PROGRAMAÇÃO WEB – FRONT-END

AULA 5

Prof. Mauricio Antonio Ferste

CONVERSA INICIAL

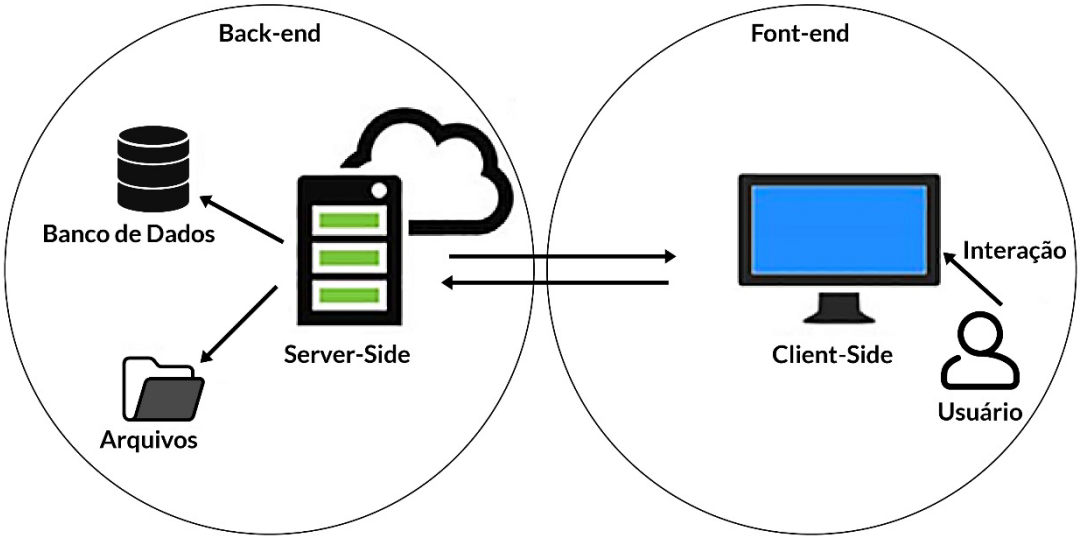
O conceito de *back-end* e *front-end* são termos amplamente utilizados na indústria de desenvolvimento de *software* e se referem a duas áreas distintas de uma aplicação. O *back-end* refere-se à parte da aplicação que lida com a lógica de negócios, processamento de dados e interação com bancos de dados. É responsável por processar as solicitações recebidas do *front-end*, executar as operações necessárias e fornecer as respostas apropriadas. O *back-end* geralmente é desenvolvido usando linguagens de programação como Python, Java, Ruby, Node.js, entre outras. Ele também envolve o gerenciamento de bancos de dados, conexões com serviços externos e a implementação das regras de negócio da aplicação.

Por outro lado, o *front-end* refere-se à parte da aplicação, que é diretamente acessada pelos usuários. É responsável pela interface do usuário, apresentação visual dos dados e interação com o usuário. O *front-end* é desenvolvido usando linguagens como HTML, CSS e JavaScript, e *frameworks* como Angular, React, Vue.js, entre outros. Ele lida com a exibição de conteúdo, formulários, validações de entrada, manipulação de eventos e outras interações visuais. Em uma arquitetura típica de aplicação *web*, o *front-end* é executado no navegador do cliente e envia solicitações ao *back-end* para obter dados ou realizar ações específicas. O *back-end* processa essas solicitações, executa as operações necessárias e retorna as respostas ao *front-end*, que então as exibe ao usuário. Essa divisão entre *back-end* e *front-end* permite uma separação clara de responsabilidades e a especialização de equipes de desenvolvimento. As equipes de *back-end* se concentram na lógica de negócios e no processamento de dados, enquanto as equipes de *front-end* se concentram na criação de interfaces amigáveis e interativas para os usuários.

No entanto, é importante ressaltar que essa separação não significa que *back-end* e *front-end* são entidades completamente isoladas. Eles precisam se comunicar entre si para trocar dados e informações necessárias para o funcionamento da aplicação conforme a Figura 1. Essa comunicação geralmente é realizada por meio de APIs (Application Programming Interfaces), onde o *back-end* expõe *endpoints* que o front-end pode chamar para obter ou enviar dados. Em resumo, o *back-end* e o *front-end* são duas partes fundamentais de uma aplicação, cada uma com suas responsabilidades específicas. O *back-end* lida

com a lógica de negócios e processamento de dados, enquanto o *front-end* cuida da interface do usuário e interações visuais. Trabalhando em conjunto, eles formam uma aplicação *web* completa e funcional.

Figura 1 – Divisão entre *back-end* e *front-end* em uma aplicação *web*



Fonte: Arte/UT.

TEMA 1 – CONCEITUAR ROTAS HTTP – ROTAS ANGULAR

Rotas HTTP são caminhos definidos em uma aplicação *web* que permitem ao cliente (navegador, aplicativo móvel etc.) interagir com diferentes recursos ou funcionalidades do servidor. Essas rotas são especificadas usando o protocolo HTTP (*Hypertext Transfer Protocol*) e são usadas para acessar, criar, atualizar ou excluir dados em um aplicativo. No contexto do Angular, um *framework* JavaScript amplamente utilizado para o desenvolvimento de aplicações *web*, as rotas são usadas para navegar entre as diferentes visualizações ou componentes da aplicação. O Angular possui um módulo de roteamento embutido chamado *Angular Router*, que permite definir e gerenciar rotas no aplicativo.

As rotas no Angular são definidas no arquivo de configuração de rotas, geralmente chamado de "app-routing.module.ts". Nesse arquivo, você pode especificar os caminhos das rotas e associá-los aos componentes correspondentes. Por exemplo, você pode ter uma rota "/home", que está associada a um componente chamado "HomeComponent" e uma rota "/about" que está associada a um componente chamado "AboutComponent". Quando um

usuário navega para uma determinada rota, o Angular carrega o componente associado e o exibe na tela. As rotas no Angular também podem ter parâmetros, que são partes variáveis do caminho. Por exemplo, você pode ter uma rota "/user/:id" que representa a página de perfil de um usuário específico. O parâmetro ":id" permite que você acesse o ID do usuário na lógica do componente correspondente.

Além disso, o Angular Router também oferece recursos avançados, como roteamento aninhado (*nested routing*), roteamento com guarda de rotas (*route guards*) para proteger determinadas rotas e roteamento *lazy-loading*, que permite carregar módulos e componentes de forma assíncrona sob demanda. Em resumo, as rotas HTTP são caminhos definidos para acessar recursos em um servidor por meio do protocolo HTTP, enquanto as rotas no Angular são usadas para navegar entre os componentes e visualizações de uma aplicação web construída com o *framework* Angular.

1.1 Conceitos de Rotas no Angular2

Rotas HTTP são caminhos definidos em uma aplicação *web* que permitem ao cliente (navegador, aplicativo móvel etc.) interagir com diferentes recursos ou funcionalidades do servidor. Essas rotas são especificadas usando o protocolo HTTP (*Hypertext Transfer Protocol*) e são usadas para acessar, criar, atualizar ou excluir dados em um aplicativo.

No contexto do Angular, um *framework* JavaScript amplamente utilizado para o desenvolvimento de aplicações web, as rotas são usadas para navegar entre as diferentes visualizações ou componentes da aplicação. O Angular possui um módulo de roteamento embutido chamado *Angular Router*, que permite definir e gerenciar rotas no aplicativo.

