



# Guia Passo a Passo para Desenvolvimento de uma API com Spring Boot

## 1. Introdução ao Spring Boot

O Spring Boot é um framework para desenvolvimento de aplicações Java que visa facilitar e agilizar o processo de criação e configuração de aplicações com o ecossistema Spring. Ele é baseado no Spring Framework e oferece diversas funcionalidades que ajudam os desenvolvedores a criar aplicações robustas e escaláveis de forma mais rápida e simples. Este guia fornecerá um passo a passo para a configuração e desenvolvimento de uma API utilizando o Spring Boot.

## 2. Pré-Requisitos

Antes de iniciar o desenvolvimento da API, é necessário instalar alguns softwares essenciais:

- [Visual Studio Code](#)
- [Banco de dados MySQL](#)
- Google Chrome (ou outro navegador)

## 3. Configuração do Ambiente de Desenvolvimento

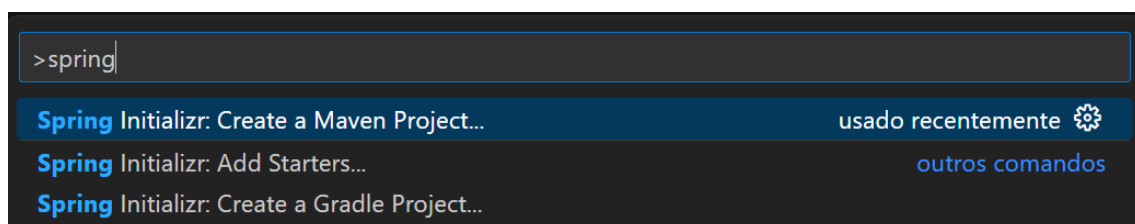
Na configuração do ambiente de desenvolvimento, será necessário adicionar duas extensões: uma para o Java e outra para o Spring Boot.

- [Extension Pack for Java](#)
- [Spring Boot Extension Pack](#)

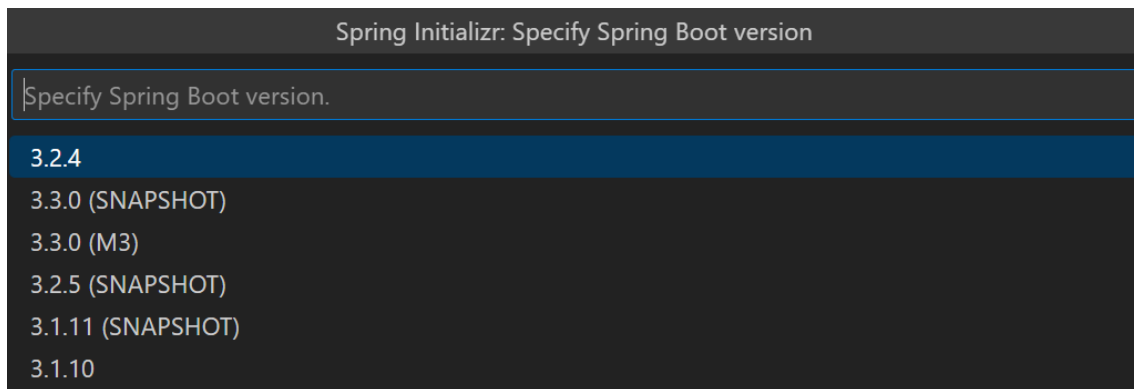
## 4. Criando o primeiro projeto

Com o Visual Studio Code aberto, acesse a paleta de comandos seguindo os passos: no menu superior, localize e clique em "Ver" -> "Paleta de Comandos". Se o Visual Studio estiver em inglês, procure por "View" -> "Command Palette".

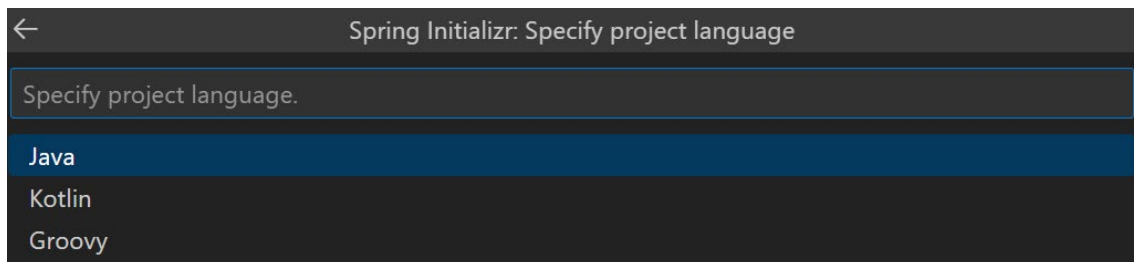
O próximo passo é inserir o termo "Spring" na paleta de comandos e selecionar a opção "Spring Initializr: Create a Maven Project...". A imagem abaixo ilustra como deve estar o seu Visual Studio Code neste momento:



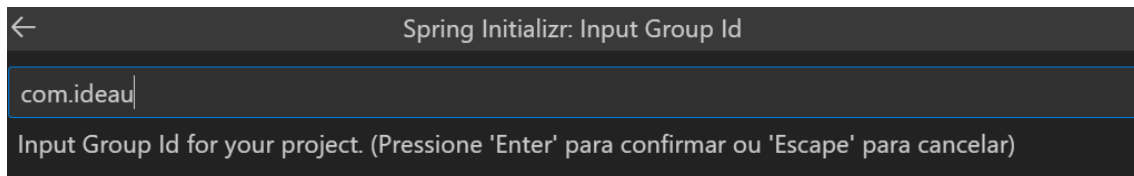
Agora é necessário especificar a versão do Spring Boot. Neste tutorial, estamos utilizando a versão 3.2.4. Recomendo fortemente deixar selecionada a primeira opção da lista de versões, pois é a versão mais estável do Spring Boot.

A screenshot of the 'Spring Initializr: Specify Spring Boot version' screen. It features a dark-themed interface with a title bar at the top. Below the title bar is a text input field containing the placeholder text 'Specify Spring Boot version.'. Underneath the input field is a list of version options: '3.2.4' (highlighted in blue), '3.3.0 (SNAPSHOT)', '3.3.0 (M3)', '3.2.5 (SNAPSHOT)', '3.1.11 (SNAPSHOT)', and '3.1.10'.

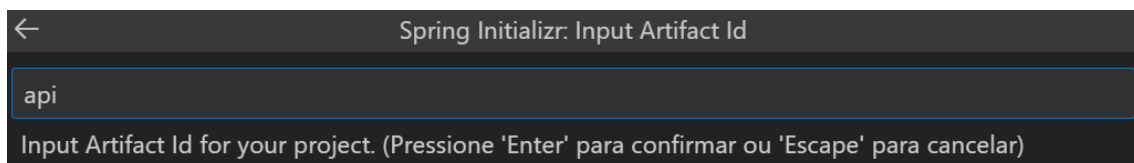
Você pode escolher o tipo de linguagem para desenvolver com o Spring. As opções disponíveis são: Java, Kotlin e Groovy. Selecione a opção "Java" e prossiga com o tutorial:

A screenshot of the 'Spring Initializr: Specify project language' screen. It has a dark theme and a title bar. Below the title bar is a text input field with the placeholder 'Specify project language.'. Below the input field is a list of language options: 'Java' (highlighted in blue), 'Kotlin', and 'Groovy'.

No próximo passo, é necessário definir o agrupamento para o projeto. A Sun Microsystems, criadora do Java, estabeleceu um padrão para facilitar a organização de arquivos em projetos Java. De acordo com as boas práticas, é recomendado utilizar o prefixo do domínio da empresa. Vamos utilizar como exemplo o domínio "com.ideal":

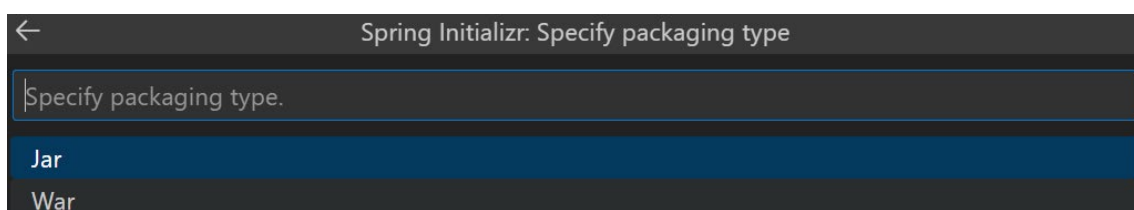
A screenshot of the 'Spring Initializr: Input Group Id' screen. It has a dark theme and a title bar. Below the title bar is a text input field containing the text 'com.ideal'. Below the input field is a prompt: 'Input Group Id for your project. (Pressione 'Enter' para confirmar ou 'Escape' para cancelar)'.

Neste passo, vamos especificar o ID do projeto, que basicamente é o seu nome. Eu utilizei o nome "api", mas sinta-se à vontade para escolher outro nome, se preferir:

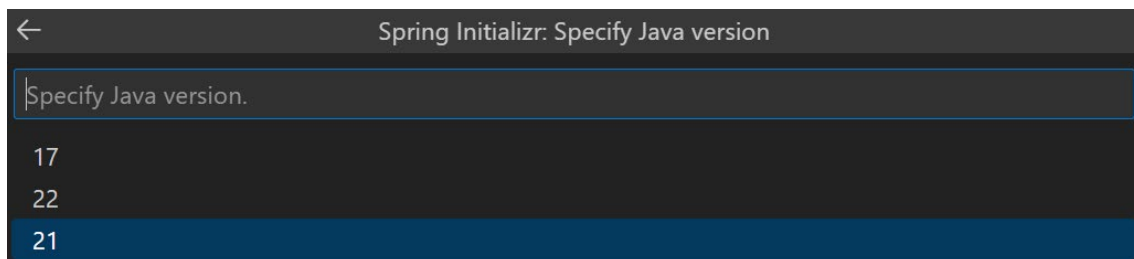
A screenshot of the 'Spring Initializr: Input Artifact Id' screen. It has a dark theme and a title bar. Below the title bar is a text input field containing the text 'api'. Below the input field is a prompt: 'Input Artifact Id for your project. (Pressione 'Enter' para confirmar ou 'Escape' para cancelar)'.

