API RESTful
COM SPRING BOOT

Guia Prático para Iniciantes

ANA PAULA FREITAS
HALEX MIRANDA
GABRIEL VINICIUS PORTELA
EDUARDO SOUSA
REGIANA CRUZ

DComp
Departamento de Computação

INSTITUTO FEDERAL
Maranhão
Campus Monte Castelo

API RESTful
COM SPRING BOOT
Guia Prático para Iniciantes

Título: API RESTful com Spring Boot: Guia Prático para Iniciantes.

Autores: Ana Paula Rabelo de Freitas, Eduardo Assunção de Sousa, Gabriel Vinicius Pinto Portela, Halex Silva Miranda e Regiana Barbosa Lima Cruz.

Resumo: Este ebook é destinado a todos que desejam aprender a construir sua primeira API RESTful. Estruturado com explicações acessíveis, o material é dividido em 4 módulos, otimizando o processo de aprendizado. Os tópicos abordados abrangem desde os conceitos básicos até a implementação prática, proporcionando uma compreensão abrangente.

Palavras-chave: API, RESTful, Spring Boot, Desenvolvimento Web, Programação, Java, padrão MVC.

Categorias: Computação, Desenvolvimento de Software, Programação, Web Development, Java, Spring Framework.

Idioma: Português.

Número de Páginas: 75

Data de Publicação: 30 de novembro de 2023.

Edição: 1ª edição.

Formato: PDF.

Sobre o livro

Este ebook é destinado a todos que desejam aprender a construir sua primeira API RESTful. Estruturado com explicações acessíveis, o material é dividido em 4 módulos, otimizando o processo de aprendizado. Os tópicos abordados abrangem desde os conceitos básicos até a implementação prática, proporcionando uma compreensão abrangente.

Ao longo do guia, exploramos aspectos cruciais, como a configuração do ambiente de desenvolvimento, operações CRUD e a integração com bancos de dados. As explicações são claras e complementadas por exemplos relevantes, capacitando os leitores a desenvolverem suas próprias APIs com confiança.

Este recurso não apenas oferece conhecimentos essenciais, mas também se destaca como uma valiosa ferramenta em sua jornada de aprendizado no desenvolvimento web. Ao fornecer uma base sólida e prática, o ebook se torna um guia indispensável para aqueles que buscam aprimorar suas habilidades nesse cenário dinâmico.

Sumário

Módulo 1

Conceitos Básicos

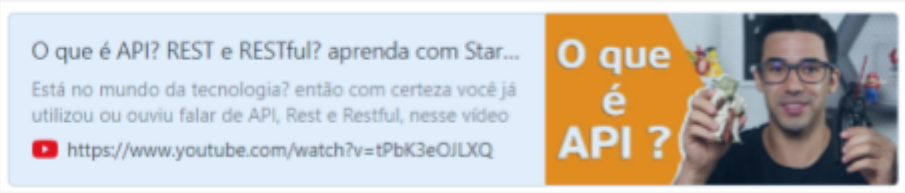
Parte 1

O que é API?

Uma **API** Web ou **API** de serviço da Web é uma interface de processamento de aplicações entre um servidor da Web e um navegador da Web. O termo API, refere-se à sigla em inglês ***Application Programming Interface*** ou em português **Interface de Programação de Aplicações**, faz referência a uma interface de aplicações, que pode comunicar-se com outros sistemas.

API trata-se de um **conjunto de rotinas** e **padrões estabelecidos,** e **documentados** por uma aplicação, para que outras aplicações consigam utilizar suas funcionalidades, sem precisar conhecer detalhes da implementação. Desta forma, entende-se que as APIs permitem uma **interoperabilidade entre aplicações**. Em outras palavras, a comunicação entre aplicações e entre os usuários. Exemplo de API: Twitter Developers e Facebook Developers.

| Dica



Nesse vídeo, API é explicada através de uma analogia com a história de “Star Wars”. É um ótimo compreender melhor os conceitos.

Parte 2

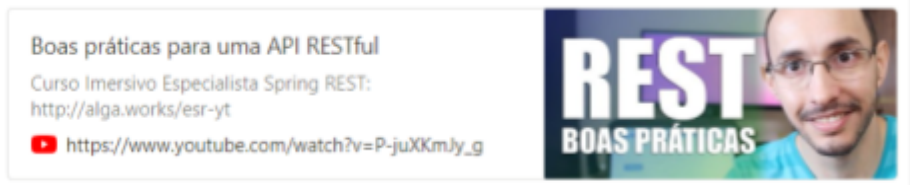
O que é REST?

REST significa ***Representational State Transfer***, em português, **Transferência de Estado Representacional.** Refere-se a uma abstração da arquitetura Web. Basicamente, o REST consiste em princípios e/ou regras que, quando seguidas, permitem a criação de um projeto com interfaces bem definidas, permitindo que as aplicações se comuniquem.

| Dica

Nesse vídeo, você pode ter uma visão geral do arquitetura REST. É um ótimo vídeo para aprender mais detalhes.

| Dica



Nesse vídeo, você pode ter uma visão geral das boas práticas para construir uma API RESTful. É um ótimo vídeo para aprender mais detalhes.

Parte 3

O que é RESTful?

Considerando que o REST é um estilo de arquitetura, ou seja, uma série de restrições que devem ser seguidas no processo de criação de um web service, podemos definir o RESTful como a utilização dos **princípios** do REST para o desenvolvimento de soluções ou serviços.

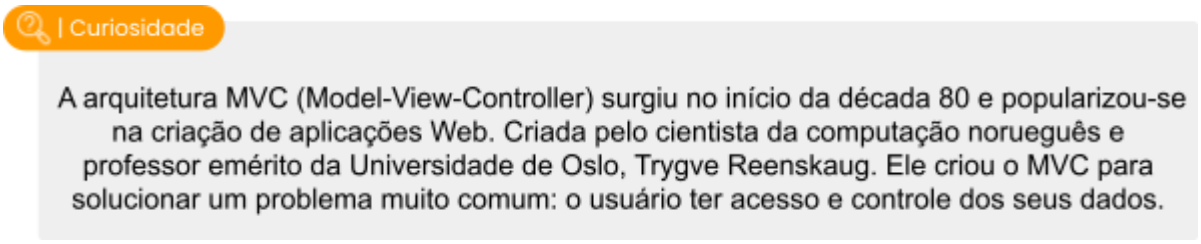
Para que a arquitetura de um sistema seja considerada RESTful, é necessário possuir certos padrões:

- **cliente-servidor:** consiste em refinar a portabilidade entre várias plataformas de interface do usuário e do servidor, permitindo uma evolução independente do sistema;
- **interface uniforme:** descreve a integração uniforme entre cliente e servidor, construindo uma interface que identifique e represente recursos bem definidos, mensagens autodescritivas;
- **stateless:** aponta que cada integração via API tem acesso a dados completos e explícitos;
- **cache:** indispensável para reduzir o tempo médio de resposta, melhorando a eficiência, desempenho e escalabilidade da comunicação;
- **camadas:** concede que a solução seja menos complexa, altamente flexível e desacoplada.

Parte 4

Introdução ao padrão MVC

Agora que você já sabe o que é uma API RESTful, vamos conhecer o padrão de arquitetura MVC. Você pode estar se perguntando: "MVC, o que isso significa? Para que serve? E por que isso é importante?". Bem, o MVC é um padrão de arquitetura de software usado para separar uma aplicação em camadas. Ele separa o código relativo à interface do usuário das regras de negócio. Cada camada tem suas próprias responsabilidades e não se preocupa com as responsabilidades das outras. Essa divisão ajuda na compreensão do código, na sua reutilização e manutenção, padroniza a comunicação entre as camadas e é muito fácil de implementar.



Mas afinal, quais camadas o MVC têm?

MVC é o acrônimo de Model-View-Controller (em português: Modelo-Visão-Controle) formado por três camadas:

- ***Model* (Modelo):** a camada de acesso e manipulação dos dados. É responsável por representar as regras de negócio e suas entidades. Ela fica aguardando ser chamada para liberar o acesso aos dados para que sejam coletados, armazenados e exibidos. Essa camada deve ser invisível ao usuário.

  Quando um usuário envia um formulário ou quando o banco de dados recebe ou envia dados, o ***Model*** está lá para orquestrar essas ações, mas não precisa saber em que momento isso acontece ou como esses dados serão apresentados para o usuário.

- ***View* (Visão):** a camada de apresentação dos dados, ou seja, a interface do usuário. Ela é responsável por receber a entrada de dados e exibi-los ao usuário, renderizando essa apresentação sempre que necessário. É a única camada que deve ser visível ao usuário.

  A camada ***View*** deve saber como apresentar os dados corretamente, mas não precisa saber quando fazer isso ou como os dados foram extraídos ou coletados; apenas os exibe para quem fez a solicitação.

  Os dados podem ser exibidos de diversas formas. Seja em uma interface feita com algum framework Web, mas também pode ser em formato PDF, JSON, XML, tokens de autenticação, ou até mesmo o retorno dos dados do servidor *RESTful*, dentre muitos outros.

- ***Controller* (Controle):** a camada de controle do fluxo da aplicação, ou seja, a lógica da aplicação. É responsável por coordenar a camada ***Model*** e a camada ***View***. É o intermediário entre a ***Model*** e a ***View***. É a primeira camada que recebe requisição dentro do padrão MVC.

  Outra responsabilidade muito importante do ***Controller*** é cuidar das requisições recebidas (*request*) e as respostas (*response*). Ele atua como o maestro que coordena a troca de informações entre o usuário e o sistema, garantindo sua comunicação.

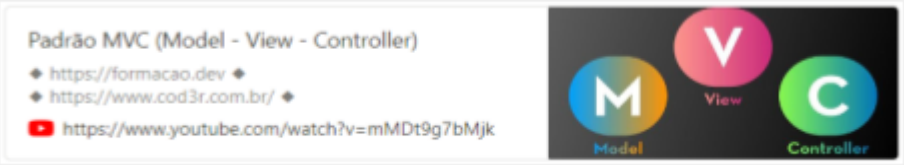
  O ***Controller*** pode tanto levar os dados de um formulário que o usuário preencheu na interface (***View***) para serem gravados (***Model***) e armazenados no banco de dados quanto pegar os dados da ***Model*** (que estão no banco) para ***View*** exibidos na ***View*** (em um *dashboard*, por exemplo).

  O ***Controller*** não precisa saber como obter os dados, muito menos como eles são mostrados para o usuário, ele se preocupa apenas em determinar quando essas ações devem acontecer.

Observação importante

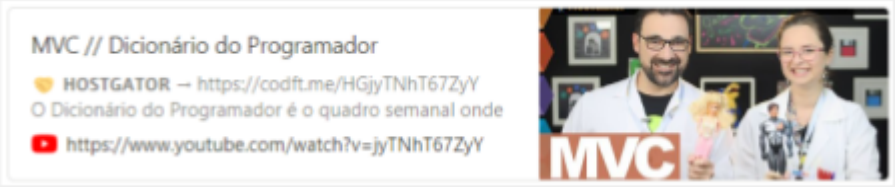
O ***Model*** sempre deve notificar as outras camadas quando há alguma mudança no seu estado. Essas notificações são necessárias para que o ***View*** atualize o que é exibido ao usuário e o ***Controller*** modifique o conjunto de ações disponíveis (caso necessário).

| Dica



Nesse vídeo, o padrão MVC é explicado através de uma analogia com a história do filme "Karate Kid", destacando a importância dos fundamentos para o desenvolvimento sólido de uma aplicação.

| Dica



Nesse vídeo, você pode ter uma visão geral do padrão MVC. É um ótimo vídeo para aprender mais detalhes sobre essa arquitetura.

| Dica

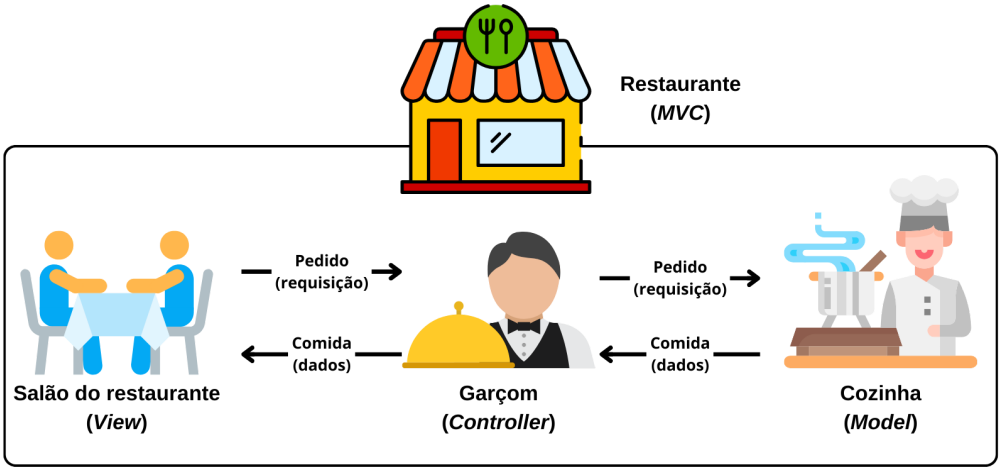
Como funciona a arquitetura MVC no desenvolvimento de software

Nesse artigo, a explicação é mais aprofundada e apresenta outros conceitos que auxiliarão você ainda mais.

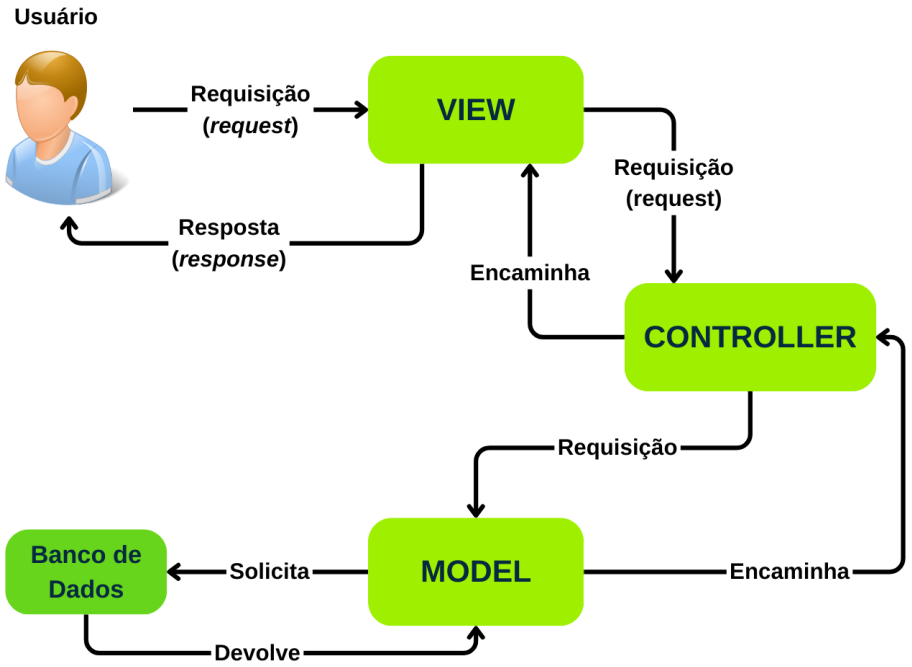
Como o MVC funciona?

Para que você entenda melhor os princípios do MVC e como essas três camadas interagem entre si, pense em um restaurante.

No restaurante (MVC), temos o salão (***View***), a cozinha (***Model***) e o garçom (***Controller***). No salão do restaurante (***View***), os clientes (usuários) interagem com a comida, apresentada da forma que o cliente pediu. A cozinha (***Model***) é responsável por todas as ações envolvendo comida (dados): preparação dos ingredientes, execução das receitas e criação dos pratos finais. Por fim, o garçom (***Controller***) é o encarregado de receber os pedidos dos clientes (requisições dos usuários), passá-los para a cozinha (***Model***), garantir que os pratos estejam corretos e servir os pratos nas mesas, apresentando aos clientes o resultado final (***View***).