As rotas no Angular são definidas no arquivo de configuração de rotas, geralmente chamado de "app-routing.module.ts". Nesse arquivo, você pode especificar os caminhos das rotas e associá-los aos componentes correspondentes. Por exemplo, você pode ter uma rota "/home" que está associada a um componente chamado "HomeComponent" e uma rota "/about" que está associada a um componente chamado "AboutComponent". Quando um usuário navega para uma determinada rota, o Angular carrega o componente associado e o exibe na tela.

Também podem ter parâmetros, que são partes variáveis do caminho. Por exemplo, você pode ter uma rota "/user/:id" que representa a página de perfil de um usuário específico. O parâmetro ":id" permite que você acesse o ID do usuário na lógica do componente correspondente. Também oferece recursos avançados, como roteamento aninhado (*nested routing*), roteamento com guarda de rotas (*route guards*) para proteger determinadas rotas e roteamento *lazy-loading*, que permite carregar módulos e componentes de forma assíncrona sob demanda.

Em resumo, as rotas HTTP são caminhos definidos para acessar recursos em um servidor por meio do protocolo HTTP, enquanto as rotas no Angular são usadas para navegar entre os componentes e visualizações de uma aplicação web construída com o *framework* Angular.

1.2 @routermodule no angular

O RouterModule é um módulo fundamental do Angular que oferece recursos e funcionalidades avançadas para o gerenciamento de rotas em aplicações *web*. Ele faz parte do pacote "@angular/router" e desempenha um papel essencial na definição e configuração das rotas em um projeto Angular. O RouterModule permite criar e configurar rotas dentro de um módulo específico da aplicação. Ele é importado no módulo principal da aplicação e permite que as rotas sejam definidas usando os métodos forRoot ou forChild, dependendo do escopo em que as rotas estão sendo configuradas. Para definir as rotas usando o RouterModule, você precisa importar as classes Routes e RouterModule do pacote "@angular/router". Em seguida, você pode definir as rotas em um *array* utilizando a sintaxe de objeto literal do TypeScript. Cada rota é um objeto que contém informações como o caminho (path) e o componente correspondente a ser renderizado quando a rota é ativada. Por exemplo, considere o seguinte trecho de código em TypeScript:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';

const routes: Routes = [
  { path: '', component: HomeComponent },
```

```
  { path: 'about', component: AboutComponent },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Saiba mais

Sobre o decorator @NgModule

Decorator, também conhecido como Wrapper (ou Decorador em português), é uma técnica que permite adicionar comportamentos extras a um objeto existente em tempo de execução. Ele permite a adição de responsabilidades adicionais a um objeto sem modificar sua estrutura original. Em vez de usar a herança para estender as funcionalidades de um objeto, o Decorator utiliza a composição, envolvendo o objeto original em um objeto decorador. O objeto decorador possui a mesma interface do objeto original, permitindo que ele seja tratado de forma transparente como o objeto original. Essa abordagem oferece uma alternativa flexível ao uso da herança, pois permite adicionar responsabilidades de maneira dinâmica e modular. Diferentes objetos decoradores podem ser combinados para adicionar diferentes comportamentos ao objeto original. Uma das vantagens do padrão Decorator é que ele permite adicionar ou remover comportamentos adicionais em tempo de execução, sem afetar outros objetos do mesmo tipo. Além disso, ele promove o princípio da responsabilidade única, adicionando responsabilidades ao objeto em vez de à classe.

Nesse exemplo, temos duas rotas definidas: uma rota padrão (*path* vazio) que renderiza o componente HomeComponent e uma rota com o caminho "about" que renderiza o componente AboutComponent. Essas rotas são configuradas no módulo principal da aplicação usando RouterModule.forRoot(routes). Além disso, o RouterModule permite configurações adicionais, como redirecionamentos e tratamento de rotas inexistentes (rota curinga). Por exemplo, é possível definir uma rota coringa que redireciona para uma página de erro personalizada quando uma rota não corresponde a nenhuma das rotas definidas anteriormente.

O RouterModule também oferece recursos avançados, como rotas aninhadas, *resolvers* de dados para carregamento prévio de informações, proteção de rotas com guardas de rota e muito mais. Esses recursos permitem uma navegação rica e dinâmica em aplicações Angular, garantindo uma experiência de usuário aprimorada. Em resumo, o RouterModule é uma parte essencial do Angular e fornece um conjunto robusto de recursos para o gerenciamento de rotas em aplicações *web*. Ele permite definir, configurar e controlar as rotas dentro de um módulo Angular, garantindo uma navegação eficiente e flexível entre os diferentes componentes da aplicação.

Abaixo segue exemplo de código o **app.routing.module.ts** do nosso projeto angular:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { UsuariosComponent } from
'./components/usuarios/usuarios.component';
import { QuestionarioComponent } from
'./components/questionario/questionario.component';
import { UsuarioComponent } from
'./components/usuario/usuario.component';
const routes: Routes = [
{path:"usuarios", component:UsuariosComponent},
{path:"questoes", component: QuestionarioComponent},
{path:"Usuario", component: UsuarioComponent},
{path:"Usuario/:id", component: UsuarioComponent},
];

