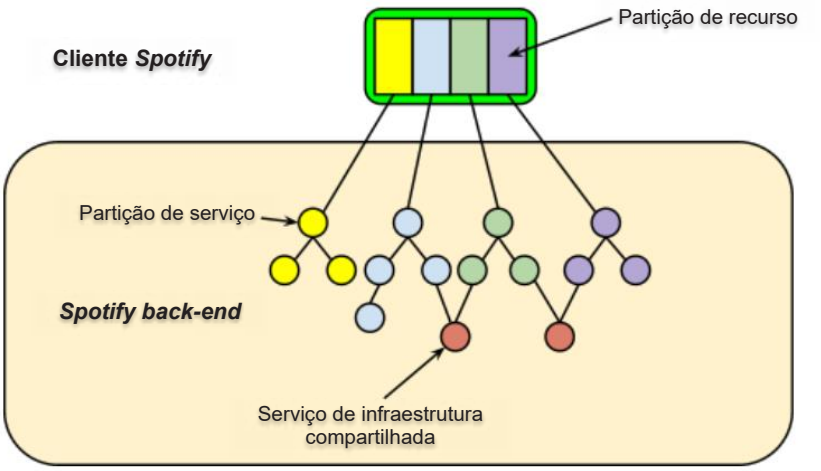
O *Spotify* tenta usar ferramentas de código aberto sempre que possível. Como o *Spotify* está constantemente pressionando os limites de escalabilidade do *software* que está sendo usando no *back-end*, é preciso ser capaz de melhorar o *software* que é usado em áreas críticas. O *Spotify* procura sempre contribuir em muitos dos projetos de código aberto que eles usam, por exemplo, *Apache Cassandra* e *ZMQ*. Não é usado por eles quase nenhum *software* proprietário simplesmente porque não podem confiar que será possível adaptá-lo às crescentes necessidades.

No *Spotify*, acredita-se fortemente em indivíduos capacitados. Isso é refletido em sua organização com equipes autônomas. Para os engenheiros, existem muitas possibilidades de mudar e tentar trabalhar em outras áreas dentro do *Spotify* para garantir que todos permaneçam apaixonados por seu trabalho. Existem dias de folga regularmente, onde as pessoas podem experimentar praticamente qualquer ideia que tenham.

### 2.4.2. Arquitetura

A Figura 3 apresenta a arquitetura *back-end* do *Spotify*.

Figura 3. Arquitetura *back-end* do *Spotify*




Fonte: Forsum (2013).

Qualquer arquitetura que precise lidar com o volume de usuários que o *Spotify* precisa particionar o problema. A arquitetura do *Spotify* particiona o problema de várias maneiras

diferentes. Primeiro, particionar por recursos. Uma descrição um pouco simplificada é que toda a área da tela física de todas as páginas e visualizações dos clientes pertence a algum esquadrão. Todos os recursos dos clientes do *Spotify* pertencem a um esquadrão específico. O esquadrão é responsável por esse recurso em todas as plataformas – desde a aparência em um dispositivo iOS ou navegador até as solicitações em tempo real tratadas pelo *back-end* do *Spotify* até a trituração de dados orientada por lote que ocorre no *cluster Hadoop*.

Se um recurso falhar, os outros recursos dos clientes são independentes e continuarão funcionando. Se houver uma dependência fraca entre os recursos, a falha de um recurso pode levar à degradação do serviço de outro recurso, mas não à falha de todo o serviço *Spotify*.

Como todos os usuários não estão usando todos os recursos ao mesmo tempo, o número de usuários que precisa ser tratado pelo *back-end* de um recurso específico é geralmente muito menor que o número de usuários de todo o serviço *Spotify*.

Como todo o conhecimento em torno de um recurso em particular está concentrado em um esquadrão, é muito fácil usar os recursos de teste A / B, observando os dados coletados e tomando uma decisão informada com todas as pessoas relevantes envolvidas. O particionamento de recursos oferece escalabilidade, confiabilidade e uma maneira eficiente de concentrar os esforços da equipe.