Adaptando essa ilustração para o contexto real de uma aplicação, funciona da seguinte forma:



A dinâmica é bem simples. O usuário interage com a interface (***View***) que recebe todas as requisições e envia para o ***Controller***. O ***Controller*** verifica as informações e encaminha para o ***Model***. No ***Model***, os dados da requisição são guardados ou recuperados do banco de dados. Em seguida, retorna para o ***Controller*** os dados indicando que a operação de criação/atualização dos dados foi bem-sucedida. O ***Controller*** processa essas informações e encaminha para ***View***, que atualiza a interface do usuário para refletir as mudanças do ***Model***.

Esse fluxo só acontece quando for necessário acessar os dados. Caso isso não aconteça, a requisição não precisa ser encaminhada para a camada ***Model***.

E a implementação do MVC?

Vai depender da linguagem de programação que você está trabalhando e do contexto da aplicação. Geralmente criamos uma classe abstrata para cada camada (Model, View e Controller), com seus respectivos atributos e funções. Se o sistema precisar de camadas diferentes para cada aspecto do seu contexto, pode-se criar subclasses dessas classes abstratas principais. Lembrando que pode ter mais de um modelo, visão e controle.

Avaliando seus conhecimentos:

Agora que você já sabe os conceitos básicos do padrão MVC e como ele funciona, vamos testar seus conhecimentos? Desafio você a responder, com suas próprias palavras, as seguintes perguntas:

1) O que é o padrão MVC?
2) Quais partes compõem essa arquitetura?
3) O que é *model*? Quais suas responsabilidades?
4) O que é *view*? Quais suas responsabilidades?
5) O que é *controller*? Quais suas responsabilidades?
6) Descreva o fluxo da aplicação (começando da requisição do usuário até o retorno/resposta) estruturada com o padrão MVC.

E aí, como foi? Se ficou difícil responder alguma pergunta, não hesite em voltar e revisar os tópicos do capítulo. Este é o momento perfeito para solidificar seu entendimento antes de seguirmos em frente. No próximo capítulo, você vai aprender como preparar o ambiente de desenvolvimento para implementar uma API RESTful com Spring Boot. Avante!

Módulo 2

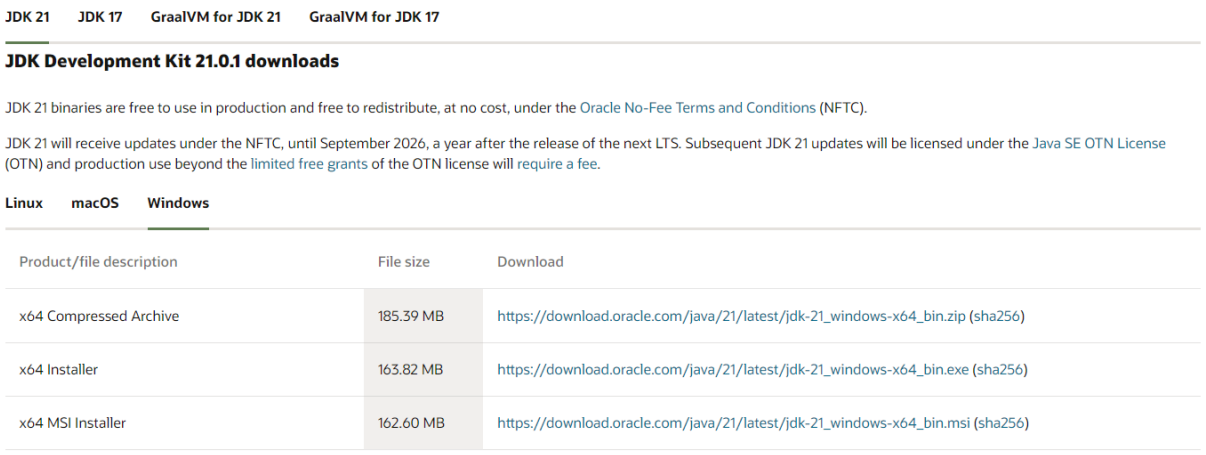
Ambiente de Desenvolvimento

Parte 1

Java

Vamos começar a configurar o ambiente de desenvolvimento para esse guia, para isso precisamos instalar o java, mais especificamente o JDK, sigla para Java Development Kit.

Para isso vamos no site da Oracle para fazer o download do JDK, segue o site: https://www.oracle.com/br/java/technologies/downloads/ . Aqui vamos selecionar qual sistema operacional usamos, no nosso caso é o windows:



JDK 21 JDK 17 GraalVM for JDK 21 GraalVM for JDK 17

JDK Development Kit 21.0.1 downloads

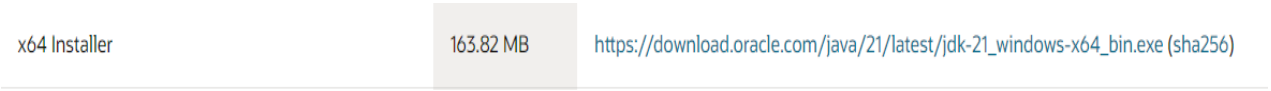
JDK 21 binaries are free to use in production and free to redistribute, at no cost, under the Oracle No-Fee Terms and Conditions (NFTC).

JDK 21 will receive updates under the NFTC, until September 2026, a year after the release of the next LTS. Subsequent JDK 21 updates will be licensed under the Java SE OTN License (OTN) and production use beyond the limited free grants of the OTN license will require a fee.

Linux macOS Windows

Product/file description	File size	Download
x64 Compressed Archive	185.39 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.zip (sha256)
x64 Installer	163.82 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.exe (sha256)
x64 MSI Installer	162.60 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.msi (sha256)

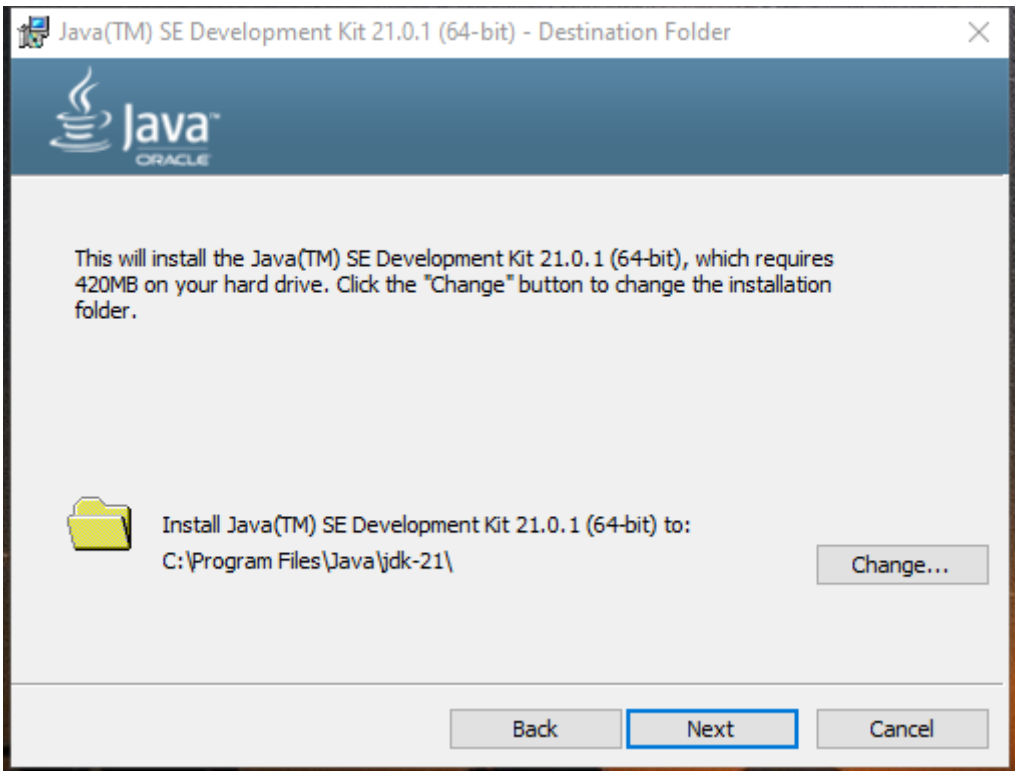
Usaremos o x64 Installer:



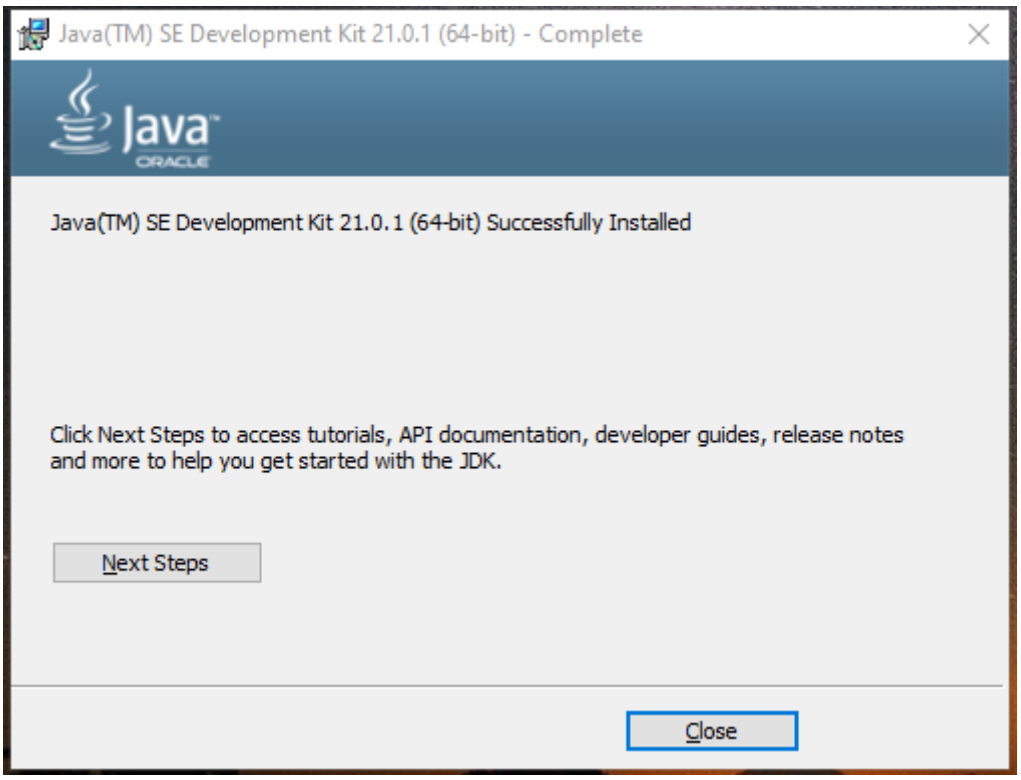
x64 Installer	163.82 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.exe (sha256)

Após terminar o download identifique onde está o instalador e clique duas vezes no ícone:

Após isso clique em next até chegar a essa tela:



Caso queira alterar o local da instalação clique em Change, após clique em next e espere a instalação acabar:



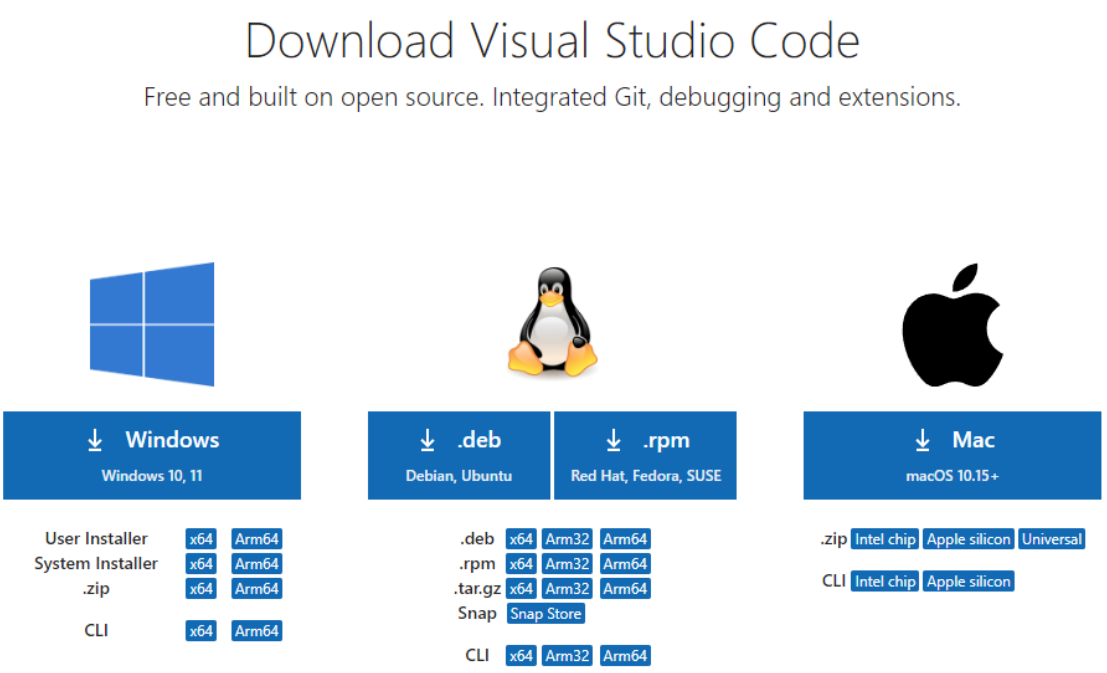
Pronto, o JDK já está instalado.

Parte 2

Visual Studio Code

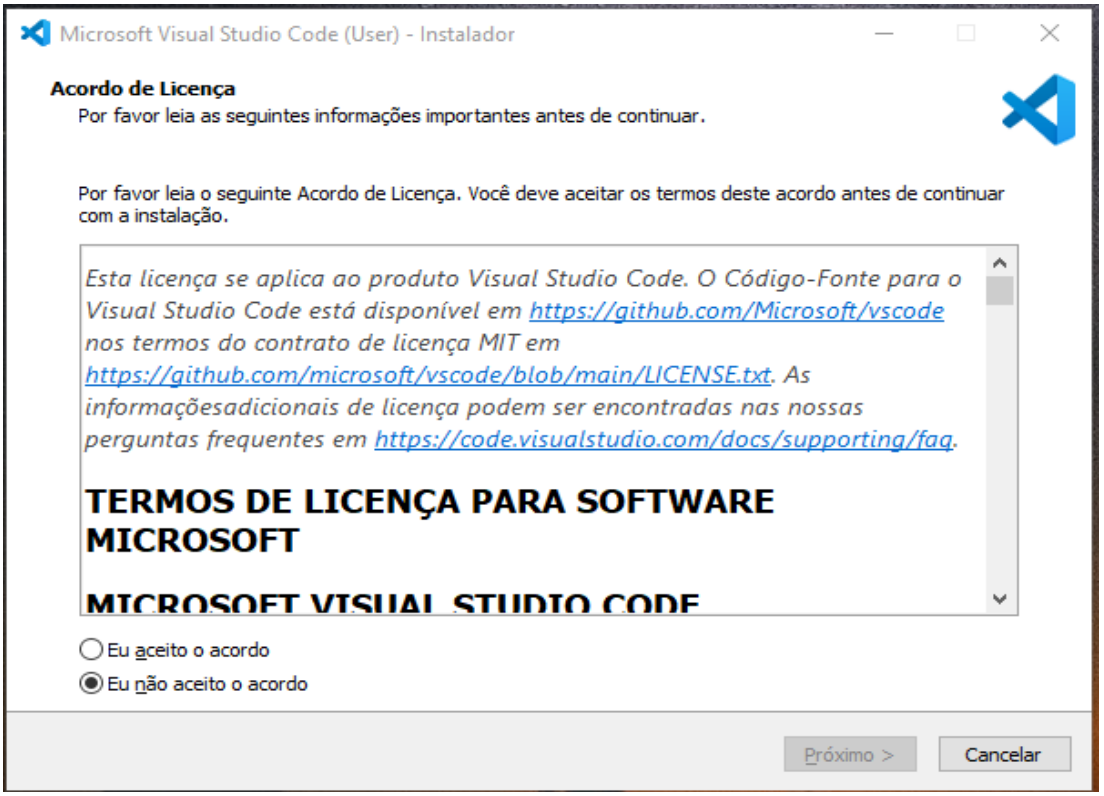
Agora que já instalamos o JDK, vamos instalar uma IDE, sigla para Integrated Development Environment. Aqui iremos usar o Visual Studio Code da Microsoft, mas você pode optar por outras como NetBeans, Eclipse, IntelliJ, entre outras.