No próximo passo, é necessário especificar o tipo de empacotamento para o projeto. Vamos utilizar o formato "Jar". Abaixo, contém uma breve explicação sobre os tipos "JAR" e "WAR":

- **Jar:** Contém todas as classes fundamentais para trabalhar com a linguagem Java.
- **War:** Contém as classes fundamentais para trabalhar com Java, além de oferecer suporte para manipular servlets e arquivos JSP.

A screenshot of the 'Spring Initializr: Specify packaging type' screen. It has a dark theme and a title bar. Below the title bar is a text input field with the placeholder 'Specify packaging type.'. Below the input field is a list of packaging options: 'Jar' (highlighted in blue) and 'War'.

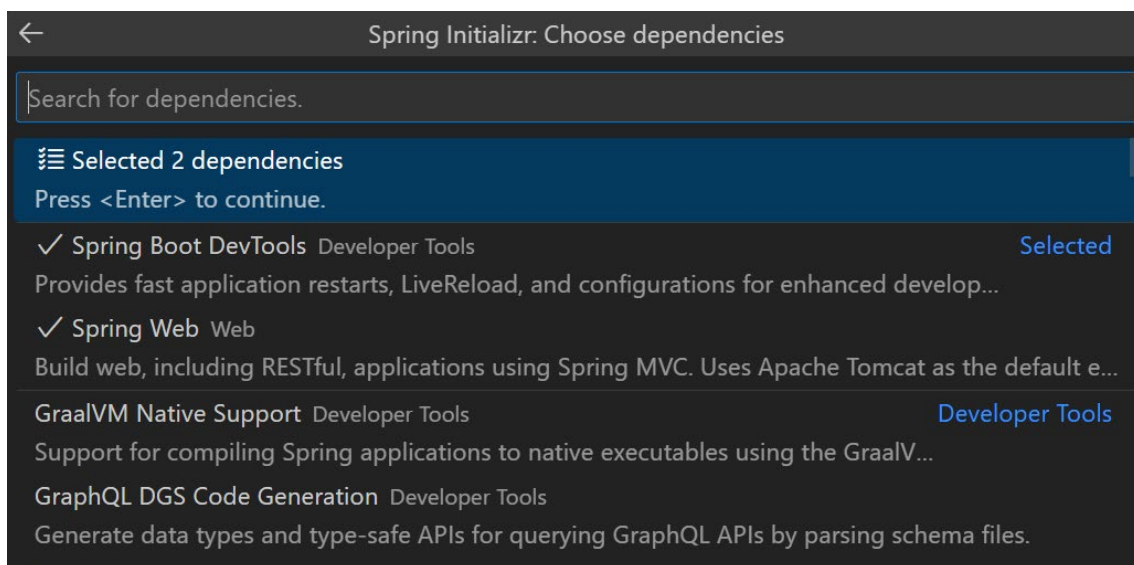
Será necessário informar a versão do Java. Recomendo fortemente utilizar a mesma versão instalada em seu computador. Neste tutorial, estava usando o JDK 21. Portanto, a versão escolhida para gerar o projeto também foi a 21. Se o projeto for gerado em uma versão diferente do JDK, pode resultar em erros de compilação.



Na nossa última etapa, é necessário selecionar as dependências que especificam as funcionalidades que nossa aplicação terá. Selecione as seguintes:

- **Spring Boot DevTools:** Fornece uma configuração básica do Spring e atualiza o servidor automaticamente sempre que algum arquivo for alterado.
- **Spring Web:** Responsável por criar aplicações RESTful utilizando o Spring MVC.

Após selecionar as duas dependências, clique na opção "Selected 2 dependencies". Aguarde um momento e, em seguida, abra o projeto criado.



A estrutura de projetos em Spring Boot é baseada nos seguintes arquivos e pastas:

**.mvn:** Esta pasta contém as configurações para o funcionamento do Maven. É importante ressaltar que o Maven é um gerenciador de dependências. Com ele, podemos gerenciar diversos complementos que o Spring pode utilizar nos projetos, como Spring Security, conexões com bancos de dados, utilização de JPA (ORM), estrutura para APIs, JWT para trabalhar com tokens, entre outras dependências.

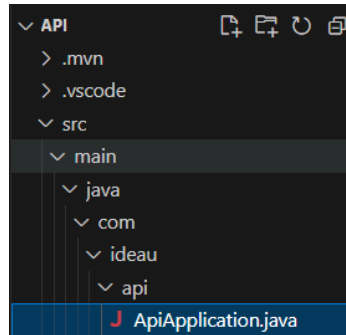
**src:** Nesta pasta, você irá desenvolver toda a estrutura do projeto. Além do código-fonte, você pode realizar algumas configurações no arquivo "application.properties", como conexões com banco de dados, alterar a porta de funcionamento, habilitar log de ações, entre outras funcionalidades.

**mvnw e mvnw.cmd:** São arquivos de configuração do Maven.

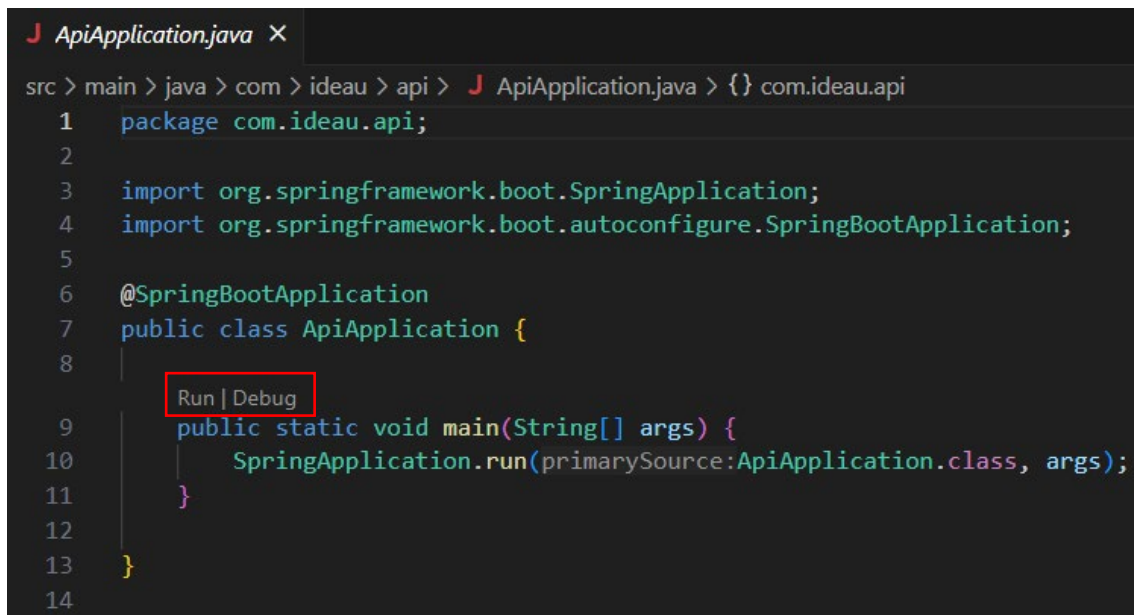
**pom.xml:** É considerado o "cérebro" do nosso projeto, pois contém todo o conhecimento necessário para realizarmos diversas funcionalidades em nossos projetos. Aqui, você pode especificar conexões com bancos de dados, configuração de servidor, segurança de aplicações, envio de e-mails, entre outras ações. É neste arquivo que iremos definir todas as dependências que desejamos utilizar nos projetos.

## 5. Executando o projeto.

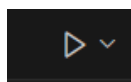
Com o projeto aberto, localize e selecione o arquivo **"ApiApplication.java"**:



Com o arquivo **"ApiApplication.java"** aberto, observe que antes do método **public static void main(String[] args) { }**, existem dois ícones ou links: um para compilar o projeto e outro para iniciar a depuração:



Outra opção para compilar ou depurar o projeto é utilizar o ícone de execução localizado na parte superior direita da interface, conforme ilustrado na imagem:



Para acessar a aplicação, abra o navegador e digite: localhost:8080. Você receberá um erro neste momento, mas vamos corrigir isso nas próximas seções:

## Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Mar 31 22:22:44 BRT 2024

There was an unexpected error (type=Not Found, status=404).

No static resource api.

org.springframework.web.servlet.resource.NoResourceFoundException: No static resource api.

at org.springframework.web.servlet.resource.ResourceHttpRequestHandler.handleRequest(ResourceHttpRequestHandler.java:585)

### 6. Implementando o Controlador (Controller)

#### 6.1. O Que é um Controlador em uma Aplicação Spring?

Um controlador tem como finalidade principal estabelecer rotas (endpoints) na sua aplicação. Isso é exemplificado por uma URL como: `http://localhost/spring`. Observe que o termo “**spring**” está destacado, indicando o nome da rota associada ao controlador.