@NgModule({
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule]
})
export class AppRoutingModule { }
```

Temos as linhas que importam as dependências necessárias do Angular para definir as rotas e importam os componentes **UsuariosComponent**, **QuestionarioComponent** e **UsuarioComponent** que serão utilizados como destinos de rota. A constante **routes** define as rotas do aplicativo. Cada objeto dentro do *array* representa uma rota, com duas propriedades principais: **path** e

component. A propriedade **path** define o caminho da URL que corresponderá a essa rota e a propriedade **component** especifica o componente a ser exibido quando essa rota for acessada. Por exemplo, quando a URL contém **/usuarios**, o componente **UsuariosComponent** será exibido. O decorador **NgModule** é usado para definir o módulo **AppRoutingModule**. Ele importa o módulo **RouterModule** e chama o método **forRoot(routes)** para configurar as rotas principais do aplicativo. Em seguida, exporta o **RouterModule** para que possa ser utilizado em outros módulos do aplicativo. Essa configuração de rotas permite que o aplicativo Angular navegue para os componentes **UsuariosComponent**, **QuestionarioComponent** e **UsuarioComponent** quando os caminhos correspondentes forem acessados na URL. Além disso, a rota **Usuario/:id** permite passar um parâmetro **id** para o componente **UsuarioComponent**.

TEMA 2 – CONSUMO DE DADOS DE BANCOS RELACIONAIS E NÃO RELACIONAIS

O consumo de dados em bancos relacionais e não relacionais pode variar dependendo de vários fatores, como a estrutura dos dados, a quantidade de registros, a complexidade das consultas e a eficiência da implementação do banco de dados. Bancos de dados relacionais, como o MySQL, PostgreSQL e Oracle, geralmente armazenam dados em tabelas com estruturas rígidas e relacionamentos definidos entre elas. O consumo de dados nesses bancos pode ser maior devido à necessidade de armazenar chaves estrangeiras e manter a consistência dos relacionamentos. Além disso, consultas complexas que envolvem várias tabelas podem exigir um processamento adicional de dados, o que pode aumentar o consumo de recursos.

Por outro lado, bancos de dados não relacionais, como MongoDB, Cassandra e Redis, são projetados para armazenar dados de forma mais flexível, usando formatos como documentos, colunas ou chave-valor. Esses bancos podem ter um consumo de dados menor, pois não exigem a definição prévia de esquemas rígidos. Além disso, eles são frequentemente otimizados para operações de leitura e gravação eficientes, o que pode resultar em um menor consumo de recursos. No entanto, é importante observar que o consumo de dados pode variar amplamente dependendo da implementação específica do banco de dados e da forma como os dados são modelados e consultados. Em

alguns casos, um banco de dados relacional bem otimizado pode consumir menos dados do que um banco de dados não relacional mal projetado.

Em resumo, não é possível afirmar categoricamente que um tipo de banco de dados consome mais ou menos dados do que o outro. O consumo de dados dependerá de vários fatores, incluindo o modelo de dados, a eficiência da implementação e a forma como os dados são consultados e manipulados.

2.1 Banco de dados relacionais

O conceito de banco de dados é a base para a organização e o armazenamento de informações em uma aplicação. Um banco de dados é uma coleção de dados inter-relacionados, que são estruturados e organizados de forma a representar informações sobre um domínio específico. Isso permite que os dados sejam armazenados, acessados e gerenciados de maneira eficiente. Um banco de dados relacional é um tipo de banco de dados que segue o modelo relacional. Nesse modelo, os dados são organizados em tabelas, em que cada tabela representa uma entidade ou um conjunto de dados relacionados. Cada tabela é composta por colunas, que representam os atributos ou características dos dados, e por linhas, que representam registros individuais. As tabelas são inter-relacionadas por meio de chaves primárias e chaves estrangeiras, que estabelecem relações entre os dados.

Para Date (1991), os Sistemas Gerenciadores de Banco de Dados Relacional (SGBDR) são *softwares* responsáveis por gerenciar e controlar o acesso aos dados armazenados em um banco de dados relacional. Eles fornecem uma interface para que os usuários possam criar, modificar e consultar os dados de forma segura e eficiente. Além disso, os SGBDRs oferecem recursos como transações, controle de concorrência, recuperação de falhas e otimização de consultas, visando garantir a consistência e a integridade dos dados. A linguagem SQL (*Structured Query Language*) é amplamente utilizada para interagir com bancos de dados relacionais. Ela permite aos usuários executarem consultas complexas para recuperar informações específicas, inserir novos dados, atualizar registros existentes e excluir dados indesejados. A SQL fornece uma sintaxe padronizada e poderosa para manipulação de dados, sendo amplamente suportada pelos SGBDRs.

No contexto do desenvolvimento de *software*, o uso de bancos de dados relacionais oferece vantagens como a capacidade de modelar e organizar os

dados de forma coerente, a flexibilidade para realizar consultas complexas e a interoperabilidade com diversas aplicações e sistemas. Em suma, um banco de dados relacional é uma ferramenta essencial no desenvolvimento de aplicações que exigem armazenamento e gerenciamento eficiente de informações. Os SGBDRs e a linguagem SQL permitem que os desenvolvedores criem aplicações robustas e escaláveis, que podem manipular grandes volumes de dados e fornecer acesso rápido e seguro às informações necessárias.

2.2 Banco de dados Não Relacionais ou NOSQL

Segundo Sadalage e Fowler (2012), os bancos não relacionais surgiram pela necessidade de lidar com grandes volumes de dados. Os bancos de dados relacionais tradicionais enfrentavam desafios em termos de escalabilidade e desempenho quando lidavam com quantidades massivas de dados. Os defensores dos bancos de dados NoSQL argumentam que eles são mais flexíveis do que os bancos de dados relacionais. Isso ocorre porque os bancos de dados NoSQL não possuem esquemas rígidos, o que significa que não é necessário definir previamente a estrutura dos dados. Isso proporciona uma maior agilidade no desenvolvimento de aplicativos, uma vez que as alterações na estrutura dos dados podem ser feitas de forma mais flexível e dinâmica.

Os bancos de dados NoSQL ou não relacionais oferecem uma abordagem diferente para o armazenamento e gerenciamento de dados em comparação com os bancos de dados relacionais. Um equívoco comum é pensar que os bancos de dados NoSQL não podem armazenar dados relacionados, mas na verdade eles podem fazer isso de maneira diferente. Os bancos de dados NoSQL permitem armazenar dados de relacionamento em uma única estrutura de dados, em vez de dividir os dados entre tabelas, como nos bancos de dados relacionais. Essa flexibilidade permite que os dados relacionados sejam armazenados de forma mais simples e sem a necessidade de definir a estrutura dos dados previamente. Em uma única tabela de um banco de dados NoSQL, é possível ter dados com propriedades diferentes. Esses bancos de dados surgiram no final dos anos 2000, à medida que o custo de armazenamento diminuiu drasticamente. Eles oferecem uma alternativa aos modelos de dados complexos dos bancos de dados relacionais, especialmente quando se trata de evitar a duplicação de dados.

Existem diferentes tipos de bancos de dados NoSQL, categorizados de acordo com a maneira como armazenam os dados. Os dois tipos mais comuns são:

- Banco de documentos: esses bancos de dados armazenam os dados em documentos semelhantes a objetos JSON (*JavaScript Object Notation*). Eles geralmente possuem linguagens de consulta poderosas e são adequados para uso geral. Um exemplo popular de banco de dados de documentos é o MongoDB;
- Chave-valor: esses bancos de dados armazenam os dados como pares de chave-valor. Os valores podem ser de qualquer tipo de dado e são recuperados por meio da referência à sua chave. Esse tipo de banco de dados é eficiente para armazenar grandes quantidades de dados, mas não é adequado para consultas complexas. Redis e DynamoDB são exemplos populares de bancos de dados chave-valor.

Os bancos de dados NoSQL oferecem vantagens, como armazenamento de grandes volumes de dados sem uma estrutura definida, uso de computação e armazenamento em nuvem e desenvolvimento rápido. Eles são adequados para cenários em que a flexibilidade e a escalabilidade são prioritárias, como desenvolvimento ágil e aplicativos com necessidades de armazenamento variáveis.

Exemplos de bancos de dados NoSQL:

- Redis: banco de dados de chave-valor muito utilizado devido à sua alta velocidade e recursos avançados, como armazenamento em memória, cache distribuído e suporte a estruturas de dados complexas. É amplamente utilizado em cenários que exigem alto desempenho e baixa latência, como cache de dados, filas de mensagens e contagem de visualizações;
- Memcached: Também é um banco de dados NoSQL de chave-valor baseado em cache de memória distribuída. É utilizado para acelerar o acesso a dados frequentemente acessados, reduzindo a carga em fontes externas, como bancos de dados tradicionais. É comumente utilizado em ambientes web para melhorar o desempenho de *sites* dinâmicos;
- Cassandra: desenvolvido pelo Facebook, o Cassandra é um banco de dados NoSQL distribuído, altamente escalável e tolerante a falhas. É

otimizado para operar em *clusters* e oferece baixa latência em atualizações. É amplamente utilizado em casos de uso que exigem escalabilidade horizontal e tolerância a falhas, como armazenamento de dados em grande escala, análise de dados e aplicativos de tempo real;

- HBase: orientado a colunas baseado no modelo de dados Bigtable do Google. Ele fornece armazenamento de dados em larga escala e é usado por empresas como LinkedIn, Facebook e Spotify para lidar com volumes massivos de dados estruturados. É especialmente adequado para casos de uso que envolvem análise de dados e recuperação rápida de informações;
- Amazon DynamoDB: oferecido pela Amazon Web Services (AWS) como serviço em nuvem. Possui baixa latência, alta escalabilidade automática e flexibilidade. É amplamente utilizado em aplicativos móveis, jogos na web e soluções de Internet das Coisas (IoT), onde a escalabilidade e o desempenho são essenciais;
- Neo4j: baseado em grafos, onde os dados são representados por meio de nós (nodes) e relacionamentos (arestas). É amplamente utilizado em casos de uso que envolvem análise de redes complexas, mineração de dados, recomendações personalizadas e reconhecimento de padrões.
- MongoDB: orientado a documentos que armazena informações em documentos semiestruturados (como JSON). Possui alta *performance*, suporte a consultas flexíveis e escalabilidade horizontal. É utilizado por várias empresas e organizações para uma ampla gama de aplicativos, desde *sites* de alto tráfego até soluções de *big data*.

2.3 CRUD

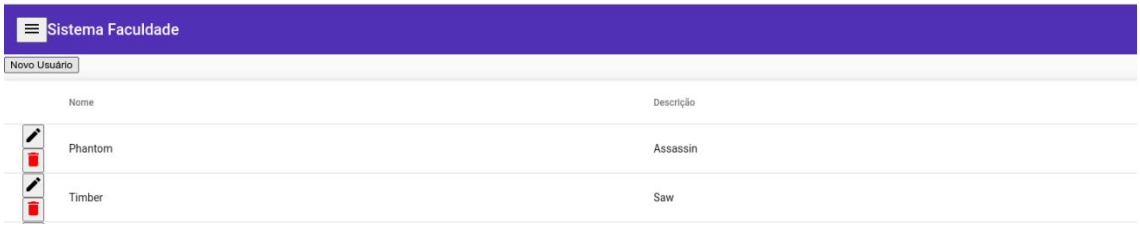
CRUD (*Create, Read, Update* e *Delete*) é um acrônimo amplamente utilizado para descrever as quatro operações básicas que podem ser realizadas em um banco de dados relacional ou em qualquer sistema de armazenamento persistente de dados (Arend, 2011). Essas operações permitem criar novos registros, recuperar informações existentes, atualizar registros existentes e excluir registros indesejados.

Aqui está uma breve descrição de cada uma das operações do CRUD:

- Create (Criar): a operação de criação envolve a inserção de novos registros ou dados no banco de dados. Isso geralmente é feito por meio de um formulário ou interface de usuário, onde o usuário fornece os dados necessários e eles são salvos no banco de dados como um novo registro;
- Read (Ler): a operação de leitura refere-se à obtenção de informações ou registros existentes do banco de dados. Isso pode ser feito por meio de consultas ou filtros, permitindo que o usuário obtenha os dados desejados com base em critérios específicos, como um ID único ou determinadas condições;
- Update (Atualizar): a operação de atualização envolve a modificação de registros existentes no banco de dados. Isso permite que o usuário faça alterações nos dados salvos, como atualizar um endereço, modificar informações pessoais ou qualquer outra alteração necessária;
- Delete (Excluir): a operação de exclusão é usada para remover registros indesejados ou desnecessários do banco de dados. Isso permite que o usuário remova informações que não são mais relevantes ou que não devem ser mantidas no sistema.

Essas operações do CRUD são essenciais para a manipulação de dados em sistemas de informação. A maioria das aplicações *web* que interagem com um banco de dados executará essas operações para fornecer funcionalidades completas aos usuários. Automatizar essas operações por meio de frameworks e bibliotecas de desenvolvimento ajuda a acelerar o processo de desenvolvimento, fornecendo uma maneira padronizada e eficiente de trabalhar com os dados (Pereira; Cogo; Charao, 2009).

Figura 2 – Imagem mostrando o sistema exemplo em funcionamento, onde o lápis refere-se à edição, o lixo vermelho exclusão, o botão Novo Usuário a criação de um novo usuário e pôr fim a listagem é a própria tela.



Fonte: Ferste, 2023.

TEMA 3 – CONCEITO DE API – CONSUMO DE APIS

API (*Application Programming Interface*) é um conjunto de regras e protocolos que permite a comunicação e interação entre diferentes *softwares* e sistemas. Ela define os métodos de comunicação, formatos de dados e regras de acesso que permitem que aplicativos se comuniquem entre si de maneira padronizada. São projetadas para expor determinadas funcionalidades ou dados de um sistema de *software* de forma controlada e segura, permitindo que outros aplicativos possam utilizá-los de maneira programática. Elas atuam como ponte entre diferentes componentes de *software*, permitindo que aplicativos se comuniquem e compartilhem informações.

As APIs podem ser usadas para diferentes propósitos, tais como:

- Integração de sistemas: uma API permite que diferentes sistemas ou aplicativos se comuniquem e compartilhem informações entre si. Por exemplo, um aplicativo de comércio eletrônico pode usar a API de um sistema de pagamento para processar transações;
- Desenvolvimento de aplicativos: as APIs podem fornecer funcionalidades específicas que podem ser utilizadas por desenvolvedores para criar novos aplicativos. Por exemplo, uma API de mapas pode permitir que os desenvolvedores incorporem recursos de localização em seus aplicativos;
- Acesso a dados: muitas APIs fornecem acesso a conjuntos de dados específicos, permitindo que os desenvolvedores obtenham informações atualizadas de fontes confiáveis. Por exemplo, uma API de clima pode fornecer informações meteorológicas atualizadas para aplicativos de previsão do tempo;
- Automação: as APIs também são usadas para automatizar tarefas e processos. Elas permitem que aplicativos interajam com outros sistemas de forma programática, sem intervenção humana direta. Por exemplo, uma API de envio de mensagens pode ser usada para automatizar o envio de notificações por *e-mail* ou mensagens de texto.

As APIs podem ser baseadas em diferentes tecnologias e protocolos, como REST (*Representational State Transfer*), SOAP (*Simple Object Access Protocol*) e GraphQL. Elas geralmente possuem documentação detalhada que descreve os *endpoints* disponíveis, os parâmetros aceitos, os formatos de dados

esperados e outras informações relevantes para facilitar o uso e a integração com os sistemas que as disponibilizam.

3.1 API Web

No contexto do desenvolvimento web, uma API é frequentemente exposta por um servidor *web* para permitir que aplicativos clientes possam interagir com o sistema e obter informações ou executar determinadas ações (W3C Arquitetura Web, 2023).

Existem diferentes estilos arquiteturais para a construção de APIs, sendo dois deles bastante comuns:

- SOAP (*Simple Object Access Protocol*): é um protocolo baseado em XML que define uma estrutura de mensagens e regras para comunicação entre sistemas distribuídos. O SOAP é mais orientado a serviços e geralmente usa o protocolo HTTP como transporte, embora também possa ser usado em outros protocolos. No entanto, a abordagem SOAP tem sido menos adotada na Web 2.0 devido à sua complexidade e sobrecarga;
- REST (*Representational State Transfer*): é um estilo arquitetural baseado em princípios e restrições que definem como os recursos de um sistema devem ser expostos e acessados. No REST, os recursos são identificados por URLs (*Uniform Resource Locators*) e são manipulados através de operações padrão do protocolo HTTP, como GET, POST, PUT e DELETE. Além disso, o REST utiliza formatos de dados mais leves e amplamente adotados, como XML ou JSON, para representar as mensagens de requisição e resposta.

A abordagem REST se tornou amplamente adotada na Web 2.0 devido à sua simplicidade, flexibilidade e suporte nativo aos recursos da web. Ela permite que as APIs sejam mais leves, escaláveis e fáceis de usar em diferentes plataformas e linguagens de programação.

É importante destacar que, embora o uso do formato JSON seja comum em APIs REST, ele não é exclusivo. O XML também é amplamente utilizado, especialmente em APIs legadas ou em sistemas que têm requisitos específicos para interoperabilidade com outras tecnologias. A abordagem REST, comumente usada na Web 2.0, oferece uma maneira simplificada e eficiente de

criar APIs, utilizando o protocolo HTTP e formatos de dados leves como XML ou JSON.

Saiba mais

Relembrando...uma boa forma de debugar código na internet é utilizando o comando console.