Para isso vamos seguir um passo a passo muito parecido com a instalação do JDK, primeiro vamos fazer o download do vscode, segue o link: https://code.visualstudio.com/download. Aqui novamente você tem que escolher qual sistema operacional baixar, assim como antes vamos com o windows.

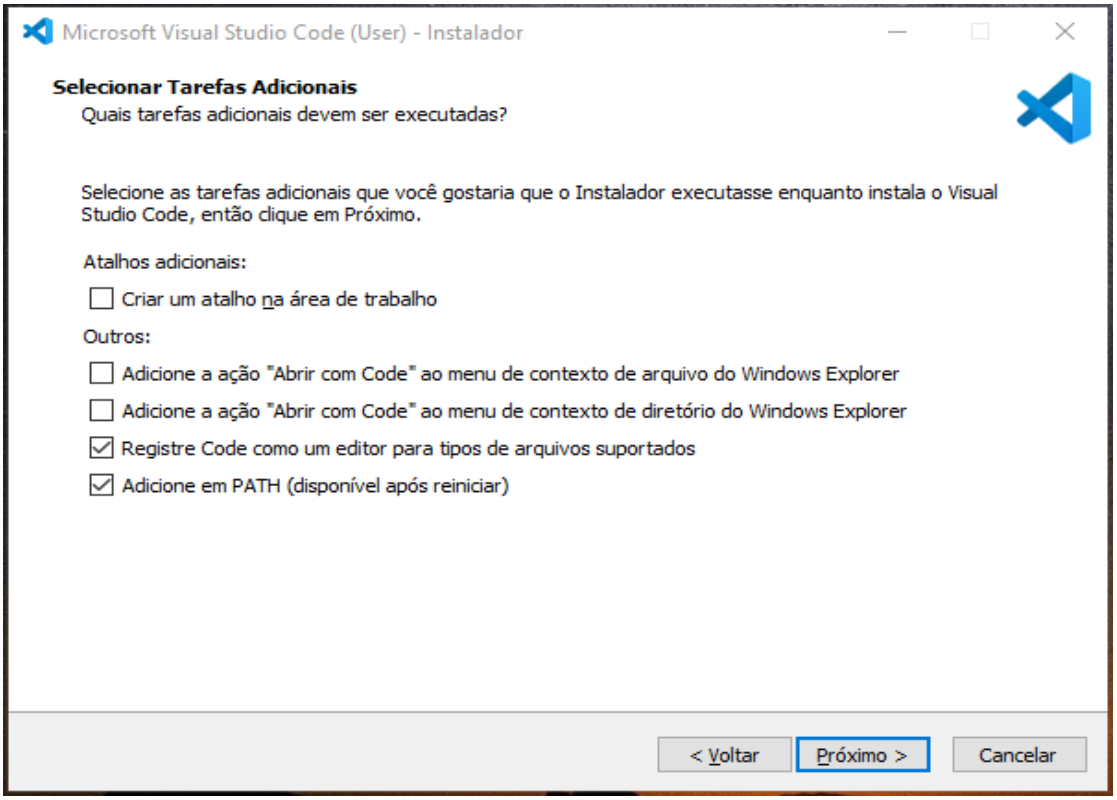


Após terminar o download identifique onde está o instalador e clique duas vezes no ícone:

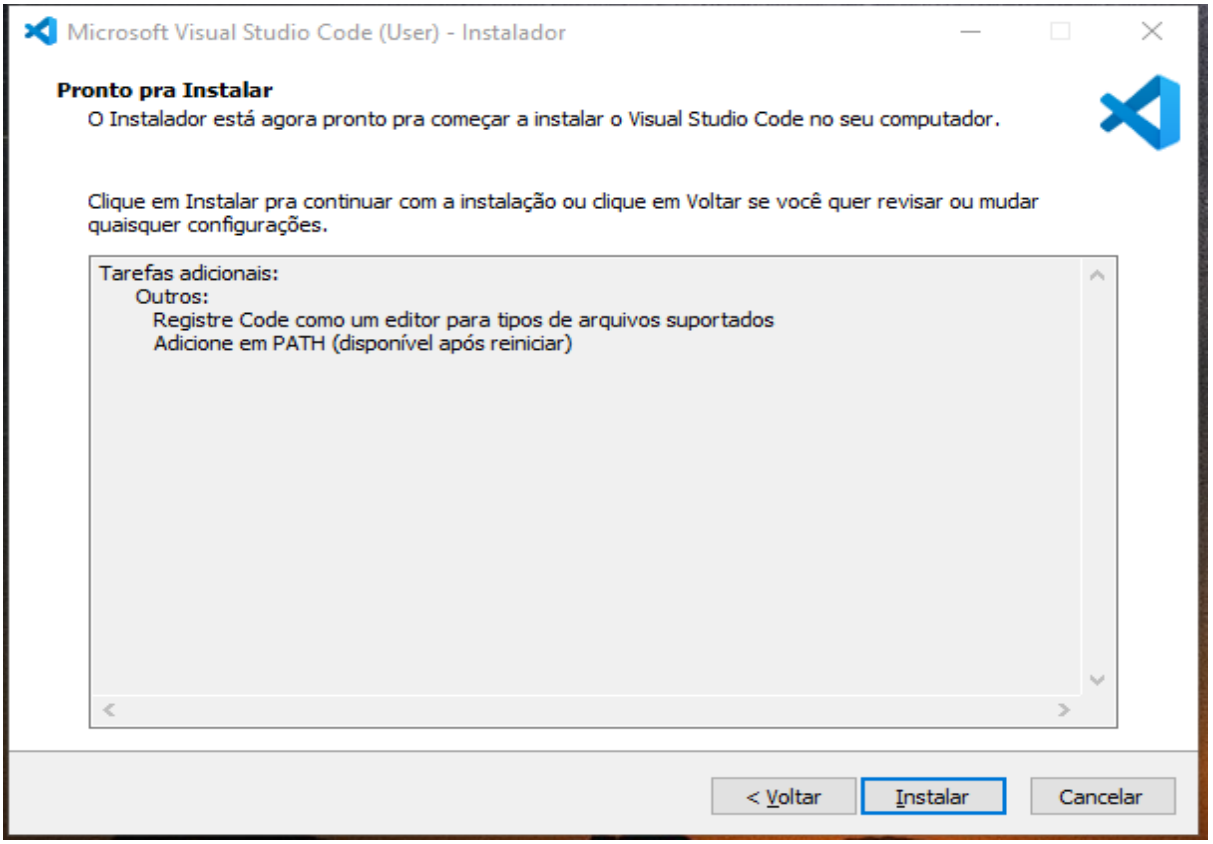
Clique em aceito o contrato e depois em próximo:



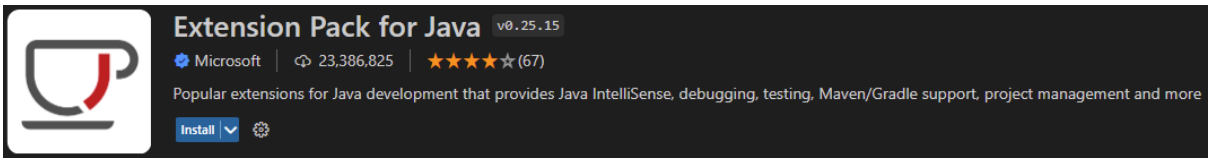
Próximo novamente:



E instalar, espere a instalação terminar:



Agora que o visual já está instalado precisamos instalar uma extensão, a Extension Pack for Java:



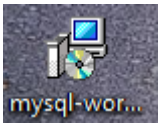
Pronto, agora podemos seguir em frente.

Parte 3

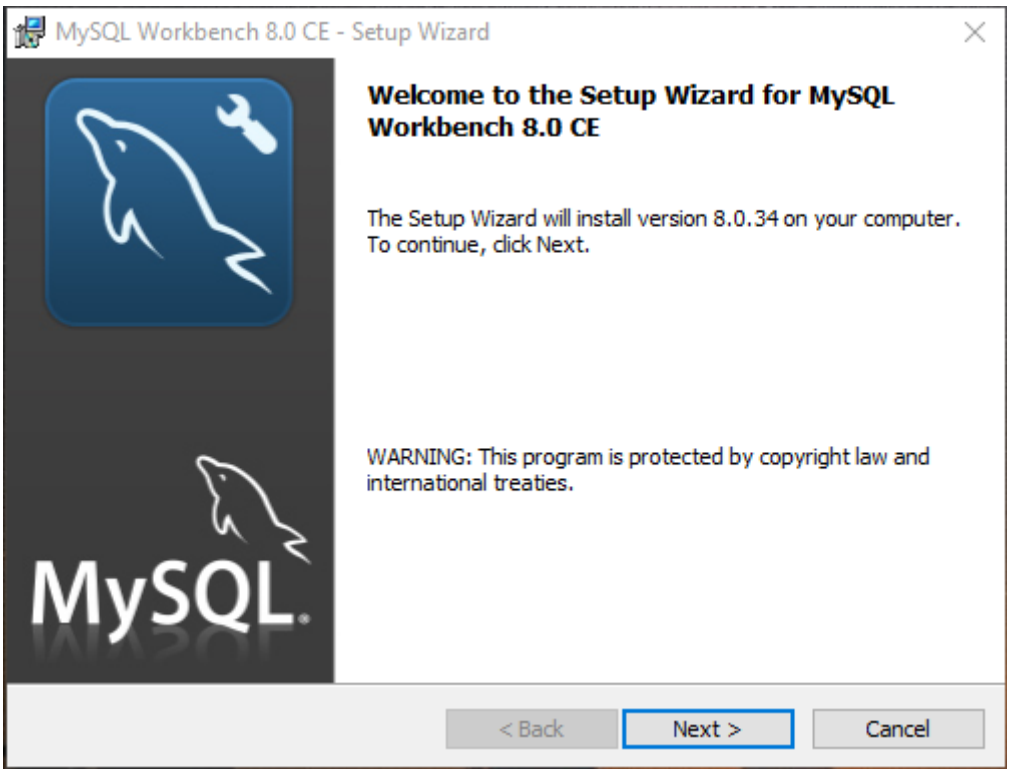
Banco de Dados

Para o banco de dados vamos utilizar o MySQL, mas caso queira utilizar outro fique a vontade, lembre-se apenas de configurar conforme o banco que escolher. Para fazer a instalação o passo a passo é muito parecido, primeiro vamos fazer o download no link: https://dev.mysql.com/downloads/workbench/.

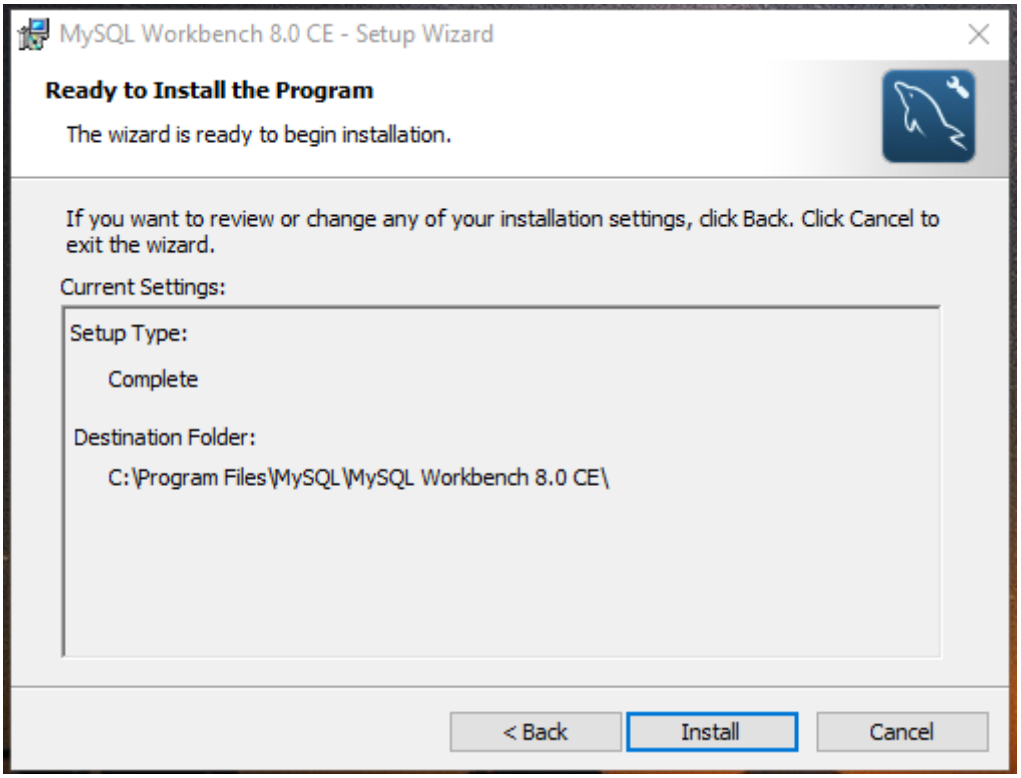
Após terminar o download identifique onde está o instalador e clique duas vezes no ícone:

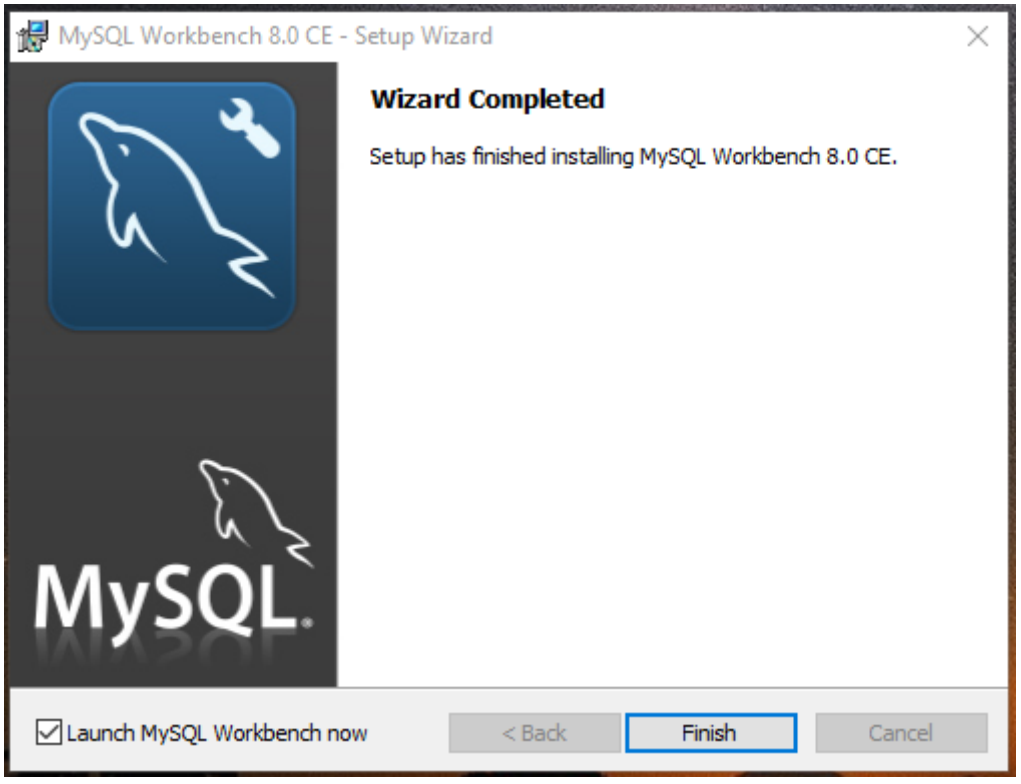


Clique em next:



Clique em Install e espere:





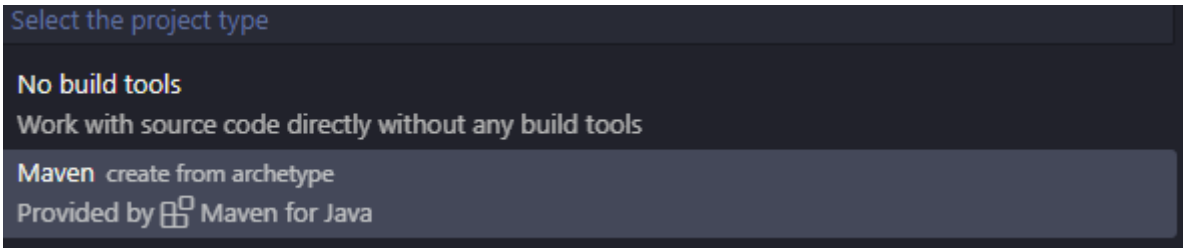
Pronto, MySQL instalado.

