



Guia Passo a Passo para Desenvolvimento de uma API com Spring Boot e aplicação web

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 e uma aplicação web com conexão no banco de dados.

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

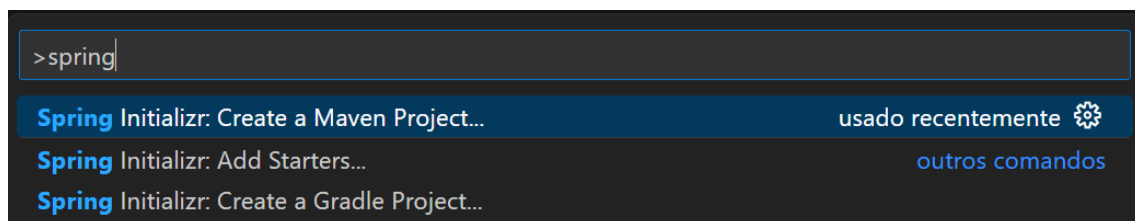
Na configuração do ambiente de desenvolvimento, será necessário adicionar duas extensões: uma para o Java e outra para o Spring Boot.

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

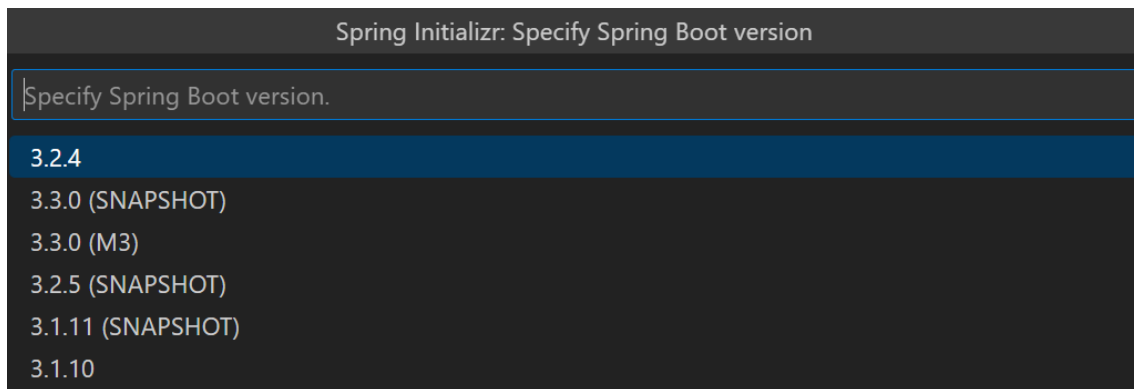
4. Criando o primeiro projeto

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".

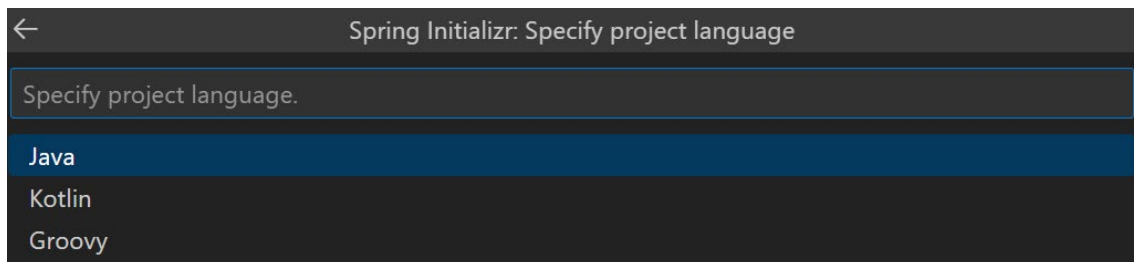
O próximo passo é inserir o termo "Spring" na paleta de comandos e selecionar a opção "Spring Initializr: Create a Maven Project...". A imagem abaixo ilustra como deve estar o seu Visual Studio Code neste momento:



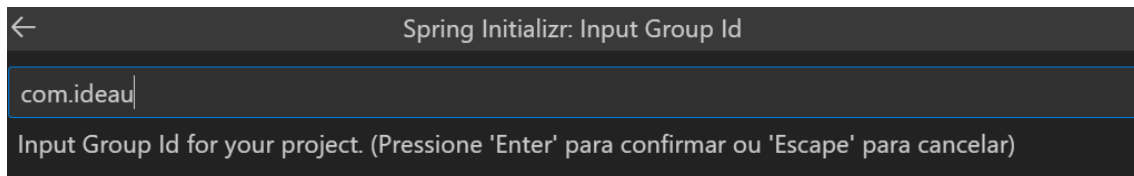
Agora é necessário especificar a versão do Spring Boot. Neste tutorial, estamos utilizando a versão 3.2.4. Recomendo fortemente deixar selecionada a primeira opção da lista de versões, pois é a versão mais estável do Spring Boot.

A screenshot of the 'Spring Initializr: Specify Spring Boot version' screen. It features a dark grey header with a back arrow and the title. Below is a text input field containing 'Specify Spring Boot version.'. A list of versions is shown below the input: '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'.

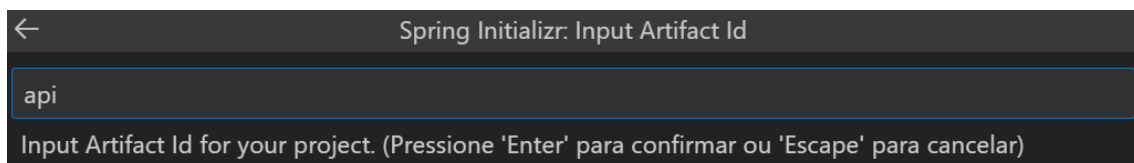
Você pode escolher o tipo de linguagem para desenvolver com o Spring. As opções disponíveis são: Java, Kotlin e Groovy. Selecione a opção "Java" e prossiga com o tutorial:

A screenshot of the 'Spring Initializr: Specify project language' screen. It has a dark grey header with a back arrow and the title. Below is a text input field with 'Specify project language.'. A list of languages is shown: '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":

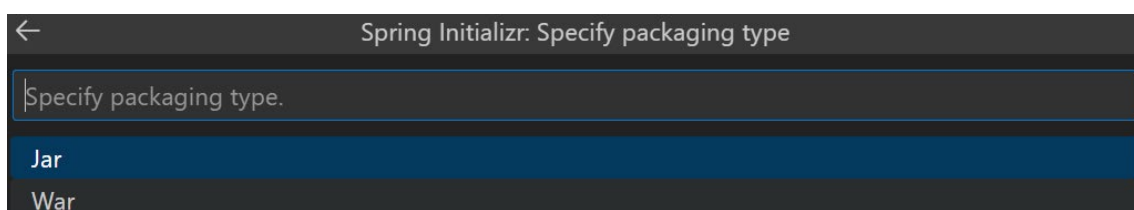
A screenshot of the 'Spring Initializr: Input Group Id' screen. It has a dark grey header with a back arrow and the title. Below is a text input field containing 'com.ideal'. At the bottom, there is a prompt: 'Input Group Id for your project. (Pressione 'Enter' para confirmar ou 'Escape' para cancelar)'.

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

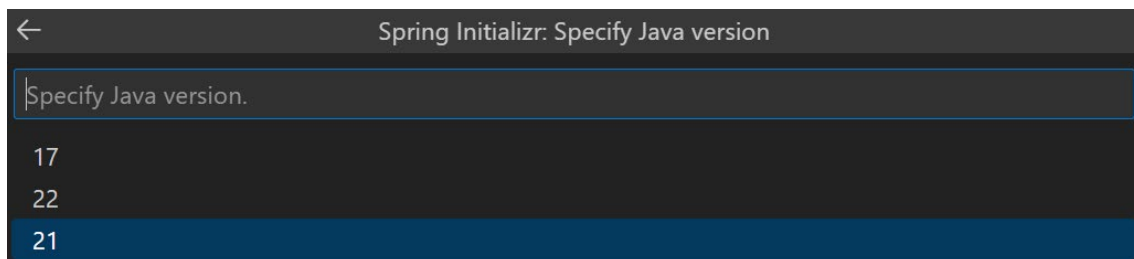
A screenshot of the 'Spring Initializr: Input Artifact Id' screen. It has a dark grey header with a back arrow and the title. Below is a text input field containing 'api'. At the bottom, there is a prompt: 'Input Artifact Id for your project. (Pressione 'Enter' para confirmar ou 'Escape' para cancelar)'.