```
console.log('Hello, world!');
```

Com ele pode-se verificar valores em tempo de execução.

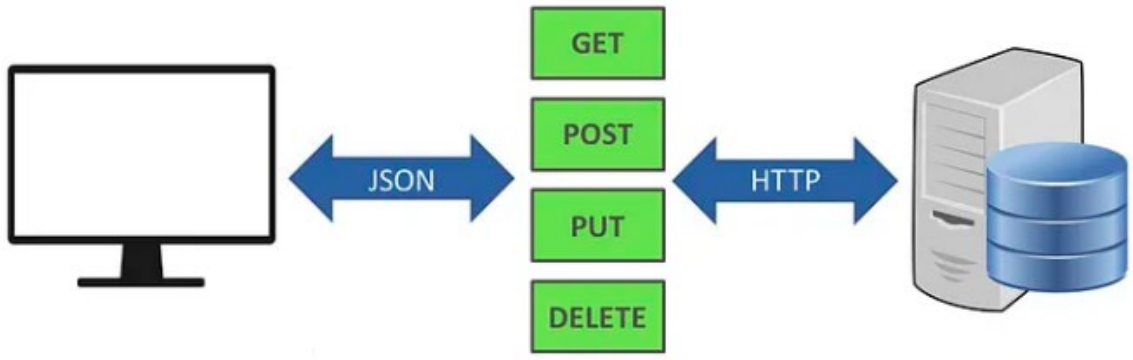
3.2 JSON

Uma das principais características do formato JSON é sua linguagem independente. Isso significa que ele pode ser utilizado em qualquer linguagem de programação, tornando-o amplamente compatível e interoperável entre diferentes sistemas e plataformas. Independentemente da linguagem de programação que você esteja usando para desenvolver sua aplicação, é possível ler, gravar e manipular dados no formato JSON de maneira fácil e eficiente. Isso torna o JSON uma escolha popular para a transferência e armazenamento de dados em diversas aplicações.

Além disso, o JSON pode ser utilizado em várias plataformas, como Windows, macOS e Linux, e em diferentes tipos de sistemas, incluindo aplicações web e aplicativos móveis. Essa flexibilidade é essencial para garantir a interoperabilidade e a comunicação eficiente entre sistemas heterogêneos. A simplicidade e a legibilidade do formato JSON também são características importantes. A estrutura dos dados no JSON é intuitiva e fácil de entender, o que facilita a criação, leitura e manutenção dos arquivos JSON. Os dados são organizados em pares chave-valor, delimitados por chaves {}, e os elementos em um *array* são delimitados por colchetes []. Isso torna o JSON amigável tanto para humanos quanto para máquinas.

Em resumo, as principais características do formato JSON são sua linguagem independente, simplicidade, legibilidade e compatibilidade com diferentes plataformas e sistemas. Essas características contribuem para sua ampla adoção e popularidade na transferência e armazenamento de dados em aplicações modernas.

Figura 3 – Demonstração sobre o processo de acesso à informação básico em um acesso na *web* usando JSON



Fonte: Ferste, 2023.

O formato JSON segue uma notação específica para organizar os dados, conforme mencionado. Aqui estão alguns exemplos adicionais para ilustrar a estrutura do JSON:

Objeto com vários pares de chave-valor:

```
{
  "nome": "João",
  "idade": 30,
  "email": "joao@example.com"
}
```

Objeto contendo um *array*:

```
{
  "pessoas": [
    {"nome": "João", "idade": 30},
    {"nome": "Maria", "idade": 25},
    {"nome": "Pedro", "idade": 35}
  ]
}
```

Objeto contendo um *array*:

```
Objeto contendo um array:

{
  "pessoas": [
    {"nome": "João", "idade": 30},
    {"nome": "Maria", "idade": 25},
    {"nome": "Pedro", "idade": 35}
```

```
  ]
}
```

Objeto aninhado com vários níveis de profundidade:

```
{
  "usuario": {
    "nome": "João",
    "endereco": {
      "rua": "Rua A",
      "cidade": "São Paulo",
      "estado": "SP"
    }
  }
}
```

Array contendo objetos:

```
[  {"nome": "João", "idade": 30},  {"nome": "Maria", "idade": 25},  {"nome": "Pedro", "idade": 35}]
```

Essa estrutura permite uma representação hierárquica e organizada dos dados, facilitando a leitura, escrita e interpretação por parte das aplicações que consomem os dados.

TEMA 4 – CONSUMO DE DADOS DO BACK-END

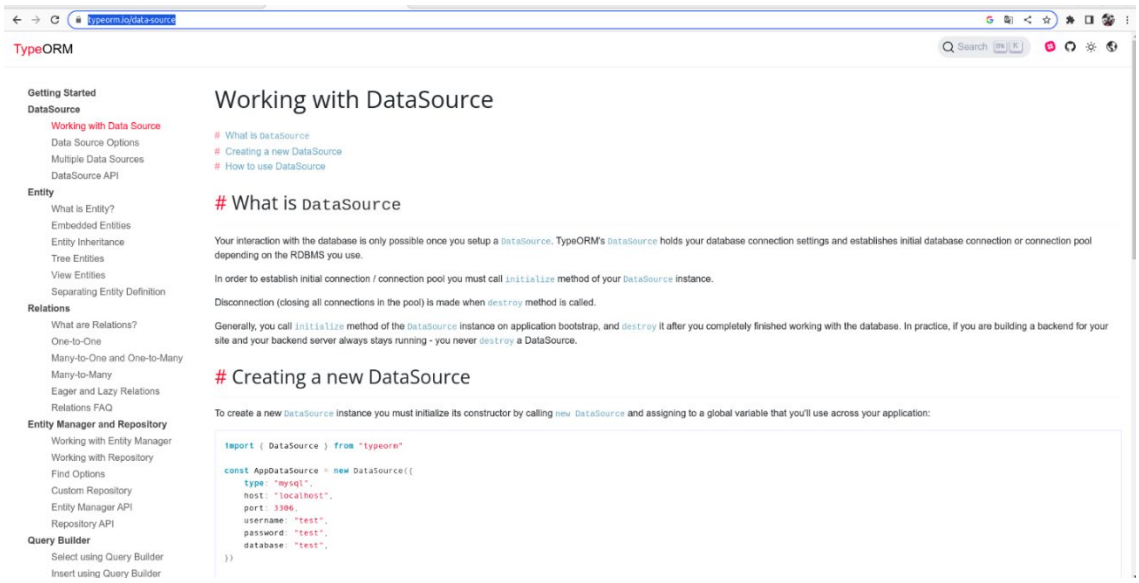
Para tratarmos do consumo de dados do *back-end* temos de ter um *back-end*, concorda? Neste sentido faremos agora um processo de construção do nosso *back-end* com ferramentas existentes e baseadas em node. Neste tópico, veremos mapeamento de banco, veremos como construir basicamente um projeto e expor os dados, para então consumir os mesmos.

4.1 Mapeamento de banco TypeOrm

Embora não seja objetivo do nosso estudo, vamos entender neste tópico como é montado um acesso a banco, basicamente falando, para isto utilizaremos a biblioteca de mapeamento objeto-relacional (ORM) escrita em TypeScript para Node.js e TypeScript (também pode ser usada em projetos JavaScript). Ela permite aos desenvolvedores mapear objetos JavaScript para tabelas de banco de dados e vice-versa, simplificando o processo de persistência de dados. Com o TypeORM, é possível definir entidades (classes JavaScript/TypeScript) que representam tabelas no banco de dados. Essas entidades são anotadas com

decoradores para definir suas propriedades, relacionamentos e configurações de mapeamento. O TypeORM utiliza essas informações para gerar automaticamente as consultas SQL necessárias para interagir com o banco de dados.

Figura 4 – Tela mostrando o sitio do Typeorm



Fonte: <https://typeorm.io/data-source>.

Saiba mais

Mostraremos alguns comandos sobre o *back-end*, para efeito de entendimento, mas o código vai normalmente ser instalado e executado quando pronto com npm i e npm start, simples assim.

Entretanto, para alguns entendimentos seguem mais comandos interessantes.