Parte 4

Spring Boot

Para preparar o Spring Boot precisamos primeiro abrir o visual studio code, criaremos um novo projeto, para isso vamos pressionar as teclas Ctrl+Shift+P para abrir o palete de comandos do visual studio code.

No palete de comandos vamos digitar Java:create java project, criaremos um projeto maven:



Após isso podemos pressionar Enter duas vezes, até chegarmos na parte de selecionar um nome para o nosso projeto:

Após isso abrimos a pasta onde o projeto foi criado, e vamos no arquivo pom.xml que é o arquivo onde passaremos a dependência do spring. Nesse arquivo vamos criar uma tag dependencies, e aqui colocaremos a tag do spring core e do spring web. Para fazer isso rapidamente vamos novamente abrir o palete de comandos do visual studio code(Ctrl+Shift+P) e digitamos Maven:add a dependency, aqui colocamos palavras chaves para procurar por uma dependência e pressionamos o Enter. No caso vamos colocar spring core e depois spring web:

```
Unset
<dependencies>
	<dependency>
		<groupId>spring</groupId>
		<artifactId>spring-web</artifactId>
		<version>1.0.2</version>
	</dependency>
	<dependency>
		<groupId>spring</groupId>
		<artifactId>spring-core</artifactId>
		<version>1.0.2</version>
	</dependency>
</dependencies>
```

Pronto, preparamos o spring para o nosso projeto.

Parte 5

Postman

Para facilitar na hora da codificação e nos testes, vamos utilizar o postman. O postman pode ser utilizado no direto pelo navegador, sendo necessário apenas criar uma conta na plataforma antes. No nosso caso vamos utilizar o aplicativo de desktop, para isso vamos fazer o download no link: https://www.postman.com/downloads/.

Após terminar o download identifique onde está o instalador e clique duas vezes no ícone:

e é isso, a instalação já está feita.

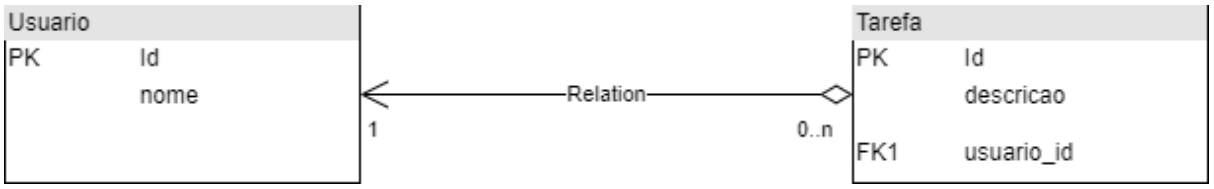
Módulo 3

Desenvolvimento da API RESTful

Parte 1

Descrição do projeto

Para ilustrar os conceitos principais de uma REST API com Spring Boot, nós vamos criar uma aplicação simples de To-Do List. Nesta aplicação, teremos duas entidades principais: `Tarefa` e `Usuario`. O diagrama de entidade-relacionamento abaixo mostra a relação entre essas entidades.



Relacionamento entre as entidades

A aplicação To-Do List permitirá criar, atualizar, deletar e listar tarefas e usuários. Nossa API terá os seguintes endpoints:

***POST /usuarios**: cria um usuário*

O body da requisição deverá ser:

```
Unset
{
  "nome": "Edu"
}
```

O retorno deve ser 201 em caso de sucesso e o body de resposta deve ser o próprio objeto criado:

```
Unset
{
  "id": 1,
  "nome": "Edu",
  "tarefas": []
}
```

PUT /usuarios/{id}: *modifica um usuário*

Devemos passar o id do usuário a ser modificado e o body da requisição deve conter os novos dados:

```
Unset
{
  "nome": "Ana"
}
```

Retorna 200 em caso de sucesso e o objeto criado.

DELETE /usuarios/{id}*: remove um usuário*

Devemos passar o id do usuário a ser excluído e um body vazio. Retorna 204 em caso de sucesso e 404 caso o não exista o id.

GET /usuarios*: retorna todos os usuários criados*

Retorna 200 e a lista de usuários criados. Caso não exista usuários cadastrados, retorna uma lista vazia.

POST /tarefas: *cria uma tarefa.*

O body da requisição deverá ser:

```
Unset
{
  "descricao": "Desenvolver uma API",
  "usuario": {
    "id": 1
  }
}
```

Deve conter a descrição da tarefa e o id do usuário associado à ela.

O retorno deve ser 201 em caso de sucesso e o body de resposta deve ser o próprio objeto criado:

```
Unset
{
  "id": 1,
  "descricao": "Desenvolver uma API"
}
```

Retorna 500 caso o id do usuário não exista.

PUT /tarefas/{id}: *modifica uma tarefa*

Devemos passar o id da tarefa a ser modificada e o body da requisição deve conter os novos dados:

```
Unset
{
  "descricao": "Desenvolver uma API RESTful",
  "usuario": {
    "id": 1
  }
}
```

Retorna 200 em caso de sucesso e o objeto criado.

DELETE /tarefas/{id}*: remove uma tarefa*

Devemos passar o id da tarefa a ser excluída e um body vazio. Retorna 204 em caso de sucesso e 404 caso o não exista o id.

GET /tarefas*: retorna todos as tarefas criadas*

Retorna 200 e a lista de tarefas criadas. Caso não exista tarefas cadastradas, retorna uma lista vazia.

Vamos então implementar essas funcionalidades!

Parte 2

Configuração do Projeto

Para iniciar nosso projeto, precisamos nos certificar de que temos o Java Development Kit instalado, pois é esse ambiente de desenvolvimento de software que irá fornecer as ferramentas e bibliotecas necessárias para nossa aplicação Java. Conseguimos checar com o seguinte comando no CMD:

```
javac -version
```

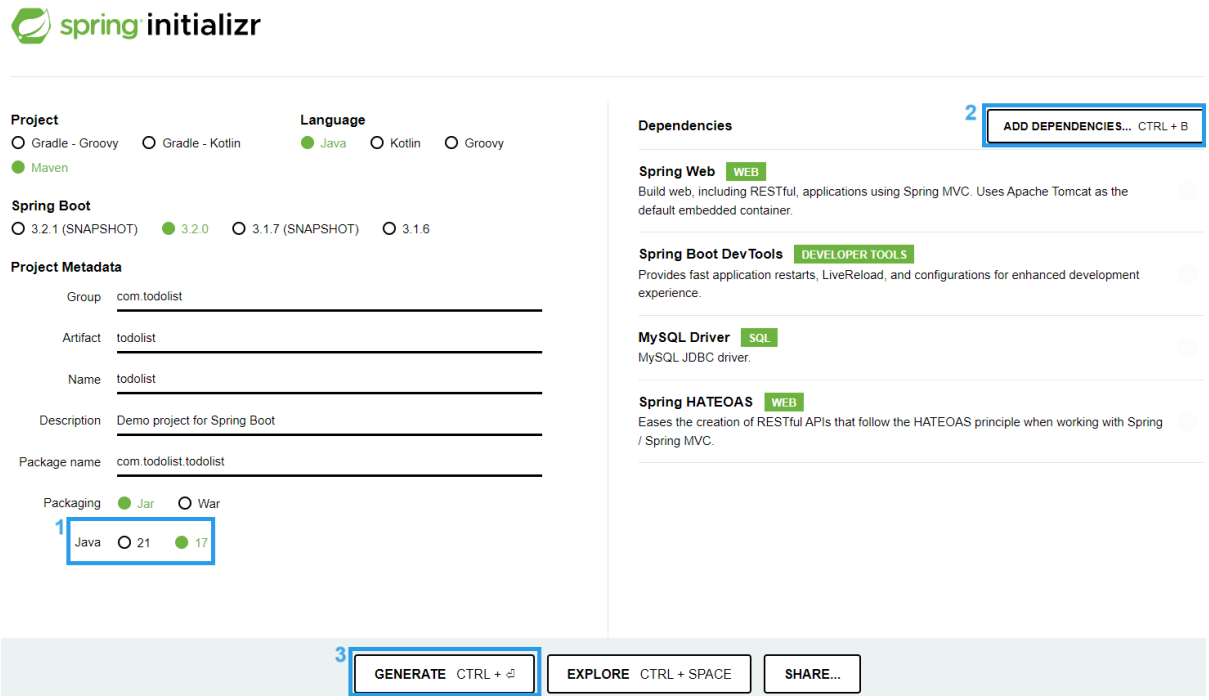
Se o comando não for reconhecido, você deve baixar e instalar a versão mais recente do JDK. Você pode conferir mais instruções no Módulo 2.

Um ambiente de desenvolvimento também facilitará o desenvolvimento. Utilizaremos o Visual Studio Code, mas sinta-se livre para utilizar a IDE de sua preferência.

> | Dica
>
> Para ajudá-lo a configurar o ambiente rapidamente, você pode instalar o Coding Pack Java, que inclui o VS Code, JDK, e todas as extensões essenciais Java.

Iniciaremos a criação da nossa aplicação utilizando o Spring Initializr, uma ferramenta que gera projetos utilizando Spring Boot. Através dele definiremos o nome e tipo do projeto, pacotes, versão, dependências e linguagem (Java, Groovy ou Kotlin).

Abra o Spring Initializer no navegador: https://start.spring.io/. Configure o projeto como mostrado na imagem abaixo. Selecione a linguagem Java no campo Language e a versão 17 no campo Java. Em Project Metadata você pode definir os campos como preferir, eles ajudarão a identificar o seu projeto.



Criação do projeto pelo Spring Initializr

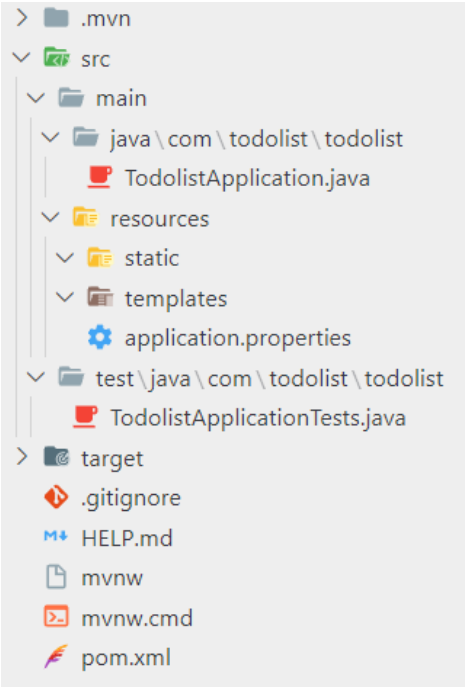
Adicione também as dependências listadas. O Spring Web facilita o desenvolvimento de aplicações web no padrão MVC (model-view-controller). O Spring Boot DevTools fornece ferramentas que agilizam o desenvolvimento durante o ciclo de vida da aplicação. O MySQL Driver permite a conexão, consulta e atualização de dados no banco de dados MySQL e a aplicação. O Spring HATEOAS (*Hypermedia As The Engine Of Application State*) é essencial para serviços RESTful, pois facilita a implementação do princípio HATEOAS, que adiciona links aos recursos que indicam ações possíveis e fornecem uma navegação mais intuitiva para os clientes da API. Entenderemos mais sobre cada dependência e sua utilidade ao longo do desenvolvimento.

! | Nota

O campo Project permite escolher entre Maven e Gradle, que são os gerenciadores de dependências mais populares na comunidade Java. Spring Boot define a versão específica do Spring Boot para o projeto, recomendamos escolher a versão mais recente estável (sem a tag “SNAPSHOT”, que indica uma versão de desenvolvimento em andamento).

Após a configuração dos campos, o botão “Generate” irá baixar um arquivo do tipo jar contendo a estrutura inicial do projeto. Descompacte o arquivo zip em um diretório da sua escolha. Abra sua IDE escolhida e importe o projeto. No VSCode, clique em “Arquivo” → “Abrir pasta” e selecione o diretório do projeto salvo.

A aplicação deverá ter a seguinte estrutura:



Estrutura inicial do projeto

O `pom.xml` é um arquivo de configuração do Maven (ou Gradle, dependendo da escolha feita durante a criação do projeto) onde você encontrará todas as dependências que você escolheu durante a criação do projeto, adicionadas automaticamente pelo Spring Initializr. As configurações de construção, plugins Maven, propriedades e outras configurações relacionadas ao projeto também são definidas no arquivo.

A classe `TodoApplication.java` contém o ponto de entrada da nossa aplicação Spring Boot, incluindo o método `main`, que será responsável por carregar todas as dependências.

A annotation ***@SpringBootApplication*** é uma anotação responsável pela configuração e inicialização eficiente da aplicação. Essa anotação combina três anotações principais e que simplifica o desenvolvimento: ***@Configuration***: sinaliza

que a classe contém configurações para o contexto da aplicação e pode ser personalizada, ***@EnableAutoConfiguration***: a configuração automática do Spring Boot é ativada com base nas dependências detectadas e ***@ComponentScan***: habilita a varredura de componentes (são eles, os controladores, serviços e repositórios, e serão explicados com mais detalhes nos próximos tópicos) no pacote da aplicação e em seus subpacotes, que são automaticamente registrados e gerenciados pelo Spring.

```java
@SpringBootApplication
public class TodolistApplication {
  public static void main(String[] args) {
    SpringApplication.run(TodolistApplication.class, args);
  }
}
```

Com o projeto criado e suas configurações iniciais definidas, criaremos nossas entidades, repositórios e controladores.

Parte 3

Criação do Modelo e Repositório

Após configurar a estrutura inicial do projeto, o próximo passo é criar as entidades que representarão os objetos do nosso domínio. Uma `Entity` representa um conjunto específico de dados que descrevem uma coisa ou conceito do domínio de aplicação. No contexto da nossa aplicação To-Do list, teremos as entidades `Tarefa` e `Usuario`.