#### 6.2. Importância das Requisições ao Criar um Controlador

Ao desenvolver um controlador, é essencial compreender como utilizar corretamente os diferentes tipos de requisições HTTP. Em uma API, é possível ter múltiplas rotas com o mesmo nome, mas que realizam ações distintas. Abaixo estão alguns exemplos dos tipos de requisições mais comuns e suas respectivas ações:

- **POST:** Utilizado para cadastrar ou enviar novos dados à aplicação.
- **GET:** Utilizado para recuperar ou listar informações existentes na aplicação.
- **PUT** ou **UPDATE:** Utilizado para modificar dados já existentes na aplicação.
- **DELETE:** Utilizado para remover dados da aplicação. Há outros tipos de requisições, porém vamos focar nesses quatro, pois são os mais utilizados em projetos que envolvam APIs.

### 7. Annotations: O que são e como utilizá-las

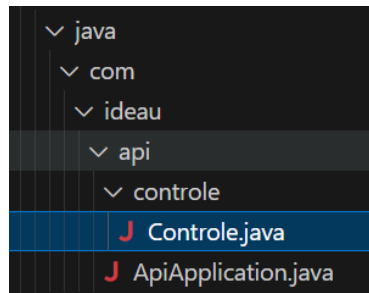
As annotations são utilizadas para adicionar funcionalidades extras a classes, atributos ou métodos em Java. Se você já tem experiência com Java ou C#, provavelmente já se deparou com a annotation **@Override**. Abaixo, apresentamos algumas annotations que serão utilizadas nesta etapa da aula:

- **@RestController:** Indica que a classe atuará como um controlador (controller) em uma aplicação Spring Boot.
- **@GetMapping:** Especifica que o método será associado a uma rota HTTP GET, permitindo a exibição de dados na aplicação.

#### 7.1. Implementando o projeto.

Agora que entendemos o conceito de controlador e as annotations relevantes, podemos prosseguir com a implementação do nosso projeto. Siga os passos abaixo para criar a estrutura de pastas e arquivos necessários:

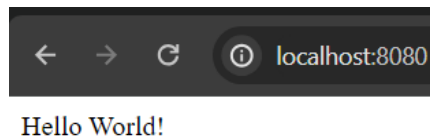
1. Clique com o botão direito do mouse na pasta **api** do seu projeto.
2. Selecione a opção para **criar uma nova pasta** e nomeie-a como **controle**.
3. Dentro da pasta controle, **crie um novo arquivo Java** e nomeie-o como **Controle.java**.



Com o novo arquivo Controle.java criado, vamos implementar nossa primeira rota que retornará a mensagem **“Hello World!”**. Siga as instruções abaixo para adicionar o código necessário:

```
1 @RestController
2 public class Controle {
3
4     @GetMapping("")
5     public String mensagem() {
6         return "Hello World!";
7     }
8 }
```

Vamos testar? Inicie a nossa aplicação, abra o navegador e acesse o endereço “localhost:8080” para verificar o resultado:



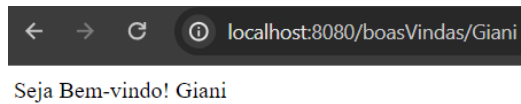
## 8. @PathVariable

A annotation “**@PathVariable**” é utilizada para recuperar dados de uma URL. Por exemplo, considerando a seguinte rota: “**http://localhost/curso/200**”, onde “curso” é o nome da rota e “**200**” é o valor que queremos extrair usando **@PathVariable**.

Sempre que precisar recuperar dados de uma URL, esta é uma das abordagens mais comuns no Spring para realizar essa tarefa. No arquivo de controle que criamos no capítulo anterior, implementaremos a seguinte rota:

```
1 @GetMapping("/boasVindas/{nome}")
2 public String boasVindas(@PathVariable String nome){
3     return "Seja Bem-vindo!" + nome;
4 }
```

Agora, você pode testar acessando o seguinte endereço no seu navegador: localhost:8080/boasVindas/Giani. O nome "Giani" pode ser substituído por qualquer outro. Esta rota irá concatenar uma mensagem com o nome fornecido na URL:



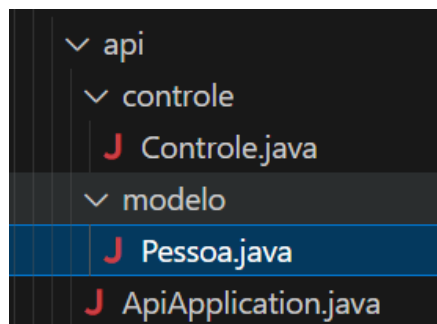
## 9. Modelo

Um modelo é uma representação estruturada de dados que desempenha duas funções principais: a manipulação de dados e a definição de tabelas (que exploraremos mais adiante).

Ao lidar com uma API, frequentemente temos grandes volumes de dados para manipular. Como facilitar essa transição de dados? Através de um objeto, que será instanciado com base em um modelo. Qualquer dado que você queira receber ou enviar em uma API que não seja através de uma URL deve ser representado por um modelo, para que o Spring possa compreender como lidar com essas informações.

As APIs são capazes de integrar dados com outras APIs ou com uma estrutura front-end usando o formato **JSON (JavaScript Object Notation)**. Embora em projetos mais antigos fosse comum utilizar **XML (eXtensible Markup Language)**, o JSON é geralmente mais fácil de manipular e oferece um processamento mais eficiente na maioria das situações.

Antes de começarmos a codificar nosso projeto, crie uma pasta chamada "modelo". Dentro desta pasta, vamos criar um arquivo chamado "**Pessoa.java**":



Com a estrutura devidamente organizada, podemos agora codificar nosso modelo. Inicialmente, um modelo é composto por atributos e pelos métodos **getters** e **setters**. Nas próximas etapas, iremos implementar essa parte:

```
1 public class Pessoa {
2
3     // Atributos da classe
4     private int codigo;
5     private String nome;
6     private int idade;
7
8     public int getCodigo() {
9         return codigo;
10    }
11    public void setCodigo(int codigo) {
12        this.codigo = codigo;
13    }
14    public String getNome() {
15        return nome;
16    }
17    public void setNome(String nome) {
18        this.nome = nome;
19    }
20 }
```

## 10. Vinculado controle e modelo

Vamos simplesmente criar uma rota para obter um objeto com as características do modelo e, em seguida, retornar esse objeto.

Estaremos introduzindo duas novas annotations, e abaixo é detalhado o funcionamento de cada uma:

- **@PostMapping**: Annotation utilizada para definir uma rota com suporte ao método POST.
- **@RequestBody**: Annotation responsável por receber dados enviados por outras APIs ou pelo front-end. Geralmente, esses dados estão estruturados em JSON e correspondem às características do modelo.

Abra o arquivo “**Controle.java**” e adicione a seguinte rota:

Obs.: Não esqueça de importar as annotations **@PostMapping** e **@RequestBody**. Com isso, estabelecemos a conexão entre nosso controlador e o modelo.

```
1 @PostMapping("/pessoa")
2 public Pessoa pessoa(@RequestBody Pessoa p){
3     return p;
4 }
```

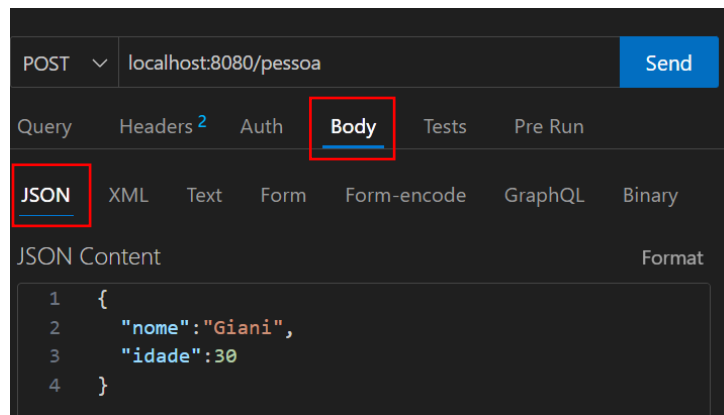
## 11. Testando APIs

Continuando o nosso tutorial sobre como criar APIs com Spring Boot, vamos aprender a realizar testes no nosso projeto. Utilizaremos uma extensão do Visual Studio Code chamada “**Thunder**

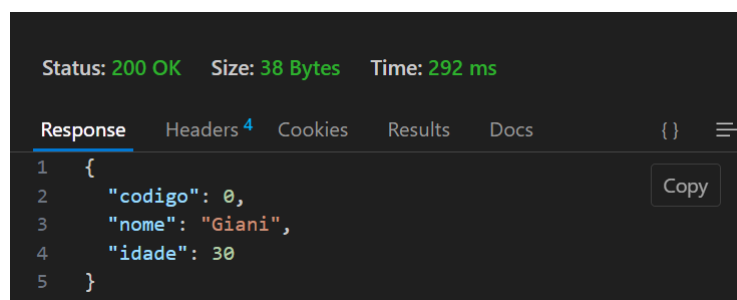


**Client**". Você pode acessar o site da extensão [clique aqui](#) ou instalá-la diretamente no Visual Studio Code.

Após instalar a extensão, localize a lateral esquerda do Visual Studio Code e selecione "**Thunder Client**". Em seguida, especifique a seguinte requisição:



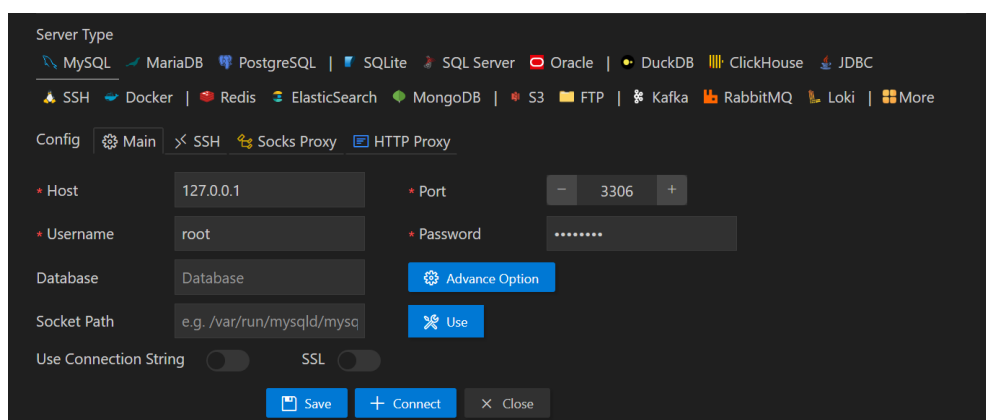
Observe que na imagem acima, o tipo de requisição foi alterado para **POST** e também foi criado um JSON com os campos “nome” e “idade”. Ao clicar em “**Send**”, o seguinte resultado será exibido:



Temos acesso ao status da requisição, que neste caso foi 200 OK, ao tamanho do retorno e ao tempo de processamento. No campo "Response", é possível observar que foi retornado um JSON com as mesmas características informadas no envio da requisição.

## 12. Configurando banco de dados MySQL

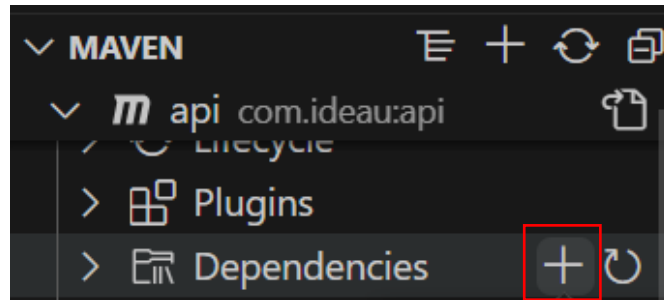
Utilizarei uma extensão do Visual Studio Code chamada "MySQL" para administrar o banco de dados. Se desejar mais informações, [clique aqui](#) para acessar o marketplace do Visual Studio Code e obter mais detalhes sobre a extensão.



### 13. Dependências

Vamos adicionar duas novas dependências para estabelecer a conexão com o banco de dados e criar as tabelas. Vejamos quais dependências serão utilizadas:

- **JPA (Java Persistence API):** Responsável por criar a estrutura de tabelas no banco de dados e interagir com os objetos recebidos e enviados.
- **MySQL:** Dependência responsável por facilitar a conexão entre uma aplicação Spring e o banco de dados MySQL.



As duas dependências que devem ser adicionadas ao arquivo pom.xml são:

- mysql-connector-java
- spring-boot-starter-data-jpa

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
  <version>3.2.4</version>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.33</version>
</dependency>
```

### 14. Gerando Tabelas

Vamos aprender a utilizar algumas annotations para gerar as tabelas. Abaixo estão as annotations que serão utilizadas:

- **@Entity:** Responsável por criar a tabela.
- **@Table:** Utilizada quando se deseja especificar características da tabela, como o nome, por exemplo.
- **@Id:** Define uma chave primária.
- **@GeneratedValue:** Implementa o auto incremento.

Para implementar essas annotations, precisaremos abrir o arquivo do modelo e adicioná-las conforme necessário:

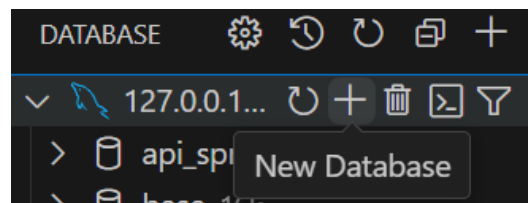
```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "pessoas")
public class Pessoa {

    // Atributos da classe
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int codigo;
    private String nome;
    private int idade;
```

## 15. Conexão com o Banco de Dados MySQL:

Inicialmente, acesse a extensão do banco de dados e crie um novo banco de dados, chamado “api\_spring”:



Em seguida, iremos configurar o arquivo “application.properties” para estabelecer a conexão com o banco MySQL. Veja o código a seguir:

Alterar o “nome\_base\_de\_dados” para “api\_spring” e também colocar o usuário e senha do MySQL.

```
# Altera a estrutura da tabela caso a entidade tenha mudanças.
spring.jpa.hibernate.ddl-auto=update

# Acesso ao banco de dados
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/nome_base_de_dados

# Usuário do banco de dados
spring.datasource.username=usuário

# Senha do banco de dados
spring.datasource.password=senha
```

Na linha “**spring.jpa.hibernate.ddl-auto=update**”, o termo “**update**” indica o comportamento que o Hibernate adotará em relação à estrutura do banco de dados. Abaixo, você pode encontrar o significado desse e de outros termos disponíveis para essa propriedade:

- **none:** Não realizará alterações na estrutura do banco de dados.
- **update:** Sempre que o projeto for executado, a estrutura das tabelas será atualizada de acordo com a estrutura das entidades. Caso já existam registros nas tabelas, eles serão mantidos.
- **create:** Sempre que o projeto for executado, uma nova tabela será criada para cada entidade. Caso já existam registros nas tabelas, eles serão perdidos, pois a lógica é remover a tabela existente e criar uma nova.

- **create-drop:** Similar ao "create", porém, ao finalizar a execução do sistema, as tabelas são removidas.

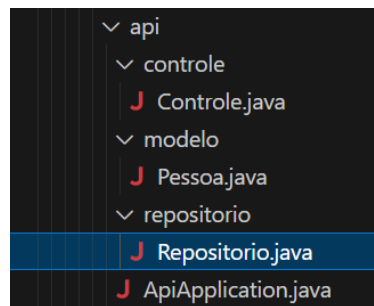
## 16. Repositório

O repositório em uma aplicação Spring tem como finalidade fornecer funcionalidades para manipular registros em um banco de dados, atuando como a camada de persistência da aplicação. Ao implementar o “**CrudRepository**”, teremos acesso a funções básicas de um banco de dados, tais como: cadastrar, selecionar, alterar, excluir e filtrar registros.

Neste tutorial, iremos utilizar o “**CrudRepository**”. No entanto, existem outras opções disponíveis que podem ser implementadas, são elas:

- **CrudRepository:** Fornece funcionalidades básicas de CRUD (Create, Read, Update e Delete).
- **PagingAndSortingRepository:** Implementa métodos para paginação e ordenação de dados.
- **JpaRepository:** Combina as funcionalidades do **CrudRepository** e **PagingAndSortingRepository**.

Para começar, vamos criar uma pasta chamada “repositorio”. Dentro desta pasta, criaremos o arquivo “**Repositorio.java**”.



Agora podemos desenvolver nosso arquivo de repositório:

```
import org.springframework.data.repository CrudRepository;
import org.springframework.stereotype.Repository;

import com.ideau.api.modelo.Pessoa;

@Repository
public interface Repositorio extends CrudRepository<Pessoa, Integer> {
}
```

Aqui estão alguns métodos disponibilizados pelo “**CrudRepository**” e suas respectivas funcionalidades:

- **save():** Realiza o cadastro ou atualização de registros.
- **findAll():** Retorna uma lista contendo todos os registros da tabela.
- **deleteById():** Remove um registro com base no seu identificador.

## 17. Autowired

A injeção de dependências é um padrão de desenvolvimento amplamente adotado por vários frameworks. Ele é utilizado para manter um baixo nível de acoplamento e alta coesão em um projeto. Vamos entender melhor esses conceitos:

- **Baixo acoplamento:** Significa que uma classe não deve depender exclusivamente de outra classe para funcionar corretamente.
- **Alta coesão:** Refere-se ao princípio de que uma classe deve ser designada para realizar ações específicas de forma coesa e independente. Por exemplo, se precisarmos manipular data e hora, trabalhar todas as operações em uma única classe resultaria em baixa coesão. No entanto, ao separar em duas classes distintas, como Data e Hora, estamos aplicando o conceito de alta coesão.

É importante destacar que a annotation **@Autowired** faz com que o objeto seja automaticamente instanciado pelo Spring durante a execução do projeto. Isso elimina a necessidade do desenvolvedor se preocupar com a criação e gerenciamento desses objetos, o que contribui para uma melhor performance do sistema.

Para implementar a injeção de dependência com **@Autowired** em nosso projeto, abra o arquivo **Controle.java** e adicione o seguinte código antes das definições das rotas:

```
@RestController
public class Controle {

    @Autowired
    private Repositorio acao;
```

## 18. Cadastrar com o comando save()

O método `save()` é responsável por cadastrar ou atualizar registros em uma tabela do banco de dados. Não é necessário criar um método específico para inserir dados na tabela; ao utilizar um repositório, o método `save()` já estará disponível para realizar operações de cadastro ou atualização.

Para utilizar o método `save()`, é essencial compreender o uso da annotation **@PostMapping**, que habilita requisições do tipo POST. Sempre que uma rota necessitar enviar dados que não sejam passados via URL, a annotation **@PostMapping** deve ser utilizada.

Agora que compreendemos a função do método `save()` e da annotation **@PostMapping**, podemos criar uma rota de cadastro no arquivo de controle. Abaixo da declaração **@Autowired**, adicione o seguinte código para implementar a rota de cadastro:

```
@Autowired
private Repositorio acao;

@PostMapping("/api")
public Pessoa cadastrar(@RequestBody Pessoa obj){
    return acao.save(obj);
}
```

Para testar a aplicação utilizando o Thunder Client no Visual Studio Code, siga os passos abaixo:

1. Execute a Aplicação:

Certifique-se de que sua aplicação Spring Boot esteja em execução.

2. Abra o Thunder Client:

No Visual Studio Code, localize o ícone do Thunder Client na barra lateral esquerda ou acesse através de View -> Open View... -> Thunder Client.

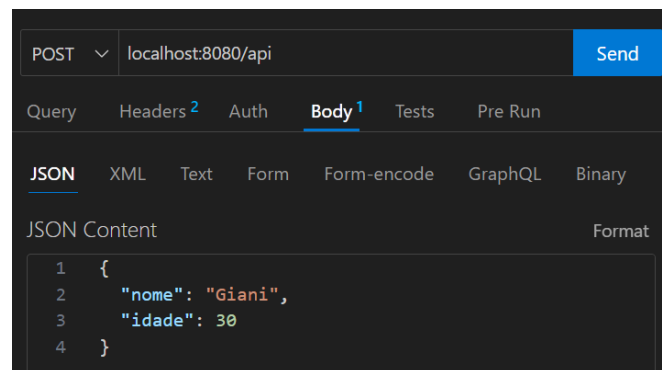
3. Realize o Teste:

Clique no sinal de + ou New Request para criar uma nova requisição.

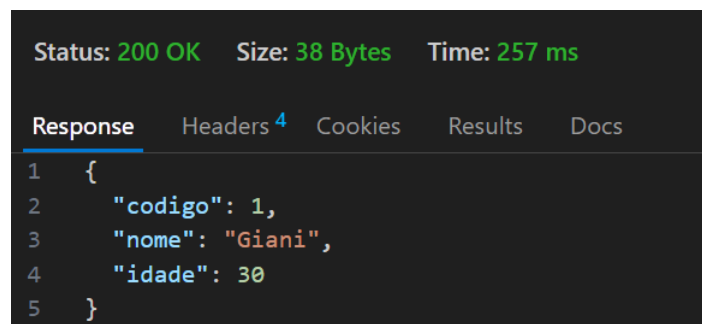
Defina o método como POST.

Insira a URL da sua rota de cadastro, por exemplo, “**http://localhost:8080/api**”.

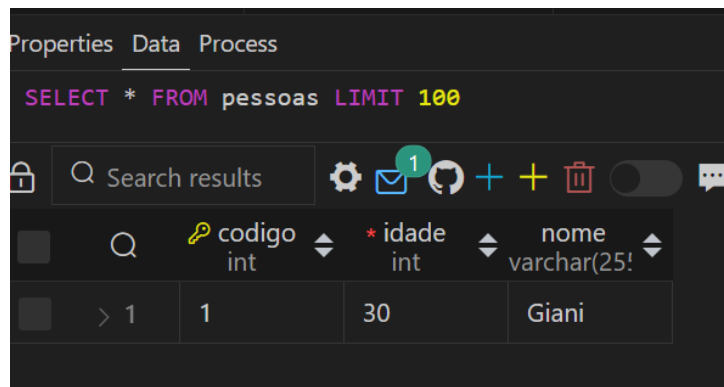
No corpo da requisição, adicione um JSON com os dados da Pessoa que você deseja cadastrar.



Após enviar a requisição, você deverá receber uma resposta indicando que o cadastro foi realizado com sucesso.



Você também pode verificar no banco de dados se o registro foi adicionado corretamente.



## 19. Selecionar com o comando findAll()

O método `findAll()` por padrão retorna um objeto do tipo `Iterable`. Frequentemente, os desenvolvedores desejam alterar esse tipo de retorno para `List`, por exemplo. Para isso, é possível sobrescrever o método `findAll()` no repositório.

Vale lembrar que utilizaremos a annotation `@GetMapping` exclusivamente para retornar listagens ou filtragens de dados. Ao contrário do `@PostMapping` que vimos anteriormente, quando usamos `@GetMapping`, não é possível receber dados no corpo da requisição; os dados são obtidos apenas através da URL utilizando `@PathVariable`.

Agora, vamos à prática. Abra o seu arquivo de repositório e faça a seguinte implementação:

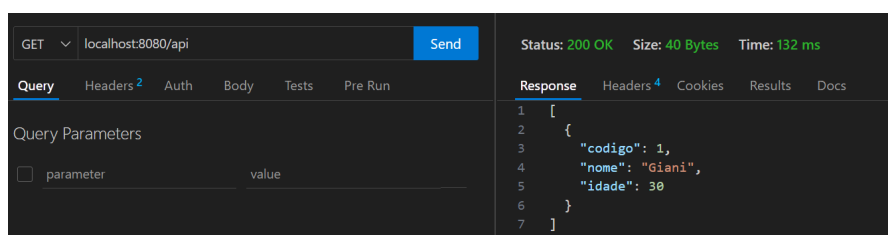
```
@Repository
public interface Repositorio extends CrudRepository<Pessoa, Integer> {

    List<Pessoa> findAll();
}
```

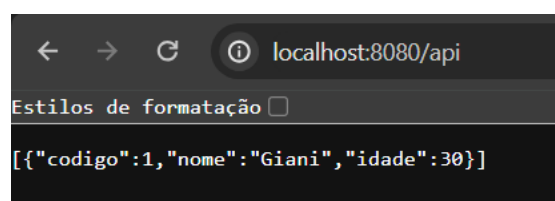
Agora podemos ir no arquivo de controle e criar uma rota para efetuar a seleção:

```
@GetMapping("/api")
public List<Pessoa> selecionar(){
    return acao.findAll();
}
```

Podemos testar esta rota de duas maneiras: uma utilizando o Thunder Client



E outra através do navegador digitando a URL `localhost:8080/api`:



## 20. Filtrar com o comando findBy()

O método findBy() funciona de forma similar ao comando “WHERE” do SQL. Para utilizá-lo corretamente, é essencial conhecer os atributos disponíveis em nosso modelo. Neste tutorial, os atributos são:

Código

Nome

Idade

Com base nisso, temos os seguintes métodos de filtragem disponíveis:

- findByCodigo(int codigo)
- findByNome(String nome)
- findByIdade(int idade)

Observe que utilizamos o prefixo findBy, seguido do nome do atributo do nosso modelo. É importante notar que o retorno pode ser de um único registro ou de vários, conforme os exemplos a seguir:

- Pessoa findByCodigo(int codigo); - Como o código é único, o método retornará apenas um registro.
- List<Pessoa> findByNome(String nome); - Se houver duas pessoas com o mesmo nome, o método retornará uma lista de pessoas.

Lembre-se: o findBy é equivalente ao comando WHERE do SQL. Portanto, ao realizar uma busca por nome, a consulta SQL seria algo como: “SELECT \* FROM tabela WHERE nome = ‘Maria’”;

Agora, mãos à obra! Vamos ao nosso repositório e criar o método findByCodigo.

```
@Repository
public interface Repositorio extends CrudRepository<Pessoa, Integer> {

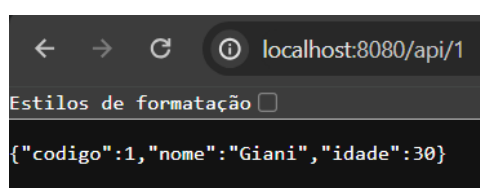
    List<Pessoa> findAll();

    Pessoa findByCodigo(int codigo);
}
```

Agora vamos criar nossa rota no arquivo de controle:

```
@GetMapping("/api/{codigo}")
public Pessoa selecionarPeloCodigo(@PathVariable int codigo){
    return acao.findByCodigo(codigo);
}
```

Podemos realizar um teste via navegador (localhost:8080/api/1):



```
localhost:8080/api/1
Estilos de formatação
{"codigo":1,"nome":"Giani","idade":30}
```



## 21. Alterar dados com o comando save()

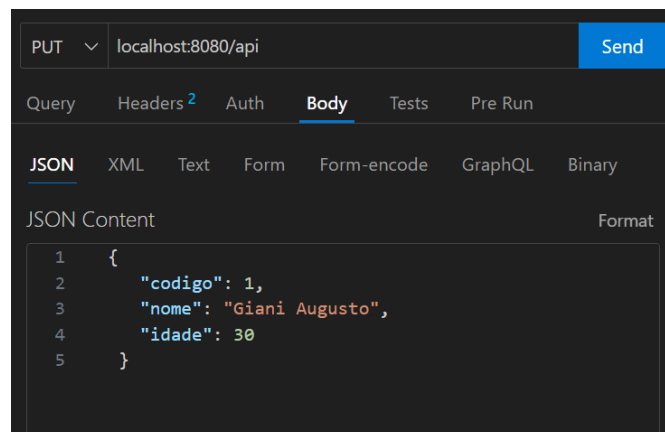
O método “save()” funciona de forma semelhante ao cadastro, porém para sua correta execução, é necessário fornecer um objeto completo contendo todas as características de uma pessoa. O nosso modelo é composto por código, nome e idade, portanto, todos esses dados devem ser fornecidos para que o método funcione corretamente.

Adicionalmente, utilizamos uma nova annotation de requisição, denominada “@PutMapping”, que informa à nossa API que uma rota com este tipo de requisição será responsável por atualizar um registro existente.

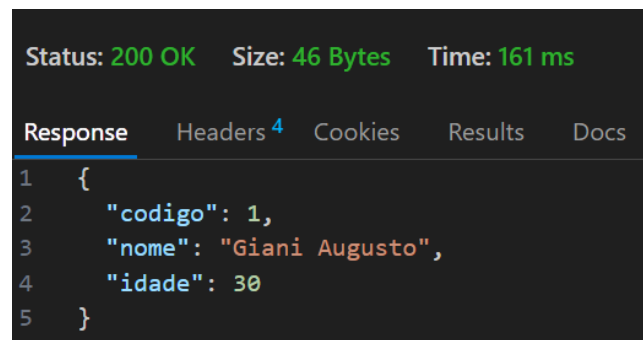
Vamos começar pela implementação da nossa rota. Abra o arquivo de controle e insira o seguinte código:

```
@PutMapping("/api")
public Pessoa editar(@RequestBody Pessoa obj){
    return acao.save(obj);
}
```

Utilizando o Thunder Client, podemos testar:



O retorno será nosso objeto atualizado:



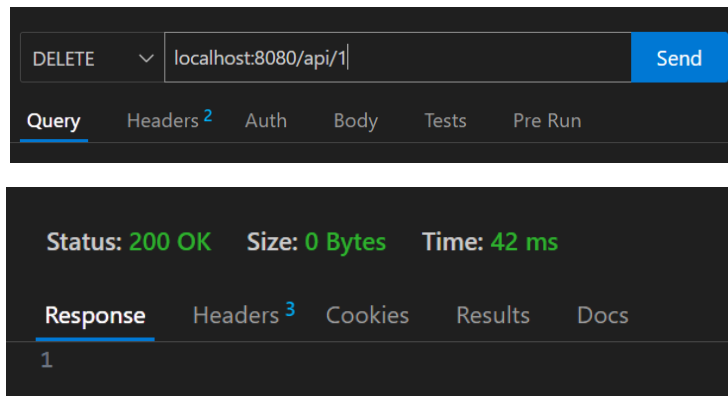
## 22. Excluir com o comando delete()

Para que o método delete() funcione corretamente, é necessário enviar um objeto completo. Na rota de exclusão, faremos uma filtragem pelo código da pessoa e, em seguida, enviaremos o objeto completo para efetuar a exclusão do registro correspondente. Utilizaremos uma nova annotation, **@DeleteMapping**, para especificar o tipo de requisição e indicar que a rota é responsável pela exclusão de registros.

Vamos implementar essa funcionalidade. No arquivo de controle, adicione a seguinte rota:

```
@DeleteMapping("/api/{codigo}")
public void remover(@PathVariable int codigo){
    Pessoa obj = selecionarPeloCodigo(codigo);
    acao.delete(obj);
}
```

Podemos testar utilizando o Thunder Client:



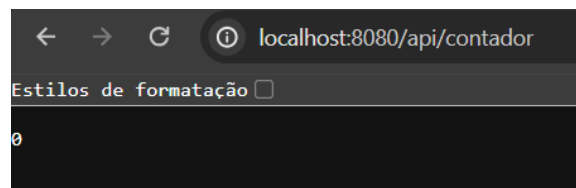
### 23. Contar registros com o comando count()

O método count() retorna uma informação do tipo long, equivalente ao comando “SQL: SELECT COUNT(\*) FROM tabela”;

Para implementar essa funcionalidade, é bastante simples. No arquivo de controle, adicione a seguinte rota:

```
@GetMapping("/api/contador")
public long contador(){
    return acao.count();
}
```

Agora podemos acessar essa rota via navegador, através da url: localhost:8080/api/contador



### 24. Ordenar registros

Vamos aprender a ordenar dados de uma tabela, técnica equivalente ao comando SQL 'ORDER BY'. Esta funcionalidade permite ordenar informações, sejam elas textuais ou numéricas. Para implementar essa ordenação, abra o seu repositório e adicione o seguinte método:

```
List<Pessoa> findByOrderByNome();
```

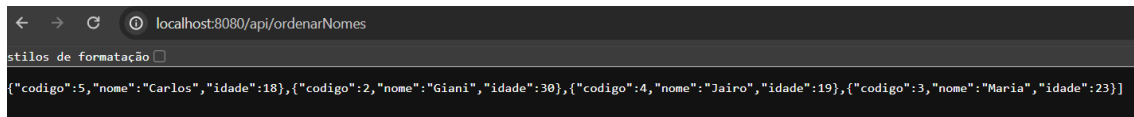
Para ordenar os dados, utilizamos o prefixo “findByOrderBy”, seguido do nome de uma característica do nosso modelo.

Dica! Por padrão, as ordenações são realizadas de A a Z ou do menor número para o maior. Caso deseje ordenar de forma inversa, ou seja, de Z a A para textos e do maior para o menor para números, basta adicionar o termo “Desc”. Por exemplo, no nosso método, poderíamos renomeá-lo para “**findByOrderByNomeDesc**” para ordenar os nomes em ordem decrescente. Faça o teste para compreender melhor o funcionamento do termo “Desc”.

Abra o arquivo de controle e adicione a seguinte rota:

```
@GetMapping("/api/ordenarNomes")
public List<Pessoa> ordenarNomes(){
    return acao.findByOrderByNome();
}
```

Para testar, abra o navegador e acesse o seguinte link: “localhost:8080/api/ordenarNomes”. O resultado esperado será:



```
{ "codigo": 5, "nome": "Carlos", "idade": 18 }, { "codigo": 2, "nome": "Giani", "idade": 30 }, { "codigo": 4, "nome": "Jairo", "idade": 19 }, { "codigo": 3, "nome": "Maria", "idade": 23 }
```

## 25. Filtrar registros com o comando containing

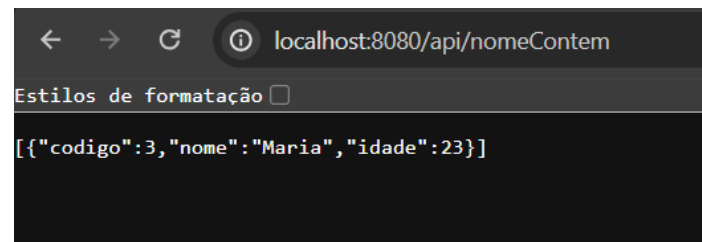
Vamos aprender a utilizar o comando “contains”, que é equivalente ao comando SQL “LIKE”. Ao usar a expressão “contains”, os registros listados serão aqueles que contêm um termo específico, que pode ser uma letra, uma palavra ou até mesmo uma frase. Para implementar essa funcionalidade, abra o seu repositório e adicione o seguinte método:

```
List<Pessoa> findByNomeContaining(String termo);
```

Agora podemos implementar uma rota. Abra o arquivo de controle e adicione o seguinte código:

```
@GetMapping("/api/nomeContem")
public List<Pessoa> nomeContem(){
    return acao.findByNomeContaining(termo: "m");
}
```

Para finalizarmos esta etapa do tutorial, vamos testar. Abra o navegador e acesse a URL: “localhost:8080/api/nomeContem”. O resultado esperado será:



```
[{"codigo": 3, "nome": "Maria", "idade": 23}]
```

## 26. Annotation @Query

Vamos utilizar a annotation “@Query”, que nos permite executar comandos SQL personalizados em nosso projeto. Como o JPA não disponibiliza métodos com todas as funcionalidades do SQL, podemos personalizar nossas consultas.

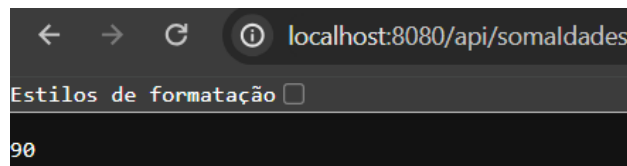
Abra o seu repositório e adicione o seguinte método:

```
@Query(value = "SELECT SUM(idade) FROM pessoas", nativeQuery = true)
int somaIdades();
```

Criamos um comando SQL que é responsável por somar as idades. Agora, podemos abrir o arquivo de controle e adicionar a seguinte rota:

```
@GetMapping("/api/somaIdades")
public int somaIdades(){
    return acao.somaIdades();
}
```

Para testar, abra o navegador e acesse o link: “localhost:8080/api/somaIdades”. O resultado esperado será:



Continuaremos utilizando a annotation “@Query”. Em alguns momentos, é necessário passar parâmetros para que nosso comando SQL consiga executar uma ação específica. Veja abaixo como implementar o uso de parâmetros na annotation “@Query”:

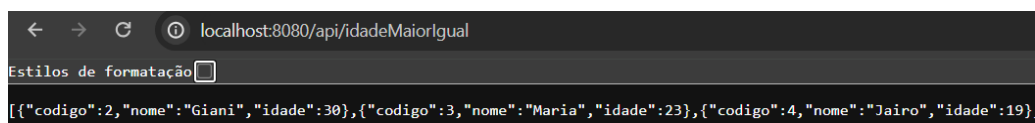
Vamos supor que desejamos listar todos os registros onde a pessoa tenha idade maior ou igual a 18 anos. Abra o arquivo “Repositorio.java” e adicione o seguinte método:

```
@Query(value = "SELECT * FROM pessoas WHERE idade >= :idade", nativeQuery = true)
List<Pessoa> idadeMaiorIgual(int idade);
```

Para manipular parâmetros em nossa “@Query”, basta utilizar dois pontos seguidos pelo nome do parâmetro da função. Neste exemplo, para extrairmos a idade do método “idadeMaiorIgual”, utilizaremos a expressão “:idade”. No nosso “Controle.java”, podemos implementar a seguinte rota:

```
@GetMapping("/api/idadeMaiorIgual")
public List<Pessoa> idadeMaiorIgual(){
    return acao.idadeMaiorIgual(idade:18);
}
```

Podemos testar, abra o navegador e acesse o link: localhost:8080/api/idadeMaiorIgual, o resultado será esse:



## 27. ResponseEntity

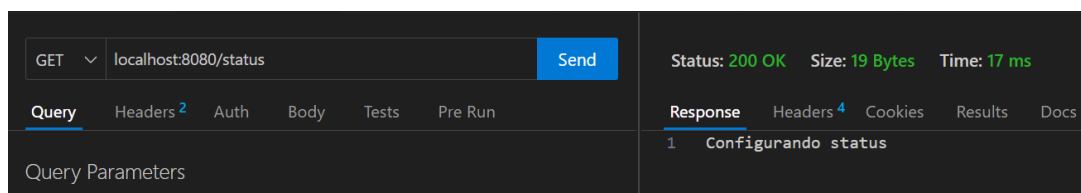
Vamos aprender a utilizar o ‘ResponseEntity’, que é responsável por retornar o status de uma requisição. Abaixo, estão alguns dos status que podemos utilizar em nossos projetos:

Código	Status	Descrição
200	Ok	Requisições bem-sucedidas
201	Created	Quando efetuado algum cadastro
202	Accepted	Requisição aceita, porém não executada
400	Bad Request	Quando não é possível realizar determinada
401	Unauthorized	Sem autorização para executar alguma
404	Not Found	O servidor não consegue encontrar a rota ou
429	Too Many Requests	Quando o usuário efetua muitas requisições

Vamos implementar nosso projeto. Primeiramente, no arquivo de controle, crie a seguinte rota:

```
@GetMapping("/status")
public String status(){
    return "Configurando status";
}
```

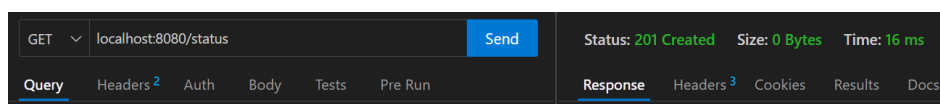
Agora podemos testar. Abra o Thunder Client e acesse a URL: `localhost:8080/status`. O resultado esperado será:



Note que o status da requisição é “200 Ok”. Mas, e se quisermos alterar esse status? É aqui que entra o nosso “ResponseEntity”. Vamos fazer a seguinte implementação na nossa rota:

```
@GetMapping("/status")
public ResponseEntity<?> status(){
    return new ResponseEntity<>(HttpStatus.CREATED);
}
```

Nossa rota agora retornará o status “201 Created”, garantindo que essa requisição especifique exatamente a ação efetuada. Veja abaixo como será o retorno dessa rota:

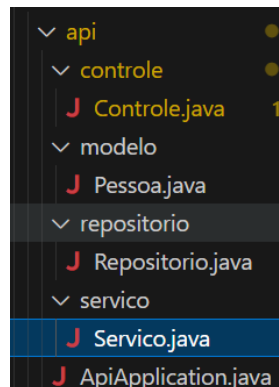


## 28. Annotation @Service

Vamos aprender a utilizar a annotation “@Service”, responsável por ter uma camada de serviços. As vantagens de criar essa camada em nossos projetos são:

- Melhor organização dos projetos, já que é nesse tipo de camada que ficam as regras de negócio.
- Todas as classes que possuem a annotation “@Service” podem usufruir da injeção de dependências (“@Autowired”).

Inicialmente, podemos criar uma pasta chamada “servico” e dentro dela um arquivo chamado “Servico.java”:



Agora podemos implementar a annotation “@Service” na classe. Não esqueça de realizar a importação da annotation:

```
import org.springframework.stereotype.Service;

@Service
public class Servico {

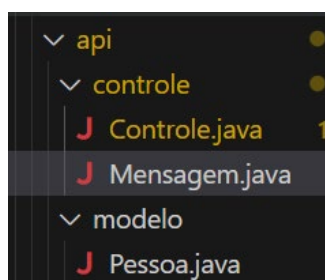
}
```

## 29. Annotation @Component

Vamos aprender a utilizar a annotation “@Component”. Esta annotation é geralmente utilizada para criar beans gerenciados pelo Spring. Diferente da annotation “@Entity”, a “@Component” não cria uma tabela no banco de dados.

Outra característica importante é a possibilidade de manipularmos um objeto da classe com a annotation “@Component” via injeção de dependências, utilizando a annotation “@Autowired”.

Vamos colocar a mão na massa e criar nossa classe dentro da pasta “modelo”. Iremos criar uma classe chamada “Mensagem.java”:



Na classe “Mensagem”, teremos apenas um atributo e os métodos “get” e “set”. Esta classe será responsável por retornar alguma mensagem em caso de falha ou quando alguma rota, por padrão, não possui retorno:

```
import org.springframework.stereotype.Component;

@Component
public class Mensagem {

    private String mensagem;

    public String getMensagem() {
        return mensagem;
    }

    public void setMensagem(String mensagem) {
        this.mensagem = mensagem;
    }
}
```

Para finalizarmos, iremos na classe “Servico” para criar um atributo do tipo “Mensagem” para utilizarmos futuramente:

```
@Service
public class Servico {

    @Autowired
    private Mensagem mensagem;
}
```

### 30. Implementando serviços

Vamos implementar um método de cadastro na classe de serviços, criando uma validação nos dados recebidos através das requisições POST.

Abra o arquivo “Servico.java” e crie um atributo do tipo “Repositorio”. Através desse atributo, conseguiremos efetuar as ações com a nossa tabela de pessoas:

```
@Service
public class Servico {

    @Autowired
    private Mensagem mensagem;

    @Autowired
    private Repositorio repositorio;
}
```

O próximo passo é criarmos um método para realizarmos o cadastro de pessoas. Será realizada uma validação nas características “nome” e “idade”. Caso estejam validados, o cadastro será realizado:

```

public ResponseEntity<> cadastrar(Pessoa obj){
    if(obj.getNome().equals(anObject:"")){
        mensagem.setMensagem(mensagem:"O nome precisa ser preenchido.");
        return new ResponseEntity<>(mensagem, HttpStatus.BAD_REQUEST);
    } else if (obj.getIdade() < 0){
        mensagem.setMensagem(mensagem:"Informe uma idade válida.");
        return new ResponseEntity<>(mensagem, HttpStatus.BAD_REQUEST);
    }else{
        return new ResponseEntity<>(acao.save(obj), HttpStatus.BAD_REQUEST);
    }
}
}

```

Podemos prosseguir para o nosso arquivo de controle e criar um atributo do tipo “Servico”:

```

@Autowired
private Servico servico;

```

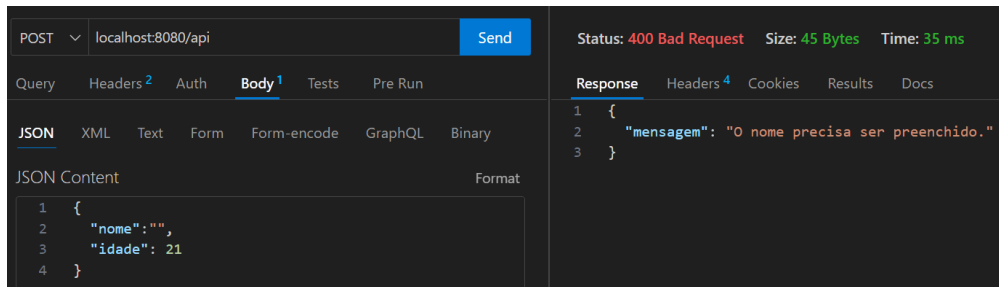
Ainda no arquivo de controle, localize a rota de cadastro e faça a seguinte implementação:

```

@PostMapping("/api")
public ResponseEntity<> cadastrar(@RequestBody Pessoa obj){
    return servico.cadastrar(obj);
}

```

Vamos realizar alguns testes. Abra o Thunder Client e tente cadastrar uma pessoa, deixando o campo do nome vazio:



Vamos implementar um método para selecionar dados na classe de serviços. Inicialmente, abra o arquivo “Servico.java” e adicione o seguinte método:

```

public ResponseEntity<> selecionar(){
    return new ResponseEntity<>(acao.findAll(), HttpStatus.OK);
}

```

O próximo passo será modificar o método responsável por selecionar os registros. Este método está no arquivo “Controle.java”.

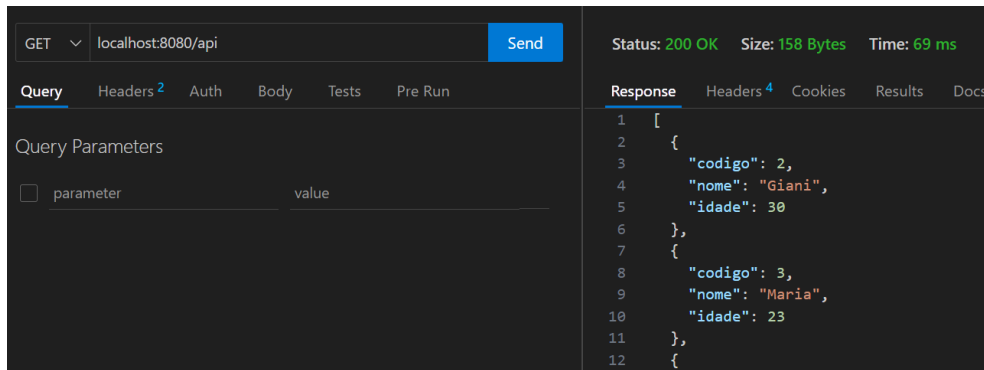
```

@GetMapping("/api")
public ResponseEntity<> selecionar(){
    return servico.selecionar();
}

```

Agora podemos testar. Abra o Thunder Client e defina a seguinte URL: “localhost:8080/api/selecionar”, certifique-se de definir o método como GET.





Vamos implementar um método para selecionar dados pelo código na classe de serviços.

Primeiramente, criaremos um método em nosso arquivo “Repositorio.java”. Este novo método será responsável por retornar a quantidade de registros com base no código. Se o retorno for zero, isso significa que não há pessoas com aquele código. Caso contrário, saberemos que existe ao menos uma pessoa com aquele código.