```
npm install -g ts-node
npm install -g typeorm
```

Os comandos npm install -g ts-node e npm install -g typeorm são usados para instalar globalmente o pacote ts-node e o pacote typeorm, respectivamente. O pacote ts-node permite que você execute arquivos TypeScript diretamente no Node.js, sem a necessidade de compilar primeiro para JavaScript. Ele é útil durante o desenvolvimento para facilitar a execução e depuração de código TypeScript.

```
typeorm init --name api --express --database pg
 typeorm init --name apiQuestionario --express --database mysql
```

O comando typeorm init --name api --express --database pg cria um novo projeto chamado "api" utilizando o Express como *framework web* e o PostgreSQL como banco de dados. Ele irá gerar uma estrutura básica de arquivos e configurações para você começar a desenvolver sua API.

O comando typeorm init --name apiQuestionario --express --database mysql é semelhante ao anterior, porém cria um novo projeto chamado "apiQuestionario" utilizando o MySQL como banco de dados. Ele também gera a estrutura inicial de arquivos e configurações necessárias para desenvolver a API.

Quando o projeto foi criado então ele pode ser utilizado, a mensagem é similar à mostrada a seguir:

```
Project created inside /projetoweb/api directory.
Please wait, installing dependencies...
Done! Start playing with a new project!
```

Seguem mais alguns comandos:

```
npm install pg --save
```

O comando npm install pg --save é utilizado para instalar o pacote pg no seu projeto Node.js e adicioná-lo como uma dependência no arquivo package.json. O pacote pg é um driver para o PostgreSQL, permitindo que você se conecte e interaja com um banco de dados PostgreSQL em sua aplicação. Ao executar esse comando, o npm irá baixar e instalar o pacote pg no diretório do seu projeto, além de atualizar o arquivo package.json com a nova dependência. O parâmetro --save indica que a dependência deve ser adicionada à seção de dependências do package.json, para que ela seja instalada novamente automaticamente em outros ambientes. Certifique-se de executar esse comando no diretório raiz do seu projeto, onde está localizado o arquivo package.json.

```
typeorm entity:create -n User
```

O comando typeorm entity:create -n User é utilizado para criar uma nova entidade chamada "User" usando o TypeORM. O TypeORM é um ORM (Object-Relational Mapping) para Node.js que permite mapear objetos JavaScript para tabelas em um banco de dados relacional. Ao executar esse comando, o

TypeORM criará automaticamente os arquivos necessários para a entidade "User", como um arquivo de modelo (*model*), um arquivo de repositório (repository) e um arquivo de migração (*migration*), se estiver configurado para isso. Certifique-se de ter o TypeORM instalado e configurado corretamente em seu projeto antes de executar esse comando. Além disso, verifique se você está no diretório raiz do seu projeto, onde o arquivo typeorm.json está localizado. Isso garantirá que o comando seja executado no contexto correto.

4.2 Exemplo de *back-end* em execução

O servidor está sendo executado na porta 3000 e você pode acessar os resultados abrindo http://localhost:3000/users no seu navegador.