O Spring Boot reconhece como componentes da aplicação todas as classes definidas no mesmo pacote da classe que contém o método `main`. Por isso, criaremos uma pasta `entity` em `com.todolist.todolist` e definiremos nossa primeira entidade `Usuario` nela.

```java
package com.todolist.todolist.entity;

import java.util.ArrayList;
import java.util.List;

import jakarta.persistence.CascadeType;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToMany;

@Entity
public class Usuario {
```

```java
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
private String nome;
@OneToMany(mappedBy="usuario", cascade=CascadeType.ALL)
private List<Tarefa> tarefas = new ArrayList<>();

// construtores
public Usuario() {
}

public Usuario(String nome) {
    this.nome = nome;
}

// getters e setters
public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
```

```
        this.nome = nome;
    }

    public List<Tarefa> getTarefas() {
        return tarefas;
    }

    public void setTarefas(List<Tarefa> tarefas) {
        this.tarefas = tarefas;
    }
}
```

Aqui apresentamos mais algumas anotações importantes.

@Entity: Anotação que marca a classe como uma entidade JPA e pode, portanto, ser salva no banco de dados.

@Id: Indica a chave primária da entidade, o atributo único e não nulo.

@GeneratedValue(strategy = GenerationType.IDENTITY): Especifica a forma de incrementar automaticamente o valor para o Id. Os tipos podem ser AUTO, IDENTITY, SEQUENCE e TABLE. O tipo IDENTITY é auto-incrementado, ou seja, irá gerar uma sequência 1, 2, 3, 4… para cada novo usuário criado.

@OneToMany(mappedBy = "usuario", cascade = CascadeType.ALL): Define um relacionamento de um para muitos com a entidade `Tarefa`, mapeado (**`mappedBy`**) pelo atributo `usuario`. Isso significa que um usuário pode ter várias tarefas. A opção **`cascade = CascadeType.ALL`** indica que operações de cascata, como salvar, excluir ou editar, também devem ser aplicadas às tarefas associadas.

Também definimos os outros atributos, os construtores e os getters e setters.

| Dica

O Lombok é uma biblioteca Java focada em produtividade e redução de código. Com ela podemos simplificar bastante nossa classe `Usuario`, eliminando a necessidade de escrever manualmente os construtores, getters e setters.

Precisamos inicialmente, adicionar no arquivo `pom.xml` a dependência da Lombok:

Unset
```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

Aqui está a versão da classe `Usuario` usando Lombok:

Java
```
import lombok.*;

import javax.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Data //getters e setters
@NoArgsConstructor //construtor sem atributos
public class Usuario {



    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
```

```
            @OneToMany(mappedBy = "usuario", cascade =
CascadeType.ALL)
    private List<Tarefa> tarefas = new ArrayList<>();
    public Usuario(String nome) {
        this.nome = nome;
    }
}
```

Também criaremos uma entity `Tarefa` usando Lombok:

```java
package com.todolist.todolist.entity;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;

@Entity
@Data
@NoArgsConstructor
public class Tarefa {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;

    private String descricao;

    @ManyToOne
    @JoinColumn(name = "usuario_id")
    @JsonIgnoreProperties("tarefas")
    private Usuario usuario;

    public Tarefa(String descricao, Usuario usuario) {
        this.descricao = descricao;
        this.usuario = usuario;
    }
}
```

As seguintes anotações foram introduzidas com esta classe, são elas:

@ManyToOne: Define um relacionamento muitos-para-um com a entidade `Usuario`. Isso significa que muitas tarefas podem pertencer a um único usuário.

@JoinColumn(name = "usuario_id"): Especifica o nome da coluna na tabela de `Tarefa` que será usada como chave estrangeira para a tabela de usuários.

@JsonIgnoreProperties("tarefas"): Evita a serialização recursiva ao transformar objetos em JSON. Ignora a propriedade `tarefas` da entidade `Usuario` ao serializar uma tarefa. Isso é importante para evitar um loop infinito ao serializar as entidades relacionadas.

❗ | Nota

Além dos relacionamentos um-para-muitos (***@OneToMany***) e muitos-para-um (***@ManyToOne***) utilizados em nossa API, a JPA oferece outras duas anotações de relacionamento: ***@ManyToMany***: em que uma instância de uma entidade está associada a várias instâncias da outra entidade, e vice-versa (dependência mútua); e ***@OneToOne***: em uma instância de uma entidade está associada a, no máximo, uma instância da outra entidade, e vice-versa.

No Spring, os `Repository` são uma abstração que encapsula o acesso aos dados e fornece métodos para realizar operações com o banco de dados. Com eles, não precisamos escrever consultas SQL manualmente. Na nossa API, utilizaremos a interface `JpaRepository`, que já fornece métodos prontos para operações de CRUD (Create, Read, Update e Delete).

Os repositórios das nossas entidades ficarão no pacote `com.todolist.todolist.repository`.

Java
```java
package com.todolist.todolist.repository;


import org.springframework.data.jpa.repository.JpaRepository;


import com.todolist.todolist.entity.Usuario;

public interface UsuarioRepository extends
JpaRepository<Usuario, Long> {
}
```

Java

```java
package com.todolist.todolist.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.todolist.todolist.entity.Tarefa;

public interface TarefaRepository extends
JpaRepository<Tarefa, Long> {
}
```

Quando estendemos o `JpaRepository` e especificamos o tipo da entidade (`Usuario` ou `Tarefa`) e o tipo da chave primária (`Long`), o Spring Data JPA automaticamente gera a implementação de métodos como `save`, `findById`, `findAll`, `deleteById`, entre outros. Isso significa que não precisamos escrever código para essas operações, pois o Spring Data JPA cuida disso nos bastidores.

Com as entidades e repositórios criados, criaremos os serviços que fornecerão as funcionalidades da nossa API. Mas antes disso, iremos configurar o banco de dados. No caminho `src.main.resources` temos o arquivo `application.properties`, onde podemos configurar diversos aspectos da nossa aplicação e onde iremos conectar nossa aplicação ao banco de dados MySQL, URL de conexão, nome de usuário, senha e driver JDBC.

Unset

```
# Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/todolist
spring.datasource.username=root
spring.datasource.password=sua_senha
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

```
spring.jpa.hibernate.ddl-auto=create

#spring.jpa.hibernate.ddl-auto=update
```

Os campos `username` e `password` são os dados de acesso do banco. A propriedade de configuração `spring.jpa.hibernate.ddl-auto=create` significa que o Spring Boot deve criar o esquema do banco de dados se ele ainda não existir. É preciso ter cuidado ao usá-la pois pode resultar na perda de dados existentes se as tabelas já existirem no banco de dados. A propriedade `update` apenas faz alterações incrementais. A configuração `spring.datasource.driver-class-name` indica a classe do driver JDBC que o Spring Boot deve usar para se conectar ao banco de dados. Você deve ajustá-la de acordo com a versão específica.

Parte 4

Criação do Serviço

Os *services* são partes fundamentais da nossa aplicação. Eles representam o centro de tomada de decisões e ações mais complexas. Enquanto as entidades lidam com os dados e os repositórios gerenciam o acesso ao banco de dados, os services coordenam as operações mais importantes, encapsulando a lógica de negócios da nossa aplicação, indicamos isso para o Spring marcando a classe com a anotação ***@Service***.

A classe `UsuarioService` deve conter os métodos básicos de CRUD: `getAllUsuarios` (que recupera todos os usuários cadastrados), `getUsuarioById` (par recuperar um usuario pela chave primária Id), `createUsuario` (para criar um usuário), `updateUsuario` (para atualizar os dados de um usuario) e `deleteUsuario` (remover um usuário).

```java
package com.todolist.todolist.service;

import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.todolist.todolist.entity.Usuario;
import com.todolist.todolist.repository.UsuarioRepository;

@Service
public class UsuarioService {

    @Autowired
    private UsuarioRepository usuarioRepository;
```

```
    public List<Usuario> getAllUsuarios() {
        return usuarioRepository.findAll();
    }

    public Optional<Usuario> getUsuarioById(Long id) {
        return usuarioRepository.findById(id);
    }

    public Usuario createUsuario(Usuario usuario) {
        return usuarioRepository.save(usuario);
    }

    public Usuario updateUsuario(Long id, Usuario usuario) {
        if (usuarioRepository.existsById(id)) {
            usuario.setId(id);
            return usuarioRepository.save(usuario);
        }
        return null; //ou lançar uma exceção, dependendo da
lógica de negócios

    }

    public void deleteUsuario(Long id) {
        usuarioRepository.deleteById(id);
    }
}
```

A classe `TarefaService` deve possuir os mesmos métodos básicos.

Java

```
package com.todolist.todolist.service;

import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.todolist.todolist.entity.Tarefa;
```

```java
import com.todolist.todolist.repository.TarefaRepository;

@Service
public class TarefaService {

    @Autowired
    private TarefaRepository tarefaRepository;

    public List<Tarefa> getAllTarefas() {
        return tarefaRepository.findAll();
    }

    public Optional<Tarefa> getTarefaById(Long id) {
        return tarefaRepository.findById(id);
    }

    public Tarefa createTarefa(Tarefa tarefa) {
        return tarefaRepository.save(tarefa);
    }

    public Tarefa updateTarefa(Long id, Tarefa tarefa) {
        if (tarefaRepository.existsById(id)) {
            tarefa.setId(id);
            return tarefaRepository.save(tarefa);
        }
        return null; // ou lançar uma exceção, dependendo da
lógica de negócios

     }

    public void deleteTarefa(Long id) {
        tarefaRepository.deleteById(id);
    }
}
```

O uso de ***@Autowired*** no campo `Repository` nas classes elimina a necessidade de criar manualmente instâncias de objetos e permite que o Spring gerencie automaticamente as dependências entre os componentes.

> **!** | Nota
>
> É importante mencionar que, para que a injeção de dependência funcione corretamente (com o uso de ***@AutoWired***), a classe do campo deve ser gerenciada pelo Spring, ou seja, ela deve ser um bean do Spring. Isso é geralmente alcançado através de anotações como ***@Component***, ***@Service***, ***@Repository***, ou outras anotações específicas do Spring.

Parte 5

Criação do Controlador

O Spring 4.0 introduziu a anotação ***@RestController***, que combina as anotações ***@Controller*** e ***@ResponseBody***, para simplificar a criação de serviços web RESTful. Com essa annotation marcamos uma classe como um controlador, e ela é responsável por lidar com requisições na nossa aplicação.

Na classe `UsuarioController`, implemetaremos as operações descritas na parte 2: listar todos os usuarios ou por Id (**GET**), criar um usuario (**POST**), atualizar um usuário (**PUT**) e remover um usuario pelo Id (**DELETE**).

```java
package com.todolist.todolist.controller;

import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.todolist.todolist.entity.Usuario;
import com.todolist.todolist.service.UsuarioService;

@RestController
@RequestMapping("/usuarios")
public class UsuarioController {
```

```
    @Autowired
    private UsuarioService usuarioService;

    @GetMapping
    public ResponseEntity<List<Usuario>> getAllUsuarios() {
        List<Usuario> listaUsuarios =
usuarioService.getAllUsuarios();
        return
ResponseEntity.status(HttpStatus.OK).body(listaUsuarios);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Usuario>
getUsuarioById(@PathVariable Long id) {
        Optional<Usuario> usuario =
usuarioService.getUsuarioById(id);
        if(usuario.isEmpty()){
            return
ResponseEntity.status(HttpStatus.NOT_FOUND).body(usuario.get()
);
        }
        return
ResponseEntity.status(HttpStatus.OK).body(usuario.get());

    }

    @PostMapping
    public ResponseEntity<Usuario> createUsuario(@RequestBody
Usuario usuario) {
        Usuario savedUsuario =
usuarioService.createUsuario(usuario);
        return
ResponseEntity.status(HttpStatus.CREATED).body(savedUsuario);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Usuario> updateUsuario(@PathVariable
Long id, @RequestBody Usuario usuario) {
        if (!usuarioService.getUsuarioById(id).isPresent()) {
```

```
            return ResponseEntity.notFound().build();
        }

        usuario.setId(id);
        Usuario updatedUsuario =
usuarioService.createUsuario(usuario);
        return ResponseEntity.ok(updatedUsuario);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUsuario(@PathVariable
Long id) {
        if (!usuarioService.getUsuarioById(id).isPresent()) {
            return ResponseEntity.notFound().build();
        }

        usuarioService.deleteUsuario(id);
        return ResponseEntity.noContent().build();
    }
}
```

As anotações ***@_Mapping*** são como instruções que dizem ao programa como associar os métodos de uma classe a URLs específicas. Quando usamos ***@RequestMapping("/usuarios")***, estamos dizendo que todos os caminhos para os métodos nesta classe começam com `/usuarios`. Se usamos ***@GetMapping*** sem um valor específico, estamos indicando que esse método responde a solicitações do tipo `GET` para `/usuarios`, mostrando uma lista de todos os usuários cadastrados. Se adicionamos um glob pattern com ***@GetMapping("/{id}")***, o programa entende que queremos extrair um valor específico da URL, por exemplo, `/usuarios/1`. Neste caso, o valor 1 seria extraído e usado no método `getUsuarioById`.

O mesmo ocorre para ***@PostMapping***, ***@PutMapping*** e ***@DeleteMapping***.

Na classe `TarefaController` teremos os mesmos mapeamentos para a entidade `Tarefa`.

Java

```java
package com.todolist.todolist.controller;

import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.todolist.todolist.entity.Tarefa;
import com.todolist.todolist.service.TarefaService;

@RestController
@RequestMapping("/tarefas")
public class TarefaController {

    @Autowired
    private TarefaService tarefaService;
    @GetMapping
    public ResponseEntity<List<Tarefa>> getAllTarefas() {
        List<Tarefa> listaTarefas =
tarefaService.getAllTarefas();
        return
ResponseEntity.status(HttpStatus.OK).body(listaTarefas);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Tarefa> getTarefaById(@PathVariable
Long id) {
        Optional<Tarefa> tarefa =
tarefaService.getTarefaById(id);
```

```
        if(tarefa.isEmpty()){
            return
ResponseEntity.status(HttpStatus.NOT_FOUND).body(tarefa.get())
;
        }
        return
ResponseEntity.status(HttpStatus.OK).body(tarefa.get());
    }

    @PostMapping
    public ResponseEntity<Tarefa> createTarefa(@RequestBody
Tarefa tarefa) {
        Tarefa savedTarefa =
tarefaService.createTarefa(tarefa);
        return
ResponseEntity.status(HttpStatus.CREATED).body(savedTarefa);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Tarefa> updateTarefa(@PathVariable
Long id, @RequestBody Tarefa tarefa) {
        if (!tarefaService.getTarefaById(id).isPresent()) {
            return ResponseEntity.notFound().build();
        }

        tarefa.setId(id);
        Tarefa updatedTarefa =
tarefaService.createTarefa(tarefa);
        return ResponseEntity.ok(updatedTarefa);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Tarefa> deleteTarefa(@PathVariable
Long id) {
        if (!tarefaService.getTarefaById(id).isPresent()) {
            return ResponseEntity.notFound().build();
        }

        tarefaService.deleteTarefa(id);
```

```
            return ResponseEntity.noContent().build();
        }
    }
```

> 💡 | Dica
>
> Existe diferentes formas de realizar o mapeamento de requisições, você pode encontrar mais informações na documentação do Spring: https://docs.spring.io/.

Você deve ter notado que nossos métodos retornam uma `ResponseEntity<T>`. Essa classe representa toda a resposta HTTP e precisamos delas para manipular solicitações nos controladores. Na classe `TarefaController` temos os seguintes usos:

`ResponseEntity<List<Tarefa>> getAllTarefas()`, esse método retorna uma lista de tarefas. A resposta `ResponseEntity.status(HttpStatus.OK)` é usada para criar uma resposta com um código de status HTTP 200 (OK). O método `.body(listaTarefas)` define o corpo da resposta como a lista de tarefas.