### 2.4.3. Infraestrutura de *back-end*

Depois de particionar o problema por recurso e dar a um esquadrão multifuncional altamente qualificado a missão de cuidar e trabalhar com esse recurso, a questão agora é: como construir uma infraestrutura que suporte esse esquadrão com eficiência?

Como garantir que a equipe desenvolva seus recursos a uma velocidade vertiginosa sem correr o risco de ser bloqueada por outras equipes? Como a infraestrutura pode resolver os problemas difíceis relacionados à escala global? Já falamos anteriormente sobre o modelo de código transparente, que sempre permite que uma equipe avance, mas há outras partes da organização além dos esquadrões de desenvolvimento de recursos.

Em muitas organizações, tem-se administradores de banco de dados que cuidam dos bancos de dados e seus esquemas e, normalmente, precisa passar por um departamento de operações para obter o *hardware* alocado nos *data centers*, etc. Essas funções especiais na organização tornam-se gargalos quando há 100 esquadrões simultaneamente exigindo seus serviços. Para resolver isso, o *Spotify* está desenvolvendo uma infraestrutura de *back-end* que é totalmente de autoatendimento. Isso significa que qualquer esquadrão

pode começar a desenvolver e interagir em um serviço no ambiente ao vivo sem precisar interagir com o restante da organização.

Para conseguir isso, é preciso resolver uma série de problemas em várias áreas diferentes. Vamos citar alguns importantes aqui.

» **Aprovisionamento**: ao desenvolver um novo recurso, um esquadrão normalmente precisa implantar esse serviço em vários locais. O *Spotify* está construindo uma infraestrutura para permitir que o esquadrão decida por si próprio se o serviço deve ser implantado nos próprios *data centers* do *Spotify* ou se o recurso pode usar uma oferta de nuvem pública. A infraestrutura do *Spotify* se esforça para minimizar a diferença entre executar nos próprios *data centers* e em uma nuvem pública. Em resumo, obtém-se melhor latência e um ambiente mais estável com os próprios *data centers*. Em uma nuvem pública, obtém-se um provisionamento de *hardware* muito mais rápido e possibilidades de dimensionamento muito mais dinâmicas. A Figura 4 ilustra clientes Spotify conectando-se a um *data center* mais próximo.

» **Armazenamento**: a maioria dos recursos requer algum tipo de armazenamento, sendo exemplos óbvios as listas de reprodução e o recurso “seguir”. Criar uma solução de armazenamento para um recurso que milhões de pessoas usarão não é uma tarefa fácil, e há muitas coisas a serem consideradas: padrões de acesso, *failover* entre *sites*, capacidade, consistência, *backups*, degradação no caso de uma divisão líquida entre *sites*, etc. Não há uma maneira fácil de cumprir todos esses requisitos de forma genérica. Para cada recurso, o esquadrão precisará criar uma solução de armazenamento que atenda às necessidades desse serviço específico. A infraestrutura do *Spotify* oferece algumas opções diferentes de armazenamento: *Cassandra*, *postgreSQL* e *memcached*.

Se os dados do recurso precisarem ser particionados, o esquadrão precisa implementar o *sharding* em seus serviços, no entanto, muitos serviços contam com o *Cassandra* fazendo réplicas completas de dados entre *sites*. A configuração de um *cluster* de armazenamento completo com replicação e *failover* entre *sites* é complicada, por isso o *Spotify* está construindo uma infraestrutura para configurar e manter os *clusters Cassandra* ou *postgreSQL* de vários *sites* como uma unidade. Para as pessoas que criam aplicativos na API do *Spotify*, haverá uma opção de armazenamento como serviço que não exigirá nenhuma configuração de *clusters*. A opção de armazenamento como serviço será limitada a um armazenamento de valores-chave muito simples.

Figura 4. Clientes *Spotify* conectando-se a um *data center* mais próximo


Fonte: Forsum, (2013).

» **Mensagens:** os clientes *Spotify* e os serviços de *back-end* se comunicam usando os seguintes paradigmas: solicitação de resposta, mensagens e *pubsub*. O *Spotify* criou sua própria camada de mensagens de baixa latência e sobrecarga e planeja estendê-la com altas garantias de entrega, roteamento de *failover* e balanceamento de carga mais sofisticado.