```
int countByCodigo(int codigo);
```

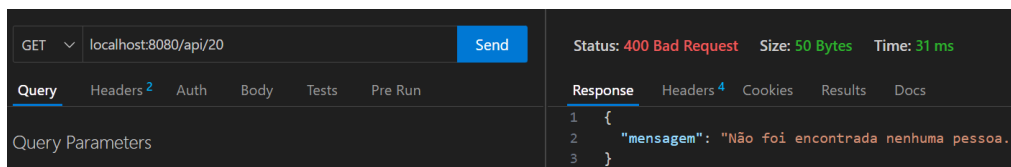
Agora podemos abrir o arquivo “Servico.java” e implementar o seguinte método:

```
public ResponseEntity<> selecionarPeloCodigo(int codigo){
    if(acao.countByCodigo(codigo) == 0){
        mensagem.setMensagem(mensagem: "Não foi encontrada nenhuma pessoa.");
        return new ResponseEntity<>(mensagem, HttpStatus.BAD_REQUEST);
    }else{
        return new ResponseEntity<>(acao.findByCodigo(codigo), HttpStatus.OK);
    }
}
```

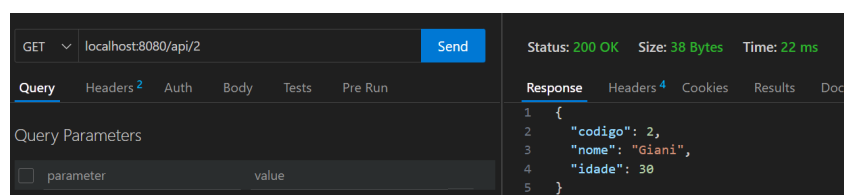
Agora, podemos abrir o arquivo “Controle.java” e implementar o método “selecionarPeloCodigo()”:

```
@GetMapping("/api/{codigo}")
public ResponseEntity<> selecionarPeloCodigo(@PathVariable int codigo){
    return servico.selecionarPeloCodigo(codigo);
}
```

Repositório, serviço e controle foram implementados. Agora, podemos realizar um teste. Abra o Thunder Client e utilize o link: “localhost:8080/api/20”. Neste teste, não temos nenhum registro com o código 20, portanto, receberemos uma mensagem informando que não existe nenhuma pessoa com aquele código:



Agora, vamos buscar um usuário que existe: “localhost:8080/api/2”. Veja o exemplo:



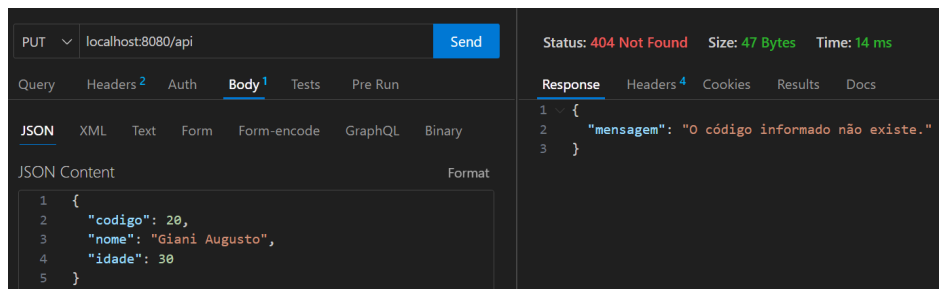
Vamos criar um método para efetuar a atualização de dados. Inicialmente, no arquivo “Servico.java”, implementaremos uma função que valida os dados antes de proceder com a atualização do registro:

```
public ResponseEntity<?> editar(Pessoa obj) {  
  
    if (acao.countByCodigo(obj.getCodigo()) == 0) {  
        mensagem.setMensagem(mensagem:"O código informado não existe.");  
        return new ResponseEntity<>(mensagem, HttpStatus.NOT_FOUND);  
    } else if (obj.getNome().equals(anObject:"")) {  
        mensagem.setMensagem(mensagem:"É necessário informar um nome");  
        return new ResponseEntity<>(mensagem, HttpStatus.BAD_REQUEST);  
    } else if (obj.getIdade() < 0) {  
        mensagem.setMensagem(mensagem:"Informe uma idade válida");  
        return new ResponseEntity<>(mensagem, HttpStatus.BAD_REQUEST);  
    } else {  
        return new ResponseEntity<>(acao.save(obj), HttpStatus.OK);  
    }  
}
```

O próximo passo é implementar o método de alteração no arquivo “Controle.java”:

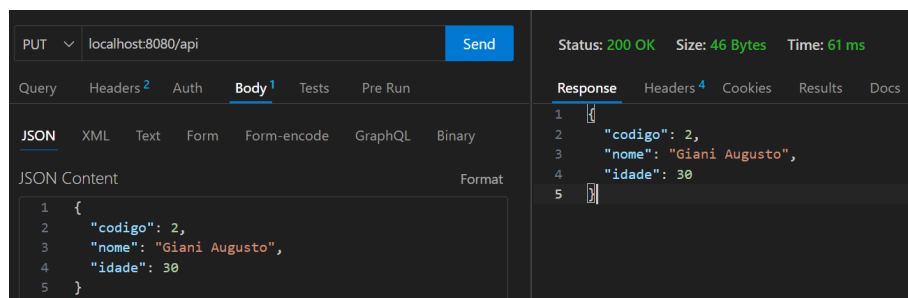
```
@PutMapping("/api")  
public ResponseEntity<?> editar(@RequestBody Pessoa obj) {  
    return servico.editar(obj);  
}
```

Vamos testar a nossa rota de alteração. Abra o Thunder Client e faça a seguinte requisição:



PUT	localhost:8080/api	Send
Query	Headers <sup>2</sup>	Auth
Body <sup>1</sup>	Tests	Pre Run
JSON	XML	Text
Form	Form-encode	GraphQL
Binary		
JSON Content		Format
1	{	
2	"codigo": 20,	
3	"nome": "Giani Augusto",	
4	"idade": 30	
5	}	
Status: 404 Not Found	Size: 47 Bytes	Time: 14 ms
Response	Headers <sup>4</sup>	Cookies
Results	Docs	
1	{	
2	"mensagem": "O código informado não existe."	
3	}	

Agora podemos passar um código válido, junto com um nome e uma idade. O nosso retorno será o seguinte:



PUT	localhost:8080/api	Send
Query	Headers <sup>2</sup>	Auth
Body <sup>1</sup>	Tests	Pre Run
JSON	XML	Text
Form	Form-encode	GraphQL
Binary		
JSON Content		Format
1	{	
2	"codigo": 2,	
3	"nome": "Giani Augusto",	
4	"idade": 30	
5	}	
Status: 200 OK	Size: 46 Bytes	Time: 61 ms
Response	Headers <sup>4</sup>	Cookies
Results	Docs	
1	{	
2	"codigo": 2,	
3	"nome": "Giani Augusto",	
4	"idade": 30	
5	}	

Vamos criar um método para excluir registros da tabela. Inicialmente, criaremos um método no arquivo “Servico.java”:

```

public ResponseEntity<?> remover(int codigo) {

    if (acao.countByCodigo(codigo) == 0) {
        mensagem.setMensagem(mensagem:"O código informado não existe.");
        return new ResponseEntity<>(mensagem, HttpStatus.NOT_FOUND);
    } else {
        Pessoa obj = acao.findByCodigo(codigo);
        acao.delete(obj);
        mensagem.setMensagem(mensagem:"Pessoa removida com sucesso!");
        return new ResponseEntity<>(mensagem, HttpStatus.OK);
    }
}

```

Observe que a imagem acima contém uma validação baseada no código. Se o código não existir, uma mensagem será retornada; caso contrário, o registro será excluído. Vamos implementar o método “remover” no arquivo “Controle.java”:

```

@DeleteMapping("/api/{codigo}")
public ResponseEntity<?> remover(@PathVariable int codigo) {
    return servico.remover(codigo);
}

```

Com o controle implementado, podemos realizar testes usando o Thunder Client. Execute o seguinte teste:

DELETE localhost:8080/api/3

Status: 200 OK Size: 43 Bytes Time: 40 ms

Response

```

1 {
2   "mensagem": "Pessoa removida com sucesso!"
3 }

```

JSON Content

```

1 {
2   "codigo": 2,
3   "nome": "Giani Augusto",
4   "idade": 30
5 }

```

Funcionou! E se tentarmos executar novamente?

Vai exibir uma mensagem de erro, pois essa pessoa não existe mais.

DELETE localhost:8080/api/3

Status: 404 Not Found Size: 47 Bytes Time: 9 ms

Response

```

1 {
2   "mensagem": "O código informado não existe."
3 }

```

JSON Content

```

1 {
2   "codigo": 2,
3   "nome": "Giani Augusto",
4   "idade": 30
5 }

```