No método `getTarefaById` se a tarefa não for encontrada, uma resposta com o código de status HTTP 404 (NOT FOUND) é retornada. Em `createTarefa`, você retorna uma resposta com o código de status HTTP 201 (CREATED) e o corpo contendo a tarefa criada. No método `deleteTarefa`, se a operação for concluída com sucesso, uma resposta com o código de status HTTP 204 (NO CONTENT) é retornada, indicando que não conteúdo para retornar.

A mesma lógica é usada para os métodos em `UsuarioController`.

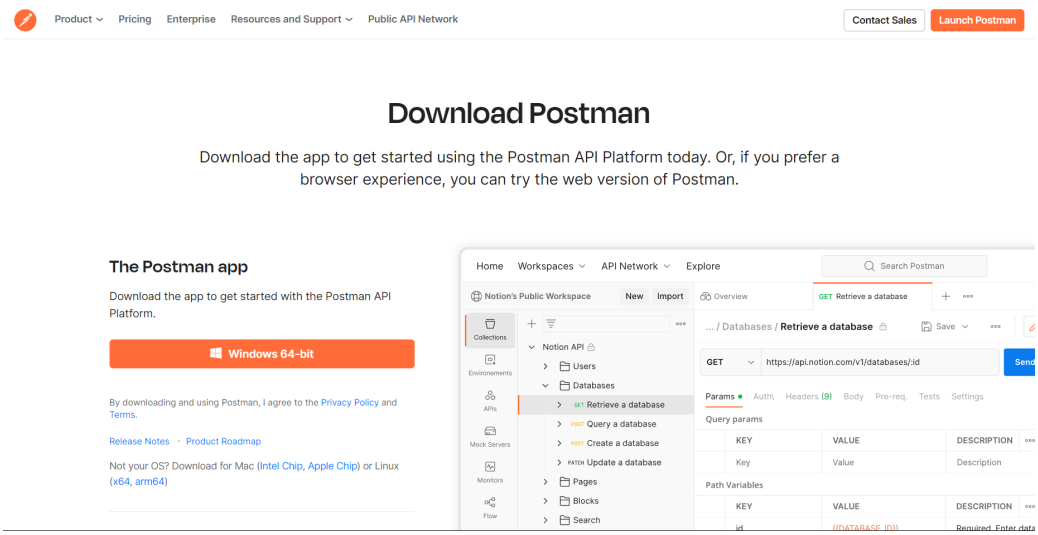
Parte 6

Testes no Postman

Após termos realizados todas as etapas anteriores de criação dos modelos (`Usuario` e `Tarefa`), repositórios (`UsuarioRepository` e `TarefaRepository`), serviços (`UsuarioService` e `TarefaService`) e controladores (`UsuarioController` e `TarefaController`), precisamos testar se todas as nossas requisições estão funcionando corretamente.

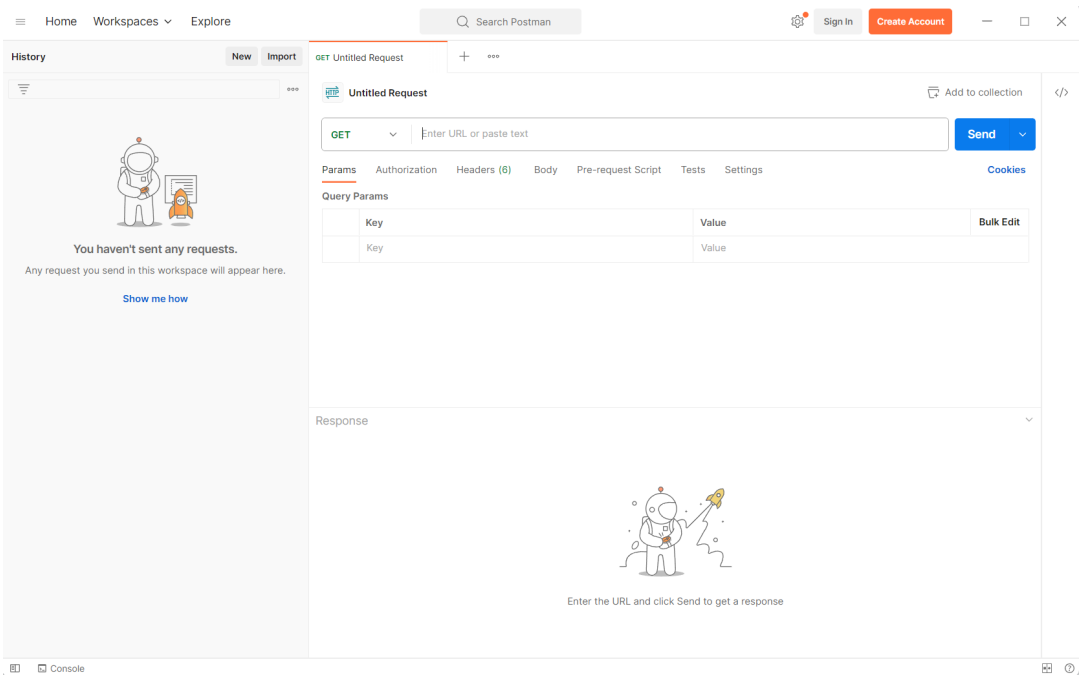
Para essa etapa, utilizaremos o Postman. O Postman, de forma bem simples, é uma ferramenta usada para testar, documentar e colaborar em APIs (Application Programming Interfaces). Ele oferece uma interface gráfica amigável que permite enviar solicitações HTTP para APIs e receber as respostas correspondentes. Desta forma, ela vai nos permitir realizar as requisições criadas nas nossas classes controller e irá nos exibir os resultados obtidos de cada uma das requisições (`GET`, `POST`, `PUT`, `DELETE`). Mas o que é necessário para utilizarmos o Postman?

O Postman pode ser utilizado tanto no navegador (https://web.postman.co/workspaces) como por meio de seu aplicativo encontrado para download a partir do seguinte link: https://www.postman.com/downloads/ . Para esse projeto, utilizaremos sua versão desktop (encontrada no link de download).



Página de download do Postman

Após a realização do download, basta abrir o executável da ferramenta e pôr a mão na massa, ou melhor, nas requisições.



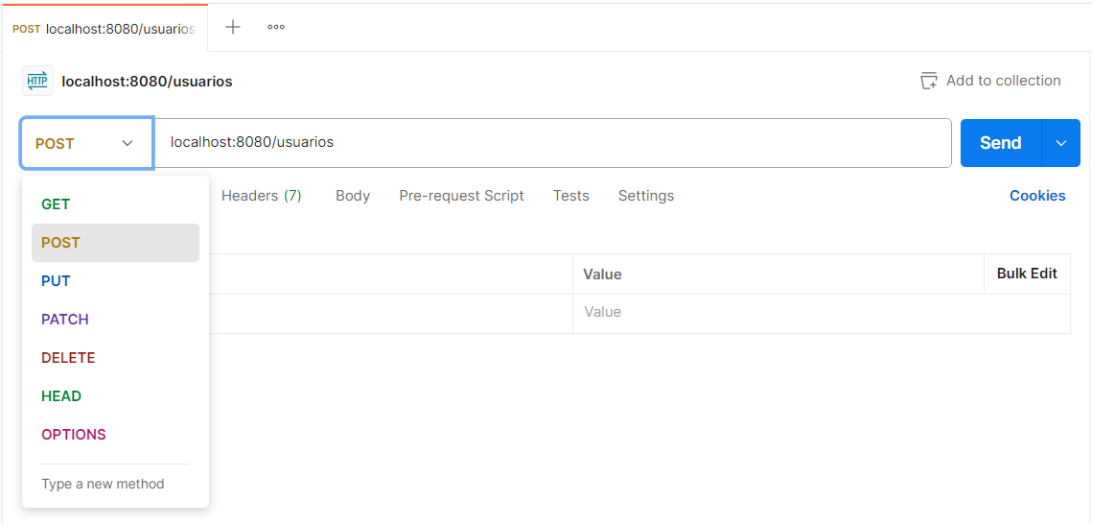
Página inicial do Postman Desktop

O Postman nos permite criar as requisições HTTP `GET`, `POST`, `PUT` e `DELETE`, e adicionar os paths (`api/tarefas`) e parâmetros (*api/tarefas/{id}*) para garantir o recurso (*[“id”: 1, “nome”: “Joao Carlos”, “tarefas”: [{“id”: 1, “descricao”: “Tarefa 1”}]]*) desejado.

Inicialmente, iremos criar a nossa requisição *POST*, já que todas as outras só serão viáveis após termos um Usuario ou uma Tarefa criada. Para essa requisição, iremos selecionar o método *POST* no nosso Postman e inserir a URL que consiste no tomcat (servidor web Java), que, por padrão, é 8080 (localhost:8080/) e o patch que corresponde ao o que queremos criar (*localhost:8080/usuarios*).

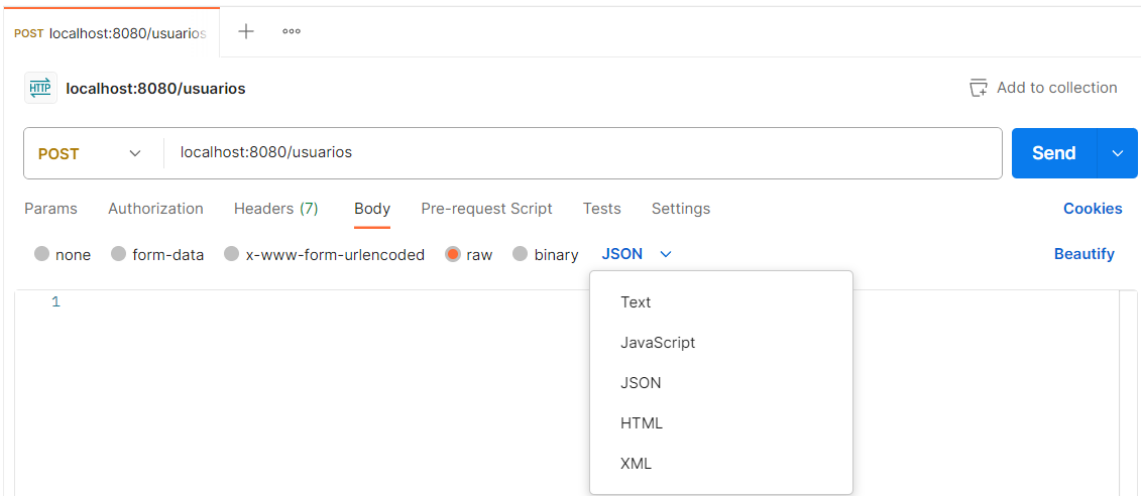
! | Nota

Lembrando que os paths foram estabelecidos nas nossas classes **controller**.



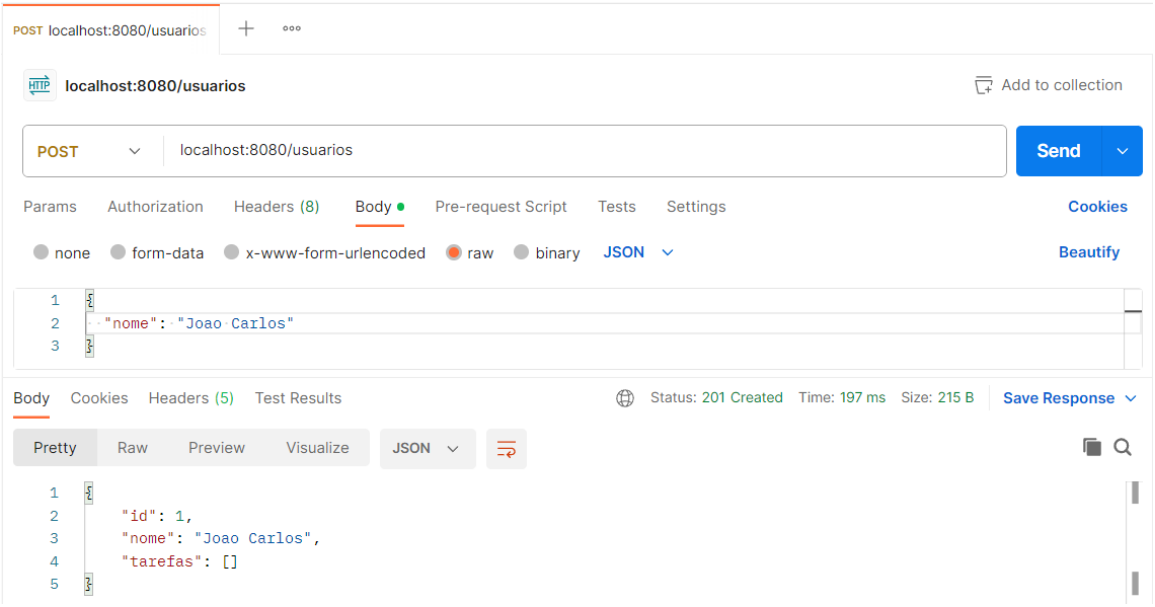
Seleção do método POST e inserção da requisição

Em continuidade, para que, finalmente, possamos criar nosso primeiro usuário, é necessário criar um Body (corpo que conterá as informações que são adicionadas ao nosso usuário criado) para a nossa requisição. Para isso, temos que selecionar o campo Body, marcar a opção raw e selecionar JSON na opção que tem como padrão o valor Text.



Configurações prévias da nossa requisição POST

Tendo feito todas essas configurações, devemos definir qual será o nosso Body. Ele deverá ter a estrutura padrão *JSON* chave-valor (*"id": 1*) e as informações que são requisitadas pela nossa entidade *Usuario*.



Criação do Body e do primeiro usuário

Como podemos ver, ao termos adicionado a nossa requisição, o *Body* e selecionado a opção *Send*, temos a criação do nosso primeiro usuário.

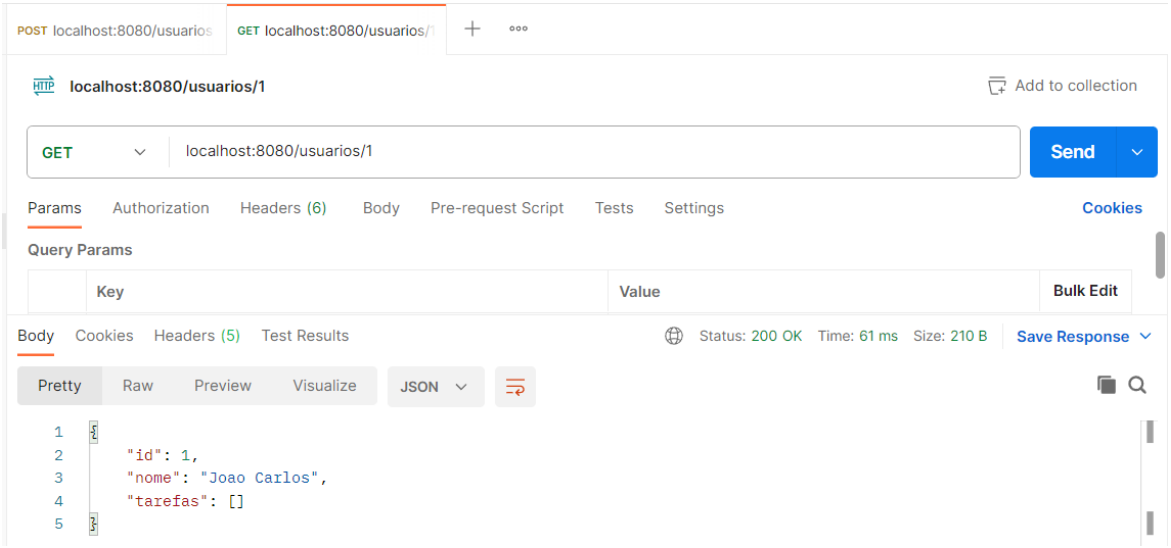
| Nota

É necessário que a nossa aplicação esteja rodando para que as requisições sejam realizada.

| Dica

Você pode executar o projeto usando o atalho F5 do teclado ou clicando na seta de play no canto superior direito da IDE (importante estar em algum dos arquivos .java ao tentar realizar a execução).