» **Planejamento de capacidade:** o crescimento do *Spotify* direciona uma grande quantidade de tráfego para o *back-end*. Cada esquadrão deve garantir que seus recursos sempre sejam dimensionados para a carga atual. O esquadrão pode optar por acompanhar isso manualmente, monitorando o tráfego de seus serviços, identificando e corrigindo gargalos e expandindo conforme necessário. Também está construindo uma infraestrutura que permite que os esquadrões escalem seus serviços automaticamente com carga. O dimensionamento automático normalmente funciona apenas para gargalos já conhecidos, portanto, sempre há certo nível de monitoramento humano com o qual o esquadrão precisa lidar. A infraestrutura do *Spotify* permite a fácil criação de gráficos e alertas para dar suporte a isso.

» **Isolamento de outros serviços:** à medida que novos recursos e serviços são desenvolvidos, eles tendem a se chamar de maneiras não triviais. É muito importante que todos os esquadrões sintam que podem correr a toda velocidade, minimizando o risco de afetar negativamente outras partes do *Spotify*. Para evitar isso, a camada de mensagens do *Spotify* possui um limite de taxa e sistema de permissões. Os limites de taxa têm um limite-padrão – isso permite que os esquadrões liguem para outros serviços para tentar algo. Se um tráfego excepcionalmente pesado for previsto, os esquadrões precisariam coordenar e concordar em como lidar com isso juntos. Recursos diferentes sempre são executados em servidores ou máquinas virtuais separadas para evitar que um serviço com comportamento inadequado desative outro.

# CAPÍTULO 3

## DESENVOLVEDOR FRONT-END,BACK-END OU FULL-STACK?

Então você decidiu por uma carreira profissional de desenvolvedor *web*. Mas que tipo de desenvolvedor *web* você pretende ser? Desenvolvedor *front-end*? Desenvolvedor *back-end*? Ou você quer ser híbrido: o desenvolvedor *full-stack*?

Vamos examinar detalhadamente cada um desses três tipos de áreas de desenvolvimento: *front-end*, *back-end* e *full-stack*. Então, no final desta leitura, você terá uma ideia melhor de onde estão suas habilidades e interesses, para qual carreira de desenvolvedor você deve se dedicar.

### 3.1. Principais habilidades de um desenvolvedor *front-end*

As principais preocupações de um desenvolvedor *front-end* estão relacionadas à camada de apresentação, eles precisam ter alguma visão artística para apresentar os dados; isso geralmente implica dominar HTML, CSS, alguns pré-processadores CSS, como SAS, e algumas estruturas *JavaScript* (convencionais), como *Angular*, *React* ou *Vue*. Um bom desenvolvedor *front-end* também deve ter um bom entendimento da interação, segurança e desempenho com base em eventos.

### 3.2. Principais habilidades de um desenvolvedor *back-end*

Os desenvolvedores *back-end* trabalham na implementação da lógica de negócios. Eles precisam ter conhecimento de estruturas, arquitetura de *software*, padrões de *design*, bancos de dados, APIs, interconectividade, *DevOps,* etc. Eles precisam ser capazes de gerenciar conceitos abstratos e lógica complexa.

Um desenvolvedor qualificado terá um profundo conhecimento de servidores e bancos de dados (SQL ou no-SQL), camada de API e linguagens de programação como *Java*, *Python*, PHP, C# e go.

### 3.3. Principais habilidades de um desenvolvedor *full-stack*

Um desenvolvedor *full-stack* terá uma combinação de habilidades de desenvolvimento *front-end* e *back-end*. Um desenvolvedor *full-stack* significa ter uma visão holística – comparar os prós e contras do *back-end* e do *front-end* antes de determinar onde a lógica deve estar.

Para um verdadeiro desenvolvedor *full-stack*, isso significa não apenas ser capaz de conhecer as tecnologias *front-end* e *back-end* e como aplicá-las corretamente. Também significa ser capaz de projetar uma solução completa – e ver onde deve estar a separação da lógica.