```
api>> npm start

> api@0.0.1 start
> ts-node src/index.ts

Express server has started on port 3000. Open http://localhost:3000/users to see results
api>> npm start
```

Essa mensagem indica que os servidores foram iniciados corretamente e estão prontos para receber solicitações. Agora você pode começar a enviar requisições para os *endpoints* definidos em seus projetos e ver os resultados no navegador. Inicialmente todo projeto usando o Express tem, pelo menos o endpoint http://localhost:3000/users, para um teste inicial, conforme a mensagem.

TEMA 5 – CONCEITUAR REST -> GET, POST, PUT, DELETE...

REST (*Representational State Transfer*) é um estilo de arquitetura de software amplamente utilizado para projetar sistemas distribuídos, especialmente na construção de APIs (*Application Programming Interfaces*) da web. Ele define um conjunto de princípios e restrições que visam facilitar a comunicação e interação entre diferentes componentes de um sistema distribuído (W3C, 2023).

Os principais princípios e conceitos do REST são:

- Recursos (*Resources*): um recurso é uma entidade identificável que pode ser acessada e manipulada por meio de uma URI (*Uniform Resource Identifier*). Exemplos de recursos podem ser usuários, posts, produtos etc.;
- Representações (*Representations*): os recursos podem ser representados em diferentes formatos, como JSON, XML, HTML, entre outros. A escolha do formato é determinada pelas necessidades da aplicação e pelas preferências do cliente;
- *Stateless* (Sem estado): o REST é baseado no princípio de que cada requisição do cliente contém todas as informações necessárias para ser processada pelo servidor. O servidor não mantém o estado da sessão entre as requisições, o que facilita a escalabilidade e a confiabilidade;
- URI (*Uniform Resource Identifier*): as URIs são usadas para identificar exclusivamente os recursos. Uma URI bem projetada deve ser descritiva, significativa e seguir as melhores práticas de nomenclatura;
- Respostas HTTP (HTTP Responses): o servidor responde às requisições REST com códigos de status HTTP, como 200 OK, 201 Created, 404 Not Found, entre outros, para indicar o resultado da operação.

O REST permite a criação de APIs simples, flexíveis, escaláveis e interoperáveis, tornando-se uma abordagem amplamente adotada para a construção de serviços web. Ao seguir os princípios REST, é possível criar uma arquitetura de software robusta, de fácil manutenção e com boa aderência aos padrões da *web* (W3C, 2023).

Saiba mais

Uma comparação entre implementações e diferentes tipos de serviços pode ser encontrada em:

ATHOS, E.; CORDEIRO, D.; SOUZA, R. de. Web Services: integração de sistemas orientado a serviços com uma proposta de aplicação na EAD. **Revista de Informática Aplicada**, v. 12, n. 2, 2016. Disponível em: <https://seer.uscs.edu.br/index.php/revista_informatica_aplicada/article/download/6918/3009/21112>. Acesso em: 31 ago. 2023.

5.1 Conceituando os verbos HTTP

Os verbos HTTP são métodos que especificam a ação a ser realizada em um determinado recurso em uma arquitetura REST. Eles indicam a intenção do cliente em relação ao recurso solicitado e permitem que as APIs sejam projetadas para fornecer operações CRUD (*Create, Read, Update, Delete*) consistentes e semânticas (W3 HTTP, 2023).

Aqui estão os principais verbos HTTP e suas definições:

- GET: O método GET é usado para recuperar informações de um recurso específico. Ele solicita a representação do recurso e não deve ter efeitos colaterais no servidor;
- POST: O método POST é usado para enviar dados para o servidor para criar um novo recurso. Geralmente é usado para submeter formulários, enviar dados de *login* etc. Pode ter efeitos colaterais no servidor, como a criação de um novo registro no banco de dados;
- PUT: O método PUT é usado para atualizar um recurso existente no servidor. Ele envia uma representação completa do recurso com as alterações desejadas. Se o recurso não existir, ele pode ser criado;
- DELETE: O método DELETE é usado para excluir um recurso específico do servidor;
- PATCH: O método PATCH é usado para aplicar modificações parciais a um recurso. Ele envia apenas as alterações específicas a serem aplicadas ao recurso.

Além desses, existem outros termos menos comuns, como *options, head, trace* e *connect*, que têm usos mais especializados em casos específicos (W3 HTTP, 2023). É importante observar que os verbos HTTP são apenas uma convenção e seu significado pode variar dependendo da implementação da API. No entanto, seguindo as diretrizes e práticas recomendadas, é possível criar APIs consistentes e compreensíveis (W3 HTTP, 2023).

5.2 Retornos de HTTP

Os verbos HTTP são métodos que especificam a ação a ser realizada em um determinado recurso em uma arquitetura RESTful. Eles indicam a intenção do cliente em relação ao recurso solicitado e permitem que as APIs sejam

projetadas para fornecer operações CRUD (*Create, Read, Update, Delete*) consistentes e semânticas. Para tratar estes retornos temos de abstrair a forma de retorno e pensar no conteúdo. O que vai explodir do lado do chamador? O que vai ser retornado para nós?

Aqui está uma lista resumida dos códigos de status HTTP mais comuns, entretanto não se deve considerar decorar a tabela abaixo, mesmo não sendo ela mutável, é melhor aprender a mesma por uso.

- 1xx (Informational): Indica que a solicitação foi recebida e o servidor está processando-a;
- 100 Continue;
- 101 Switching Protocols;
- 2xx (*Successful*): Indica que a solicitação foi recebida, compreendida e aceita com sucesso;
- 200 OK;
- 201 Created;
- 204 No Content;
- 3xx (Redirection): Indica que a solicitação precisa ser redirecionada para outro local;
- 301 Moved Permanently;
- 302 Found;
- 304 Not Modified;
- 4xx (*Client Error*): Indica que ocorreu um erro por parte do cliente ao fazer a solicitação;
- 400 Bad Request;
- 401 Unauthorized;
- 404 Not Found;
- 5xx (*Server Error*): Indica que ocorreu um erro no servidor ao processar a solicitação;
- 500 Internal Server Error;
- 503 Service Unavailable;
- 504 Gateway Timeout.

Essa é apenas uma amostra dos códigos de *status* HTTP existentes. Há muitos outros códigos que podem ser encontrados, cada um com seu próprio significado específico. Ao desenvolver aplicativos *web* ou lidar com solicitações

HTTP, é útil consultar a lista completa de códigos de *status* HTTP para obter informações detalhadas sobre cada código.

5.3 Um exemplo mínimo de uma API

Esse código básico cria uma API simples com algumas rotas definidas e faz a API escutar na porta 3000. Você pode adicionar mais rotas e lógica de negócio conforme suas necessidades.

```
const express = require("express")
const app = express();

app.listen(3000,()=> {

})
const express = require("express")
const app = express();
const bodyParser = require('body-parser')

app.use(bodyParser())
app.listen(3000,()=> {
  console.log('api inicializada')
})

app.get('/',(req,res)=> {
    res.status(200).send("olá mundo")
})

app.post('/', (req,res)=> {
    res.status(201).send(req.body)
})

app.put ('/:id', (req,res)=> {
    res.status(202).send({
       cod: req.params.id,
       corpo:req.body
    })
})
```