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 grey header with a back arrow and the title. Below is a text input field with 'Specify packaging type.'. A list of packaging types is shown: 'Jar' (highlighted in blue) and 'War'.

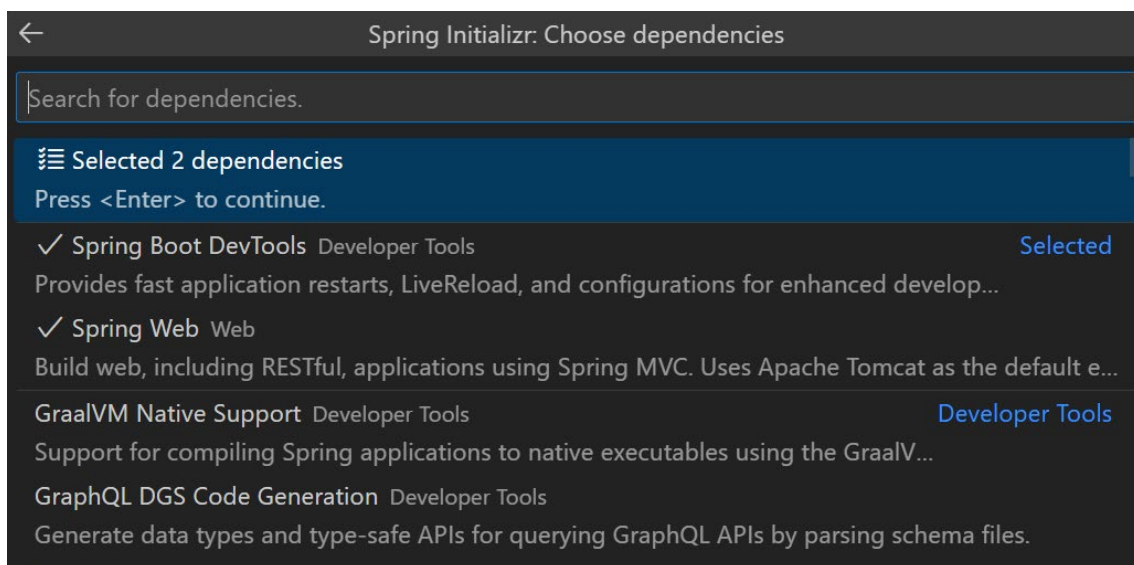
Será necessário informar a versão do Java. Recomendo fortemente utilizar a mesma versão instalada em seu computador. Neste tutorial, estava usando o JDK 21. Portanto, a versão escolhida para gerar o projeto também foi a 21. Se o projeto for gerado em uma versão diferente do JDK, pode resultar em erros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

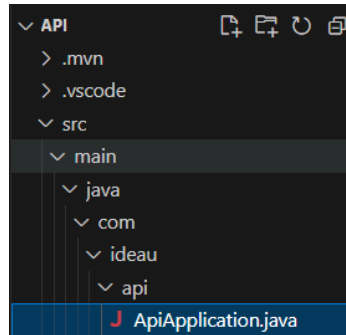
src: Nesta pasta, você irá desenvolver toda a estrutura do projeto. Além do código-fonte, você pode realizar algumas configurações no arquivo "application.properties", como conexões com banco de dados, alterar a porta de funcionamento, habilitar log de ações, entre outras funcionalidades.