## 3.4. Principais diferenças entre desenvolvedores *front-end* e *back-end*

O desenvolvedor *front-end* requer interação com os usuários e a lógica, trabalhando com diferentes navegadores e recursos e uma compreensão da maneira como o conteúdo é apresentado nos desktops e dispositivos da plataforma. Essencialmente, o desenvolvimento *front-end* tem tudo a ver com *design* e desenvolvimento de interface do Usuário (UI) e Experiência do Usuário (UX), garantindo uma interface intuitiva e responsiva que funciona em navegadores / dispositivos.

O desenvolvimento de *back-end*, por outro lado, envolve a interação com os clientes e um entendimento de como modelar domínios e relacionamentos de negócios. Há, também, um forte foco em dados, com desenvolvedores de *back-end* trabalhando em vários bancos de dados e integrações de diferentes provedores de serviços. No desenvolvimento de *back-end*, você nunca precisa se preocupar com a aparência de alguma coisa – apenas sobre como ela funciona e garantir que a funcionalidade e o *design* subjacente sejam adequados.

Ah! Antes de decidir sobre um nicho no desenvolvimento *front-end* ou *back-end*, experimente os dois. Procure alguns cursos de curta duração e crie seu próprio projeto onde você precisa fazer tudo.

**Tudo o que você precisa para ser um programador *front-end***. Disponível em: https://www.hostgator.com.br/blog/tudo-o-que-voce-precisa-para-ser-um-programador-front-end/. (HOSTGATOR BRASIL, 2018).

**Como é trabalhar como desenvolvedor *front-end*, por Felipe Fialho**. Disponível em: https://medium.com/trainingcenter/como-%C3%A9-trabalhar-como-desenvolvedor-front-end-por-felipe-fialho-1e3efbadef90. (OLIVEIRA, 2017).

**Como é trabalhar como Desenvolvedor(a) *back-end*, por Giovanni Cruz**. Disponível em: https://medium.com/trainingcenter/como-%C3%A9-trabalhar-como-desenvolvedor-backend-b40255d75626. (CRUZ, 2017).

**Como se tornar um desenvolvedor *back-end*? 4 dicas para começar a sua carreira**. Disponível em: https://take.net/blog/devs/desenvolvedor-back-end/. (TAKE BLOG, 2019).

**Desenvolvimento *full-stack*: o que significa ser um profissional completo**. Disponível em: https://movile.blog/desenvolvimento-full-stack-o-que-significa-ser-um-profissional-completo/. (MONTEIRO, 2019).

**O que faz um desenvolvedor *full-stack*?** Disponível em: https://www.igti.com.br/blog/o-que-faz-um-desenvolvedor-full-stack/. (IGTI BLOG, 2018).

# REFERÊNCIAS


ALLIED. **7 Surprising Facts About Cloud Computing**. 2015. Disponível em: https://www.alliedtelecom.net/facts-about-cloud-computing/. Acesso em 23 mar. 2020.

BELSHE, M.; PEON, R. **Hypertext Transfer Protocol Version 2**. 2015. Disponível em: https://httpwg.org/specs/rfc7540.html. Acesso em 19 mar. 2020.

CONFLUENT. **Featuring Apache Kafka in the Netflix Studio and Finance World.** 2020. Disponível em: https://www.confluent.io/blog/how-kafka-is-used-by-netflix/. Acesso em 28 mar. 2020.

CONVENTIONAL COMMITS. **Conventional Commits 1.0.0-beta.**2. 2020. Disponível em: https://www.conventionalcommits.org/en/v1.0.0-beta.2/. Acesso em 29 out. 2020.

CROCKFORD, D. **Introducing JSON**. 2020. Disponível em: https://www.json.org/json-en.html. Acesso em 19 mar. 2020.

CRUZ, G. **Como é trabalhar como Desenvolvedor(a) Back-End, por Giovanni Cruz**. 2017. Disponível em: https://medium.com/trainingcenter/como-%C3%A9-trabalhar-como-desenvolvedor-backend-b40255d75626. Acesso em 23 mar. 2020.