```
app.delete ('/:id',(req,res)=> {
    res.status(204).send (req.params.id)
})
```

Importação dos módulos: o código começa importando os módulos necessários para a construção da API. O *express* é o módulo principal do *framework* Express.js, responsável por criar o servidor HTTP. O *body-parser* é um *middleware* que facilita o processamento dos dados enviados no corpo das requisições.

Criação da instância do aplicativo Express: O código cria uma instância do aplicativo Express usando a função express(). Essa instância representa a nossa API.

Configuração do middleware body-parser: O código utiliza o *middleware body-parser* para facilitar o processamento dos dados enviados no corpo das requisições. A linha app.use(bodyParser()) adiciona o *middleware* ao *pipeline* de execução da API.

Configuração das rotas: O código define algumas rotas para a API usando os métodos HTTP, como GET, POST, PUT e DELETE. Cada rota é associada a uma função de *callback* que é executada quando a rota é acessada.

A rota '/' é uma rota GET que retorna a mensagem "olá mundo" com o status 200 (OK).

A rota '/' também é uma rota POST que retorna o corpo da requisição como resposta com o *status* 201 (*Created*).

A rota '/id' é uma rota PUT que retorna um objeto com o código do parâmetro de rota e o corpo da requisição como resposta com o status 202 (*Accepted*).

A rota '/id' é uma rota *Delete* que retorna o código do parâmetro de rota como resposta com o status 204 (*No Content*).

Inicialização do servidor: O código utiliza o método listen() para iniciar o servidor HTTP e fazer a API escutar na porta 3000. Quando o servidor é iniciado, a mensagem "api inicializada" é exibida no console.

5.4 Algumas APIS para teste

Para consumir APIs (Interfaces de Programação de Aplicativos), existem várias ferramentas disponíveis para testar e consumir dados. Algumas das mais populares são:

- **Postman**: é uma ferramenta de desenvolvimento de API completa que permite enviar solicitações HTTP, testar *endpoints*, visualizar e analisar respostas, além de fornecer recursos avançados de automação e documentação de API;
- **Insomnia**: é uma alternativa ao Postman, oferecendo recursos semelhantes para teste e consumo de APIs. Ele fornece uma interface amigável para criar e enviar solicitações, visualizar respostas e gerenciar várias coleções de API;
- **cURL**: é uma ferramenta de linha de comando amplamente utilizada para enviar solicitações HTTP e visualizar as respostas correspondentes. É uma opção popular para testes rápidos e simples de API;
- **Paw**: é uma ferramenta avançada para testar e consumir APIs, com recursos como geração automática de código, testes automatizados, ambiente de execução de *scripts* e muito mais;
- **SoapUI**: embora seja focado principalmente em testes de serviços SOAP, o SoapUI também suporta testes de APIs RESTful. Ele oferece recursos abrangentes para criar, testar e simular solicitações HTTP, além de recursos avançados de geração de relatórios e automação de testes.

Essas são apenas algumas das opções disponíveis, e cada uma delas tem seus próprios recursos e vantagens. A escolha da ferramenta depende das suas necessidades específicas e preferências pessoais.

5.5 Algumas dicas sobre consumo de APIS

Lembre-se de que, embora esses códigos de status forneçam informações gerais sobre o resultado de uma solicitação, é importante considerar o contexto específico em que são usados e entender as necessidades do seu aplicativo para lidar com eles adequadamente.

Pense na forma de consumir estes retornos, não é complicado precisamos utilizar os programas mencionados anteriormente para ler os retornos.

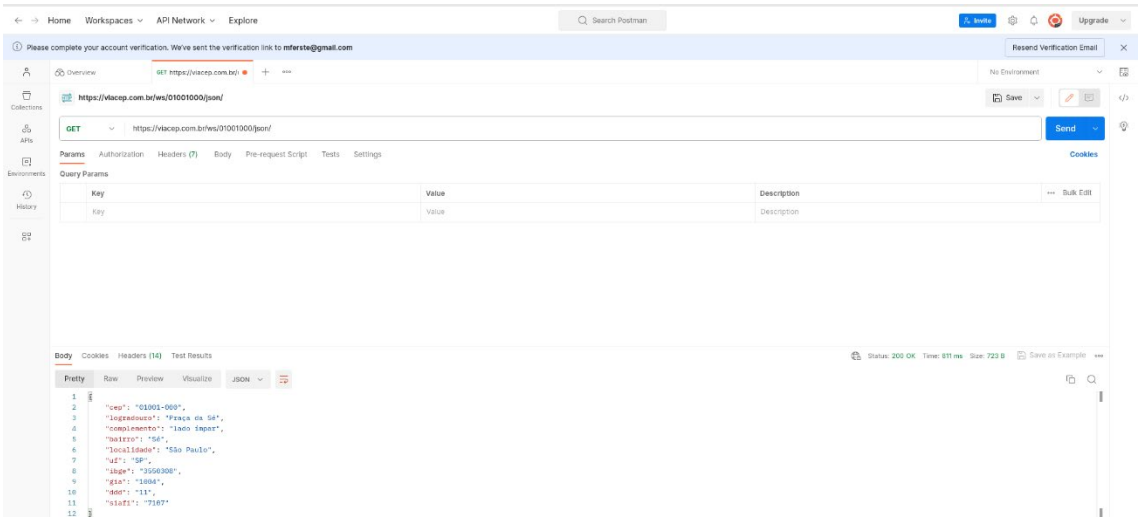
Saiba mais

Por exemplo, na tela abaixo temos o consumo dos serviços de um site muito bom para trabalho, chamado, ViaCEP, do IBGE. Ao entrar no *site* ViaCep, este sugere um teste com um CEP genérico.

VIACEP. Disponível em: <https://viacep.com.br/>. Acesso em: 31 ago. 2023.

Tente utilizar em um navegador, o endereço do ViaCEP sugerido... https://viacep.com.br/ws/01001000/json/

Ele vai descarregar no *browser* o conteúdo, entretanto se queremos tratar com qualidade o conteúdo devemos buscar um programa para tratamento, como Postman ou ViaCEP, como abaixo.



Neste caso dizemos que estamos “consumindo” o *end-point* com um comando GET.

É extremamente necessário aprender a utilizar programas para acessar os serviços *rest*, pois normalmente, em uma visão moderna o desenvolvimento do front é feito em separado do *back*.

FINALIZANDO

A Web, ou World Wide Web, revolucionou a forma como interagimos e compartilhamos informações. Ela é baseada em tecnologias como o protocolo

HTTP (Protocolo de Transferência de Hipertexto), que permite a transferência de dados entre um cliente e um servidor.

Um formato amplamente utilizado na Web para troca de dados é o JSON (*JavaScript Object Notation*). O JSON é um formato leve e legível que utiliza uma estrutura de pares de chave-valor para representar informações. Ele suporta tipos de dados comuns, como *strings*, números, booleanos, objetos e *arrays*. A integração de sistemas desempenha um papel fundamental na Web e em muitas aplicações modernas. Ela envolve a conexão e a comunicação entre diferentes sistemas de *software*. Para facilitar essa comunicação, são usadas APIs (*Application Programming Interfaces*), que definem interfaces e *endpoints* que os sistemas podem usar para interagir uns com os outros. Isso possibilita a interoperabilidade entre diferentes sistemas e facilita a automação de processos.

As APIs desempenham um papel fundamental na integração de sistemas, pois fornecem pontos de acesso para que os sistemas possam se comunicar. Essas APIs podem usar o formato JSON para estruturar os dados enviados e recebidos, garantindo que as informações sejam compreendidas e processadas corretamente.

Em resumo, a Web, o JSON, o HTTP e a integração de sistemas estão interligados e impulsionam a comunicação e a troca de dados entre sistemas de software. Essas tecnologias permitem a construção de aplicativos e serviços robustos, escaláveis e interoperáveis, facilitando a criação de soluções integradas e eficientes para as necessidades atuais da sociedade.

REFERÊNCIAS

ANGULAR. Disponível em: <https://www.angular.io>. Acesso em: 31 ago. 2023.

APACHE; Disponível em: <https://struts.apache.org/>. Acesso em 31 ago. 2023.

AREND, F. G. **Geração de Operações CRUD a partir de metadados**. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Centro de Tecnologia, Universidade Federal de Santa Maria, Santa Maria, 2011.

BALSAMIQ; Disponível em: <https://balsamiq.com/>. Acesso em: 31 ago. 2023.

GOULÃO, M. **Component-Based software engineering**: a quantitative approach. Lisboa: Universidade Nova de Lisboa, 2008.

INSOMNIA; Disponível em: <https://insomnia.rest/>. Acesso em: 31 aqgo. 2023.

LARAVEL; Disponível em: <https://laravel.com/>. Acesso em: 31 ago. 2023.

NORMAN, D. A. **Design for a better world**: meaningful, sustainable, humanity centered. Cambridge, Massachusetts: The MIT Press. 2023.

NPM. Node Package Manager, Disponível em: <https://www.npmjs.com/>. Acesso em: 31 ago. 2023.

PEREIRA, A.; COGO, V. V.; CHARAO, A. S. *Framework* para desenvolvimento rápido de aplicações web: um estudo de caso com CakePHP e Django. **Programa de Educação Tutorial (PET)**, p. 1-6, 2009.

PRESSMAN, R. S. Engenharia de Software. 6. ed. São Paulo: McGraw-Hill, 2006.

SADALAGE, P.; FOWLER, M. **NoSQL Distilled**: A brief guide to the emerging world of polyglot persistence. New York: Addison-Wesley Professional, 2012.

SCHMIDT, D. C.; WALLNAU, K. ***Component-Based Software Engineering: Putting the Pieces Together*** por Clemens Szyperski. New York: Addison Wesley, 2001.

TYPESCRIPT; Disponível em: <https://www.typescriptlang.org/>. Acesso em: 31 ago. 2023.

W3. Disponível em: <https://www.w3.org/Protocols/>. Acesso em: 31 ago. 2023.

W3C. Disponível em: <https://www.w3schools.in/mvc-architecture>. Acesso em: 31 ago. 2023.

W3C ARQUITETURA WEB. Disponível em: <https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest>. Acesso em: 31 ago. 2023.

WORKING with DataSource. **TypeORM**, S.d. Disponível em: <https://typeorm.io/data-source>. Acesso em: 31 ago. 2023.