Tendo nosso primeiro usuário criado, podemos realizar as outras requisições que requisitam a sua criação. Portanto, agora iremos realizar o *GET* do usuário que acabamos de criar.

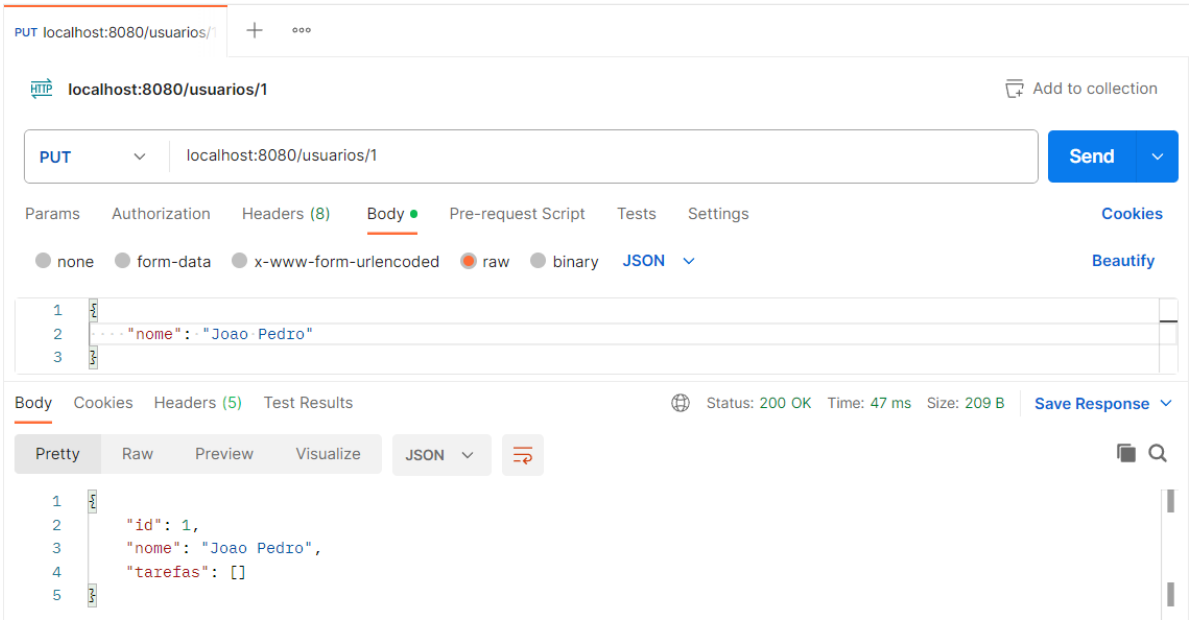


Método *GET* pelo id do usuário criado

Para esse método *GET*, utilizaremos a mesma requisição, mas teremos que adicionar o parâmetro (*{id}*) do usuário que foi criado para que ele seja retornado. Além disso, devemos selecionar a opção *GET* na nossa lista de métodos.

Em seguida, também podemos realizar o método *PUT* (atualizar) e *DELETE* (deletar) para o nosso usuário.

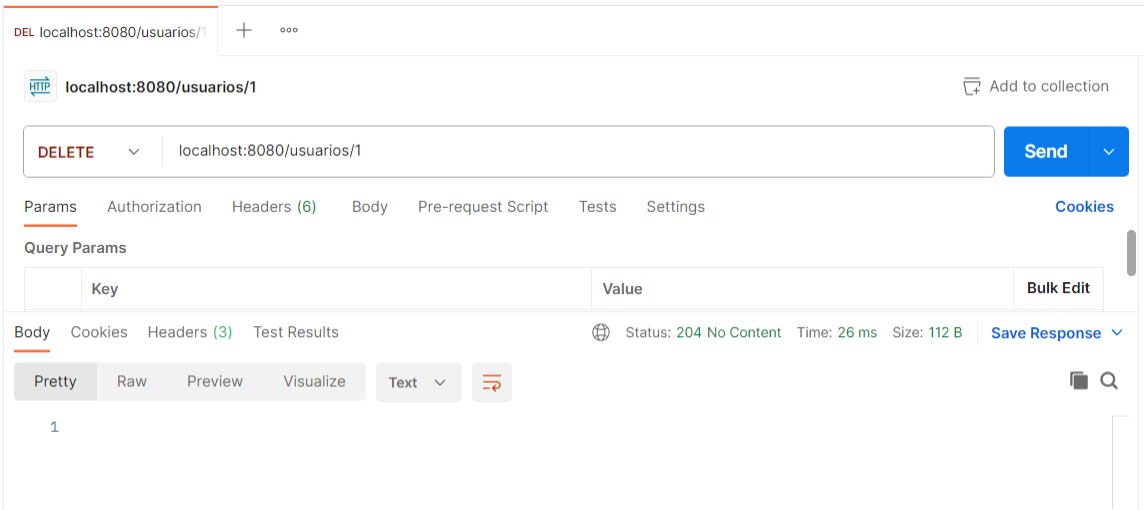
No caso do *PUT*, devemos, assim como no *GET*, utilizar um id como parâmetro para retornar um usuário específico e, assim como no *POST*, criar um Body que conterá as informações que serão atualizadas. Portanto, iremos construir o mesmo modelo utilizado no *POST*, realizar as alterações desejadas, selecionar o método *PUT* e selecionar *Send*.



Método *PUT* para atualização do usuário

Tendo feito as etapas, o nosso usuário terá a sua informação atualizada de acordo com o *Body* que foi passado.

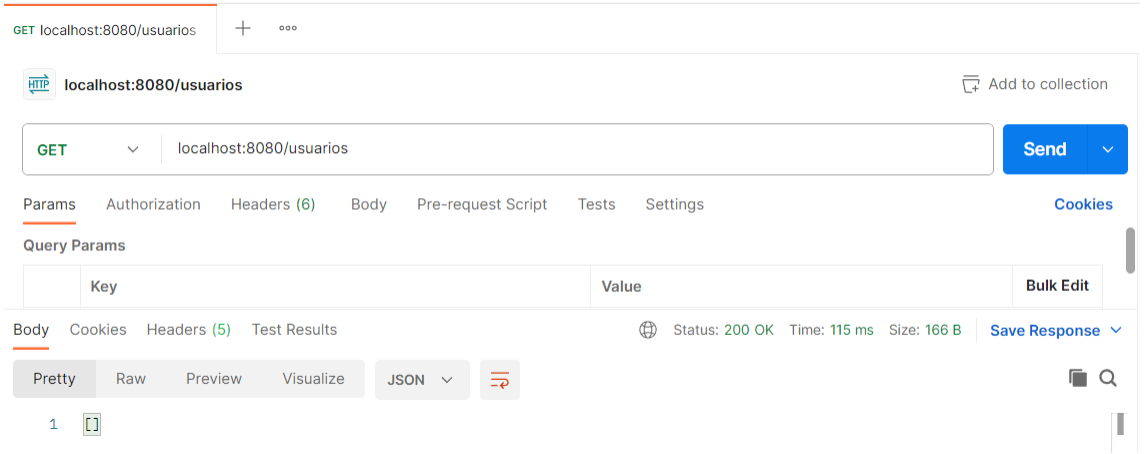
Por fim, podemos realizar o *DELETE* do nosso usuário por meio do seu id. Para isso, devemos selecionar a opção *DELETE* no nosso método, usar o mesma requisição utilizada na nossa requisição *POST* ou *PUT* e selecionar o *Send*. Após termos feito isso, nosso usuário será deletado e não teremos mais nenhum usuário.



Método *DELETE* para deleção do usuário

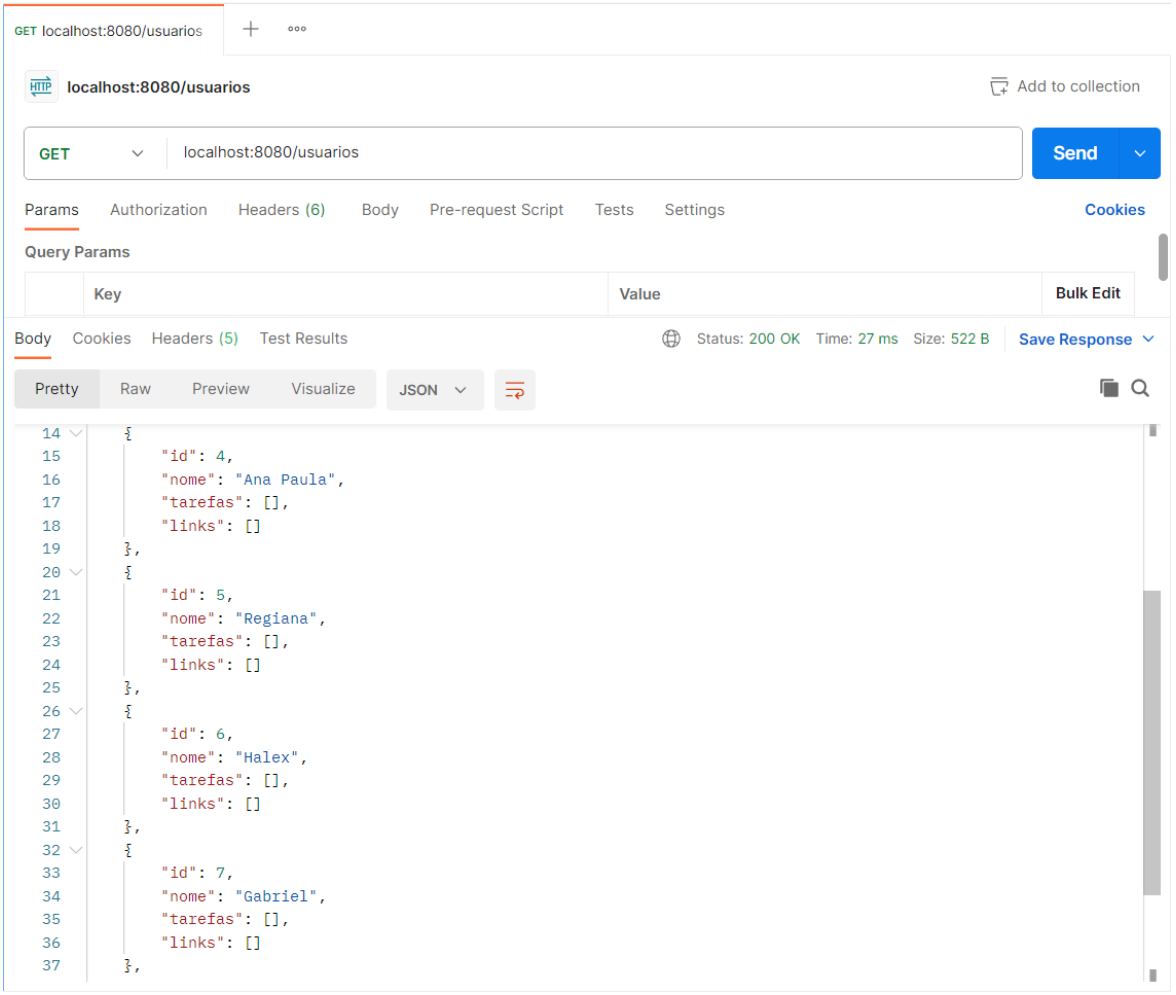
Como podemos observar, tivemos o retorno *204* (No Content) que indica que nosso usuário com id 1 foi deletado.

Para termos certeza disso, podemos realizar um *GET* para pegar todos os nossos usuários. Sendo assim, dessa vez, não precisaremos passar um parâmetro para a nossa requisição.



Método *GET* para requisitar todos os usuários

Ao realizar o método GET sem passar um parâmetro, podemos retornar todos os usuários criados, mas, como único criado foi deletado, o nosso recurso esperado está vazio.

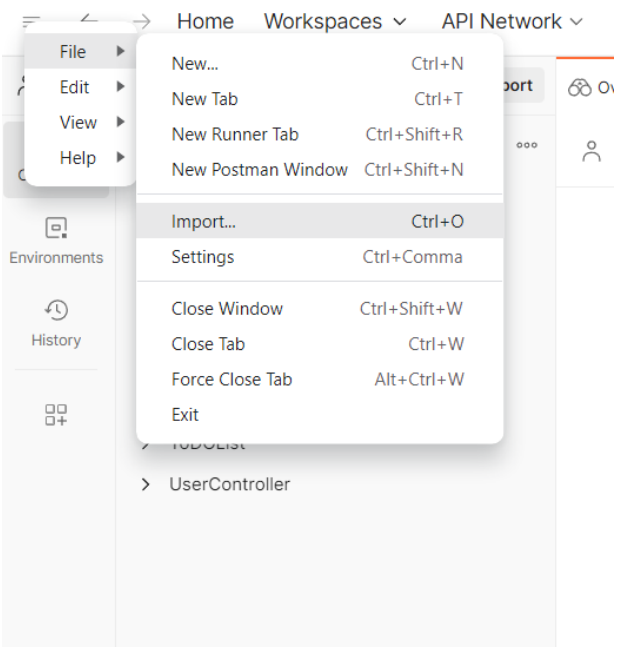


Exemplo de método *GET* de todos os usuários

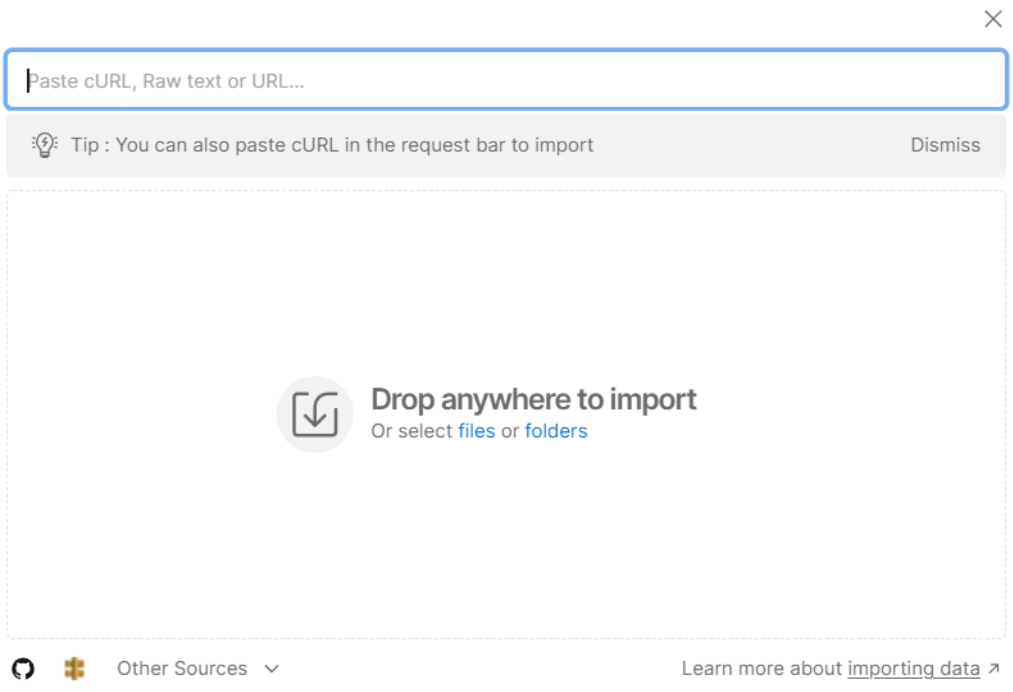
! | Nota

Fica como exercício de casa a realização dos testes com o Postman para as requisições da entidade Tarefa.