DAVIS, D; VENNAM, B**. Knative 101: Exercises designed to help you achieve an understanding of Knative.** 2019. Disponível em: https://developer.ibm.com/tutorials/knative-101-labs/. Acesso em 24 mar. 2020.

DEVMEDIA. **O que é HTML5.** 2012. Disponível em: https://www.devmedia.com.br/o-que-e-o-html5/25820. Acesso em 20 mar. 2020.

ECMA. **ECMAScript Language Specification**. 2018. Disponível em: http://ecma-international.org/ecma-262/9.0/index.html#Title. Acesso em 19 mar. 2020.

FREECODCAMP. **Introduction to NPM Scripts**. 2018. Disponível em: https://www.freecodecamp.org/news/introduction-to-npm-scripts-1dbb2ae01633/. Acesso em 29 out. 2020.

FORSUM, G. **Backend infrastructure at Spotify**. 2013. Disponível em: https://labs.spotify.com/2013/03/15/backend-infrastructure-at-spotify/. Acesso em 21 mar. 2020.

FOWLER, C. O programador apaixonado. Casa do Código, 1º ed, 2014.

GITHUB. 2020. Disponível em: https://github.com/nodejs/node. Acesso em 29 out. 2020.

GRUNT. 2020. Disponível em: https://gruntjs.com/. Acesso em 29 out. 2020.

GULP. 2020. Disponível em: https://gulpjs.com/. Acesso em 29 out. 2020.

HOSTGATOR BRASIL**. Tudo o que você precisa para ser um programador front-end**. 2018. Disponível em: https://www.hostgator.com.br/blog/tudo-o-que-voce-precisa-para-ser-um-programador-front-end/. Acesso em 23 mar. 2020.

IBM. **Cloud Storage**. 2019. Disponível em: https://www.ibm.com/cloud/learn/cloud-storage. Acesso em 25 mar. 2020.

IBM. **Hypervisors**. 2019. Disponível em: https://www.ibm.com/cloud/learn/hypervisors. Acesso em 25 mar. 2020.

IBM. **Istio**. 2019. Disponível em: https://www.ibm.com/cloud/learn/istio. Acesso em 24 mar. 2020.

IBM. **Microservices.** Disponível em: https://www.ibm.com/cloud/learn/microservices. Acesso em 24 mar. 2020.

IBM. **Virtual Machines (VMs)**. 2019. Disponível em: https://www.ibm.com/cloud/learn/virtual-machines. Acesso em 25 mar. 2020.

IBM. **Desktop-as-a-Service (DaaS).** 2018. Disponível em: https://www.ibm.com/cloud/learn/desktop-as-a-service. Acesso em 25 mar. 2020.

IBM. **The state of container-based application development**. 2017. Disponível em: https://www.ibm.com/cloud-computing/info/container-development/. Acesso em 24 mar. 2020.

IBM Cloud. Containerization Explained. 2019. Disponível em: https://www.youtube.com/watch?v=0qotVMX-J5s&feature=emb_logo. Acesso em 29 out. 2020.

IBM Cloud. Kubernetes Explained. 2019. Disponível em: https://www.youtube.com/watch?v=aSrqRSk43lY&feature=emb_logo. Acesso em 29 out. 2020.

IBM Cloud. Implantações Kubernetes: comece rapidamente. 2019. Disponível em: https://www.youtube.com/watch?v=Sulw5ndbE88&feature=emb_logo. Acesso em 29 out. 2020.

IBM Cloud. Virtualization Explained. 2019. Disponível em: https://www.youtube.com/watch?v=FZR0rG3HKIk&feature=emb_logo. Acesso em 29 out. 2020.

IBM Cloud. GPUs: Explained. 2019. Disponível em: https://www.youtube.com/watch?v=LfdK-voSbGI&feature=emb_logo. Acesso em 29 out. 2020.

IGTI BLOG. **O que faz um Desenvolvedor Full Stack?** 2018. Disponível em: https://www.igti.com.br/blog/o-que-faz-um-desenvolvedor-full-stack/. Acesso em 23 mar. 2020.