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

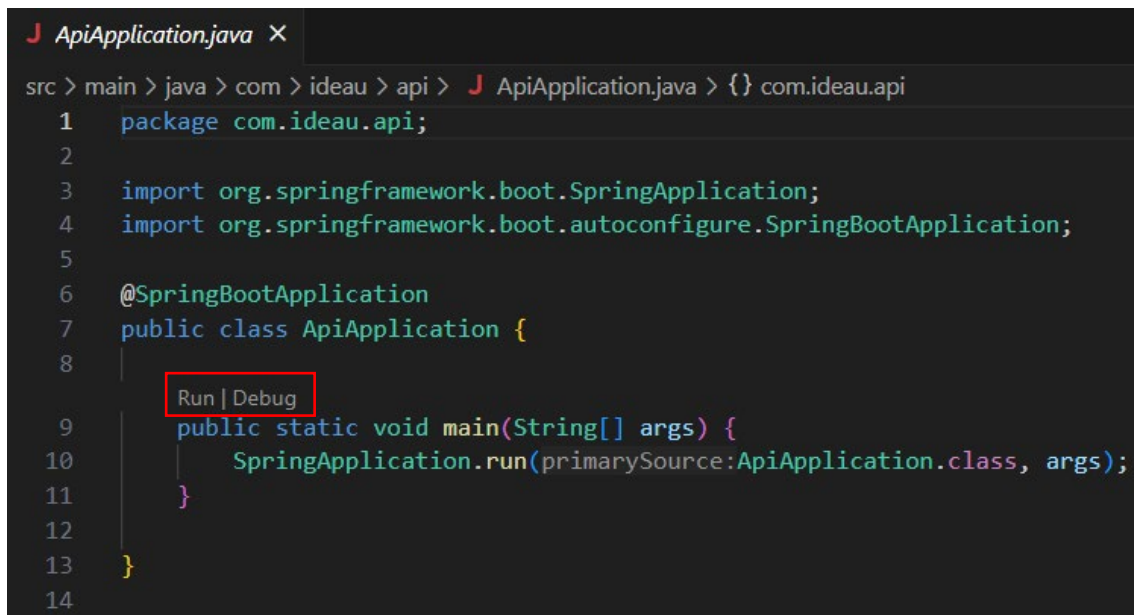
pom.xml: É considerado o "cérebro" do nosso projeto, pois contém todo o conhecimento necessário para realizarmos diversas funcionalidades em nossos projetos. Aqui, você pode especificar conexões com bancos de dados, configuração de servidor, segurança de aplicações, envio de e-mails, entre outras ações.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

5. Executando o projeto.

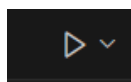
Com o projeto aberto, localize e selecione o arquivo “**ApiApplication.java**”:



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

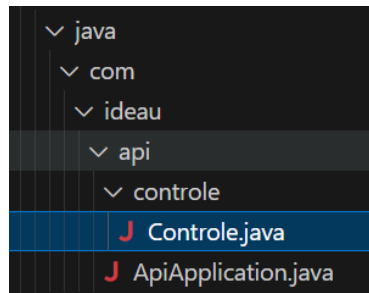
As annotations são utilizadas para adicionar funcionalidades extras a classes, atributos ou métodos em Java. Se você já tem experiência com Java ou C#, provavelmente já se deparou com a annotation **@Override**. Abaixo, apresentamos algumas annotations que serão utilizadas nesta etapa da aula:

- **@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:

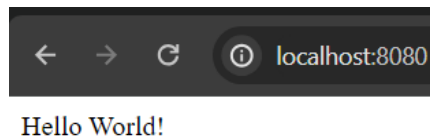
1. Clique com o botão direito do mouse na pasta **api** do seu projeto.
2. Selecione a opção para **criar uma nova pasta** e nomeie-a como **controle**.
3. Dentro da pasta controle, **crie um novo arquivo Java** e nomeie-o como **Controle.java**.