As requisições apresentadas estão presentes no repositório do projeto e podem ser importadas pelo Postman. Para realizar isso, é necessário a criação de uma conta e, após ter sido criada e o login ter sido realizado, você deve ir até campo arquivo (*file*) e depois em importar (*import…*). Tendo feito isso, apenas arraste ou selecione a opção arquivo (*file*) para abrir o arquivo de requisições do Postman.



Campo para importar as requisições de um arquivo Postman



Local para para arrastar ou abrir o arquivo

Testes Automatizados

Além dos testes realizados por meio da ferramenta Postman, também podemos construir algo mais sofisticado para verificar se nossas requisições estão funcionando como deveriam.

Os testes automatizados visam explorar cenários de sucesso e de erro, onde são realizadas requisições e é verificado se as requisições feitas seguem o que é estabelecido pela regra de negócio da nossa aplicação.

Para que isso seja possível, temos que importar algumas dependências adicionais, sendo elas o WebFlux e o Banco de Dados H2. O WebFlux irá nos oferecer suporte à programação reativa para construir aplicativos web escaláveis e eficientes. Além disso, com eles poderemos utilizar o WebTestClient que é um cliente de teste reativo (assíncrono) que nos permitirá chamar o endpoint de criação dos usuários ou das tarefas. Quando ao Banco de Dados H2, utilizaremos ele para a execução dos nossos testes, já que não é necessário nenhuma definição robusta para a sua utilização.

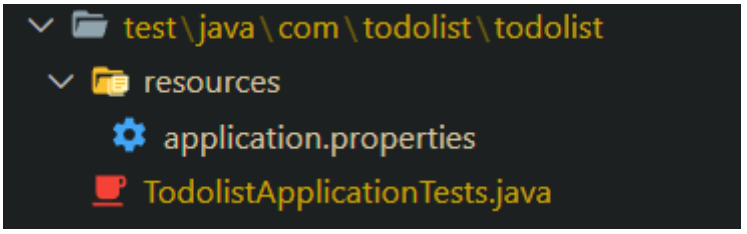
Portanto, inicialmente, vamos colocar as seguinte dependências no nosso arquivo pom.xml:

```
Unset
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
	<groupId>com.h2database</groupId>
	<artifactId>h2</artifactId>
	<scope>runtime</scope>
```

```
</dependency>
```

Após ter feito isso, iremos criar uma pasta chamada resources no diretório test\java\com\todolist\todolist (esse caminho pode variar dependendo de como você criou o projeto) e, nessa pasta, você irá criar um arquivo chamado application.properties. Nesse arquivo você irá colocar a configuração para o Banco de Dados H2. Lembrando que ele será utilizado apenas para os nossos testes automatizados.

```
Unset

spring.datasource.url=jdbc:h2:mem:todolist
```



Criação de pasta resources e arquivo application.properties

Tendo feito essas modificações, agora podemos começar a implementar nossos testes.

No arquivo TodolistApplicationTestes.java iremos iniciar a seguinte notação em cima do nome da classe: `@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)`. Esta notação indica que este é um teste de integração Spring Boot e solicita que o ambiente de teste use uma porta aleatória para evitar conflitos de porta com outros testes em execução. Após isso, devemos realizar a injeção de dependência do *WebTestClient* (utilizando o *@Autowired*), que, como já mencionado, permitirá termos acesso aos endpoints de criação de usuários e tarefas.

Portanto, até o momento, nossa implementação estará da seguinte forma:

Java

```java
package com.todolist.todolist;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.context.SpringBootTest;

import
org.springframework.boot.test.context.SpringBootTest.WebEnviro
nment;

import
org.springframework.test.web.reactive.server.WebTestClient;

import com.todolist.todolist.entity.Tarefa;

import com.todolist.todolist.entity.Usuario;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)

class TodolistApplicationTests {

    @Autowired

    private WebTestClient webTestClient;

    @Test

    void teste() {

    }

}
```

Em seguida, iremos renomear nosso método para *testeCriarUsuario*, mas você pode usar o nome que quiser. Já que queremos fazer o *POST* de um usuário, precisamos criar um objeto do tipo *Usuario* para ser utilizado no teste. Por fim, iremos colocar a seguinte informações:

Java

```
webTestClient:
    • post().uri("/usuarios"): Envia uma solicitação POST
      para o endpoint "/usuarios".
    • bodyValue(usuario): Define o corpo da solicitação
      com o objeto de usuário criado.
    • exchange(): Realiza a troca da solicitação e obtém a
      resposta.
    • expectStatus().isCreated(): Espera que o status da
      resposta seja 201 (Created).
expectBody():
    • jsonPath("$").isMap(): Espera que o corpo da
      resposta seja um objeto JSON.
    • jsonPath("$.length()").isEqualTo(3): Espera que o
      objeto tenha 3 campos.
    • jsonPath("$.nome").isEqualTo(usuario.getNome()):
      Espera que o campo "nome" seja igual ao nome do
      usuário criado.
```

Portanto, após colocar essas informações, o nosso teste para criação de um usuário está completo.

Java

```java
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class TodolistApplicationTests {

    @Autowired
    private WebTestClient webTestClient;

    @Test
    void testeCriarUsuario() {
        var usuario = new Usuario("Eduardo");
        webTestClient
            .post()
            .uri("/usuarios")
            .bodyValue(usuario)
            .exchange()
            .expectStatus().isCreated()
            .expectBody()
            .jsonPath("$").isMap()
            .jsonPath("$.length()").isEqualTo(3)
            .jsonPath("$.nome").isEqualTo(usuario.getNome());
    }
}
```

Para executar o tempo, é necessário apenas usar o atalho F5 do teclado ou usando o botão de play no canto superior direito. Ao executar, teremos o retorno de sucesso

no teste ou de fracasso, caso alguma regra não tenha sido seguida na criação do objeto.

O mesmo processo ocorre para a criação de testes para a entidade tarefa, portanto, fica como dever de casa para você a criação de outros possíveis testes.

Até este ponto, nossa API desenvolvida com o Spring Boot já abrange diversas funcionalidades, desde a configuração inicial até a criação de entidades, serviços e controladores para gerenciar tarefas e usuários. No entanto, uma abordagem que ainda não exploramos é o HATEOAS, e é exatamente isso que abordaremos a seguir.

Parte 6

Implementação do HATEOAS

O HATEOAS (Hypermedia as the Engine of Application State) desempenha um papel crucial no contexto da maturidade das APIs RESTful, sendo geralmente associado ao Nível 3 no modelo Richardson Maturity Model. Este modelo foi proposto por Leonard Richardson para avaliar o grau de aderência aos princípios REST em uma API. Nesse nível, a API fornece links dinâmicos nos recursos que informam aos clientes quais ações podem ser executadas a partir de um determinado estado, isso ajuda os clientes a consumirem uma API sem a necessidade de conhecimento prévio e sem esquentar a cabeça lendo documentação.

Implementaremos o HATEOAS em nossa API usando a dependência do Spring Boot, já adicionada no início do projeto com o Spring Initializr. Essa dependência fornece a classe base `RepresentationModel<T>` para ajudar na criação de recursos HATEOAS, só precisamos especificar o tipo de recurso que será representado (`Tarefa` e `Usuario` na nossa API).

Java

```
@Entity
public class Usuario extends RepresentationModel<Usuario>{
...
}
```

Java

```
@Entity
public class Tarefa extends RepresentationModel<Tarefa>{
...
}
```

Agora precisamos adicionar os links ao recursos retornados pelos métodos `GET` da nossa aplicação. Esses métodos estão definidos nas classes `Controller`. Faremos então as seguintes modificações para incluir os links HATEOAS:

```java
@RestController
@RequestMapping("/usuarios")
public class UsuarioController {

    @Autowired
    UsuarioService usuarioService;

    @GetMapping
    public ResponseEntity<List<Usuario>> getAllUsuarios() {
        List<Usuario> listaUsuarios =
usuarioService.getAllUsuarios();
        if (!listaUsuarios.isEmpty()) {
            for (Usuario usuario : listaUsuarios) {
                Long id = usuario.getId();

usuario.add(linkTo(methodOn(UsuarioController.class).getUsuari
oById(id)).withSelfRel());
            }
        }
        return
ResponseEntity.status(HttpStatus.OK).body(listaUsuarios);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Usuario>
getUsuarioById(@PathVariable Long id) {
        Optional<Usuario> usuario =
usuarioService.getUsuarioById(id);
        if(usuario.isEmpty()){
            return
ResponseEntity.status(HttpStatus.NOT_FOUND).body(usuario.get()
);
        }
```

```
usuario.get().add(linkTo(methodOn(UsuarioController.class).get
AllUsuarios()).withRel("Lista de usuarios"));
        return
ResponseEntity.status(HttpStatus.OK).body(usuario.get());

    }
}
```

No método `getAllUsuarios`, após obter a lista de usuários, iteramos sobre ela e adicionamos um link HATEOAS para cada usuário. O link criado é um link (`linkTo`) para o método `getUsuarioById` do `UsuarioController` (`methodOn`). Este link é associado ao conceito de "self" (auto-relacionamento), indicando que o link leva ao próprio recurso (`withSelfRel`). Isso é um princípio fundamental do HATEOAS.

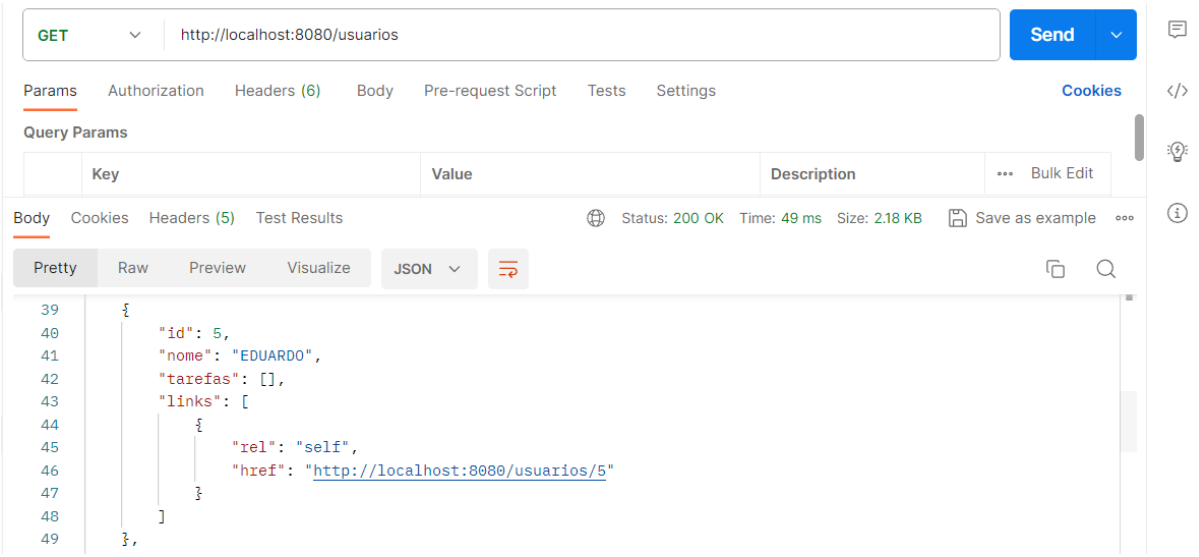
O mesmo padrão é aplicado ao método `getUsuarioById`, onde, além de adicionar um link "self" para o recurso individual, você também adiciona um link para a lista completa de usuários, fornecendo uma forma de navegar entre os recursos.

Vamos repetir essa prática nos métodos correspondentes do `TarefaController` para as entidades `Tarefa`:

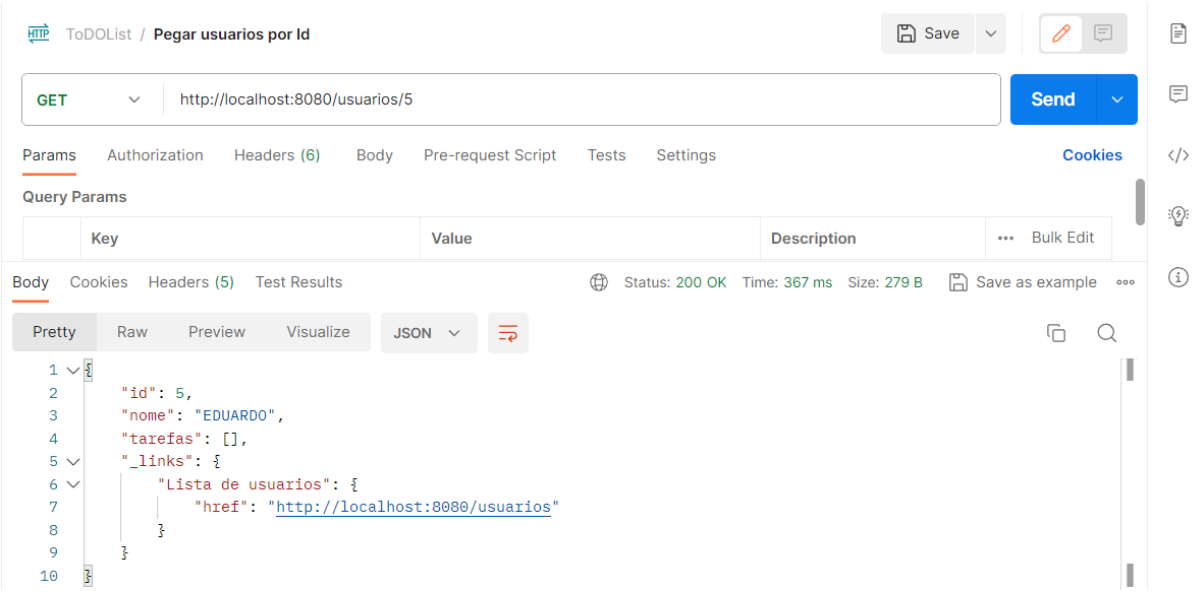
Java
```
@RestController
@RequestMapping("/tarefas")
public class TarefaController {

    @Autowired
    private TarefaService tarefaService;
    @GetMapping
    public ResponseEntity<List<Tarefa>> getAllTarefas() {
        List<Tarefa> listaTarefas =
tarefaService.getAllTarefas();
        if (!listaTarefas.isEmpty()) {
            for (Tarefa tarefa : listaTarefas) {
                Long id = tarefa.getId();
```

```
tarefa.add(linkTo(methodOn(TarefaController.class).getTarefaBy
Id(id)).withSelfRel());
            }
        }
        return
ResponseEntity.status(HttpStatus.OK).body(listaTarefas);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Tarefa> getTarefaById(@PathVariable
Long id) {
        Optional<Tarefa> tarefa =
tarefaService.getTarefaById(id);
        if(tarefa.isEmpty()){
            return
ResponseEntity.status(HttpStatus.NOT_FOUND).body(tarefa.get())
;
        }

tarefa.get().add(linkTo(methodOn(TarefaController.class).getAl
lTarefas()).withRel("Lista de tarefas"));
        return
ResponseEntity.status(HttpStatus.OK).body(tarefa.get());
    }
}
```

Assim, quando alguém consultar um recurso, pode encontrar links para outros recursos relacionados:



Teste no Postman do método `getAllUsuarios`



Teste no Postman do método `getUsuarioById`

Ao incorporar links HATEOAS aos recursos de nossa API, alcançamos o nível 3 de maturidade na arquitetura REST. Agora você aprendeu o básico para desenvolver uma API RESTFul com Spring Boot! Todo código desenvolvido está disponível no repositório no Github. Sinta-se à vontade para explorar mais recursos e personalizar seus projetos conforme necessário. Boa codificação! 🚀