IPERIUS BACKUP BRASIL. **Computação em nuvem: Inovações em TI para impulsionar os negócios**. 2019. Disponível em: https://www.iperiusbackup.net/pt-br/computacao-em-nuvem-inovacoes-em-ti-para-impulsionar-os-negocios/. Acesso em: 24 mar. 2020.

KATZ, B. **How Much Does It Cost to Fight in the Streaming Wars?** 2019. Disponível em: https://observer.com/2019/10/netflix-disney-apple-amazon-hbo-max-peacock-content-budgets/. Acesso em 28 mar. 2020.

KIM, J. **Kubernetes Networking: a lab on basic networking concepts**. 2019. Disponível em: https://developer.ibm.com/tutorials/kubernetes-networking-101-lab/. Acesso em 24 mar. 2020.

LINDLEY, C. **Front-End Developer Handbook 2018**. 2018. Disponível em: https://frontendmasters.com/books/front-end-handbook/2018/. Acesso em 18 mar. 2020.

LINTHICUM, D. **The essential guide to software containers for application development**. 2020. Disponível em: https://techbeacon.com/enterprise-it/essential-guide-software-containers-application-development. Acesso em 24 mar. 2020.

MAMP. 2020. Disponível em: https://www.mamp.info/en/windows/. Acesso em 29 out. 2020.

MATTOKA, M. **Are you sure you're using microservices?** 2019. Disponível em: https://blog.softwaremill.com/are-you-sure-youre-using-microservices-f8d4e912d014. Acesso em 28 mar. 2020.

MDN. **CSS Reference**. 2020. Disponível em: https://developer.mozilla.org/en-US/docs/Web/CSS/Reference. Acesso em 19 mar. 2020.

MONTEIRO, C. **Desenvolvimento full stack: o que significa ser um profissional completo**. 2019. Disponível em: https://movile.blog/desenvolvimento-full-stack-o-que-significa-ser-um-profissional-completo/. Acesso em 23 mar. 2020.

NARKNEDE, N; SHAPIRA, G; PALINO, T. **Kafka: The definitive guide: real-time data and streams Processing at scale.** 2017. O'Reilly Media, Inc.

NEEMAN, T. **Debug and log your Kubernetes applications.** 2019. Disponível em: https://developer.ibm.com/tutorials/debug-and-log-your-kubernetes-app/. Acesso em 24 mar. 2020.

NETFLIX. **Evolution of the Netflix Data Pipeline**. 2016. Disponível em: https://netflixtechblog.com/evolution-of-the-netflix-data-pipeline-da246ca36905. Acesso em 28 mar. 2020.

NETFLIX. **Spark and Spark Streaming at Netfix-(Kedar Sedekar and Monal Daxini, Netflix)**. 2015. Disponível em: https://www.slideshare.net/SparkSummit/spark-and-spark-streaming-at-netfix-sedakar-daxini. Acesso em 28 mar. 2020.

NODEJS. 2020. Disponível em: https://nodejs.org. Acesso em 29 out. 2020.

OLIVEIRA, W. **Como é trabalhar como Desenvolvedor Front-End, por Felipe Fialho**. 2017. Disponível em: https://medium.com/trainingcenter/como-%C3%A9-trabalhar-como-desenvolvedor-front-end-por-felipe-fialho-1e3efbadef90. Acesso em 23 mar. 2020.

PAGANI, T. **Documentos acessíveis com WAI-ARIA em HTML5.** 2011. Disponível em: https://tableless.com.br/documentos-acessiveis-com-aria-em-html5/. Acesso em 20 mar. 2020.

PARIS-WHITE, D. **Scale security while innovating microservices fast.** 2018. Disponível em: https://www.ibm.com/cloud/blog/scale-security-innovating-microservices-fast. Acesso em 24 mar. 2020.

PINTEREST. 2020. Disponível em: https://i.pinimg.com/564x/57/b8/05/57b805a0f624c0837b10e0c4cc5b1c55.jpg. Acesso em 29 out 2020.

REDMONK. **The Continued Rise of Apache Kafka.** 2017. Disponível em: https://redmonk.com/fryan/2017/05/07/the-continued-rise-of-apache-kafka/. Acesso em 28 mar. 2020.

SHAPLAND, R; COLE, B. **What are cloud containers and how do they work?** 2019. Disponível em: https://searchcloudsecurity.techtarget.com/feature/Cloud-containers-what-they-are-and-how-they-work. Acesso em 24 mar. 2020.

SOMAN, C. *et al*. **uReplicator: Uber Engineering's Robust Apache Kafka Replicator**. 2016. Disponível em: https://eng.uber.com/ureplicator-apache-kafka-replicator/. Acesso em 29 mar. 2020.

STORYBOOK. Build bulletproof UI components faster. 2020. Disponível em: https://storybook.js.org/. Acesso em 29 out. 2020.

TAKE BLOG. **Como se tornar um desenvolvedor back-end? 4 dicas para começar a sua carreira**. 2019. Disponível em: https://take.net/blog/devs/desenvolvedor-back-end/. Acesso em 23 mar. 2020.

TUTORIALSTEACHER.COM. **JavaScript Tutorial**. 2020. Disponível em: https://www.tutorialsteacher.com/javascript/javascript-tutorials. Acesso em 29 out. 2020.

TUTORIALSTEACHER.COM. **Node.js Process Model**. 2020. Disponível em: https://www.tutorialsteacher.com/nodejs/nodejs-process-model. Acesso em 26 mar. 2020.

VENNAM, S. **Kubernetes Clusters: Architecture for Rapid, Controlled Cloud App Delivery.** 2019. Disponível em: https://www.ibm.com/cloud/blog/new-builders/kubernetes-clusters-architecture-for-rapid-controlled-cloud-app-delivery. Acesso em 24 mar. 2020.

WACHAL, M. **Digital transformation with streaming application development.** 2019. Disponível em: https://blog.softwaremill.com/digital-transformation-with-streaming-application-development-100fb4189149. Acesso em 28 mar. 2020.

WAMP SERVER. 2020. Disponível em: https://www.wampserver.com/en/. Acesso em 29 out. 2020.

WARSKI, A. **Evaluating persistent, replicated message queues.** 2017. Disponível em: https://softwaremill.com/mqperf/. Acesso em 28 mar. 2020.

WARSKI, A. **Event sourcing using Kafka.** 2018. Disponível em: https://blog.softwaremill.com/event-sourcing-using-kafka-53dfd72ad45d. Acesso em 28 mar. 2020.

WEBPACK. 2020. Disponível em: <https://webpack.js.org/. Acesso em 29 out. 2020.

WHATWG. **DOM Living Standard**. 2020. Disponível em: https://dom.spec.whatwg.org/. Acesso em 19 mar. 2020.

WHATWG. **HTTP Living Standard**. 2020. Disponível em: https://html.spec.whatwg.org/multipage/introduction.html#introduction. Acesso em 19 mar. 2020.

WHATWG. **URL Living Standard**. 2020. Disponível em: https://url.spec.whatwg.org/. Acesso em 19 mar. 2020.

WODEHOUSE, C. **The role of a front-end web developer: creating user experience & interactivity**. 2015. Disponível em: https://www.upwork.com/hiring/development/front-end-developer/. Acesso em 18 mar. 2020.

W3C BRASIL. **Cartilha de Acessibilidade na Web.** 2020. Disponível em: https://ceweb.br/cartilhas/cartilha-w3cbr-acessibilidade-web-fasciculo-I.html. Acesso em 20 mar. 2020.

W3 SCHOOLS. **Cloud Computing Architecture**. 2020. Disponível em: https://www.w3schools.in/cloud-computing/cloud-computing-architecture/. Acesso em 23 mar. 2020.

W3TECHS. **Usage statistics of PHP for websites**. 2020. Disponível em: https://w3techs.com/technologies/details/pl-php. Acesso em 21 mar. 2020.

WOODIE, A. **The Real-Time Rise of Apacha Kafka.** 2016. Disponível em: https://www.datanami.com/2016/04/06/real-time-rise-apache-kafka/. Acesso em 28 mar. 2020.