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:



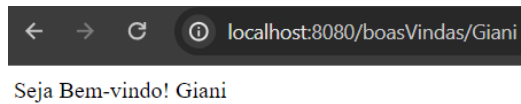
8. @PathVariable

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 }
```

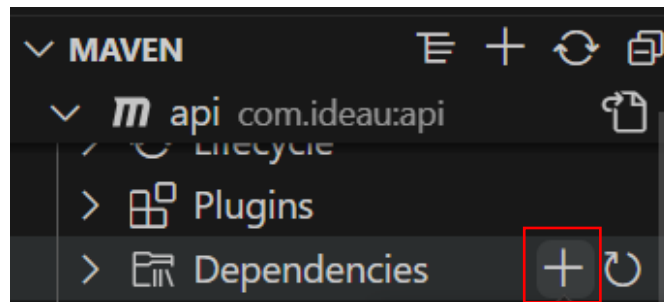
Agora, você pode testar acessando o seguinte endereço no seu navegador: localhost:8080/boasVindas/Giani. O nome "Giani" pode ser substituído por qualquer outro. Esta rota irá concatenar uma mensagem com o nome fornecido na URL:



9. 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:

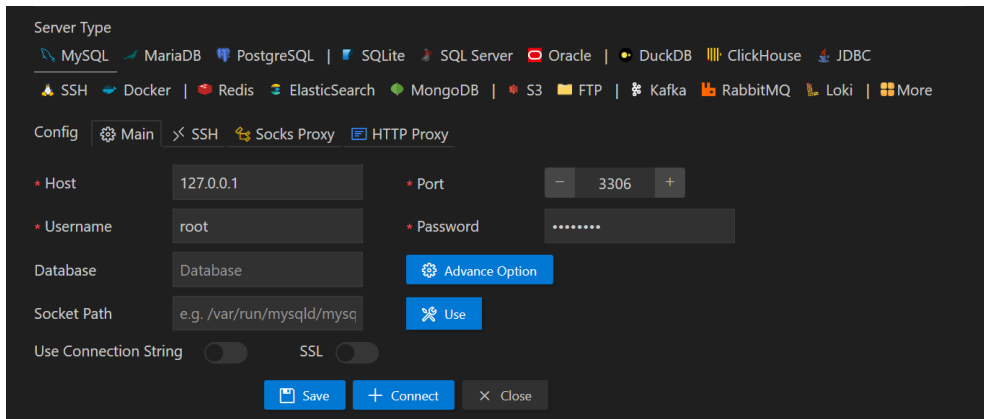
- mysql-connector-java
- spring-boot-starter-data-jpa

```
<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>
```

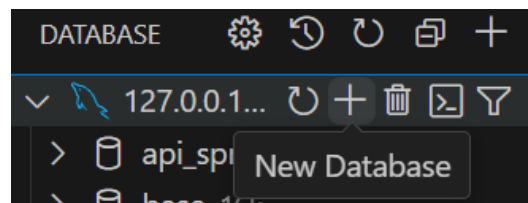
10. Configurando banco de dados MySQL

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.



11. 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 “loja_online” 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.

12. Gerando Tabelas e Criando a classe Modelo

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 **"Produto.java"**:



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:

```

1
2 import jakarta.persistence.Entity;
3 import jakarta.persistence.GeneratedValue;
4 import jakarta.persistence.GenerationType;
5 import jakarta.persistence.Id;
6 import jakarta.persistence.Table;
7
8 @Entity
9 @Table(name = "produto")
10 public class Produto {
11
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Long id;
15
16     private String nome;
17     private Double preco;
18     private String categoria;
19
20
21     public Long getId() {
22         return id;
23     }
24
25     public void setId(Long id) {
26         this.id = id;
27     }

```

13. 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:

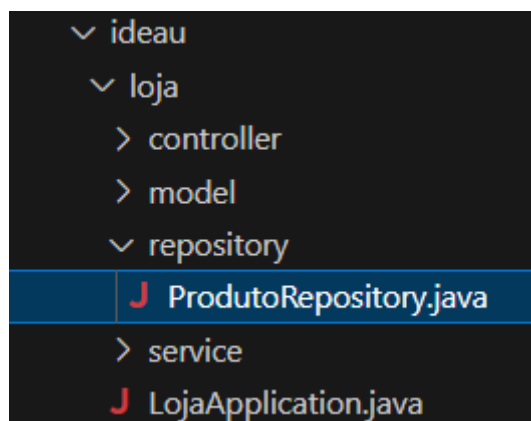
14. Repositório

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 “**ProdutoRepository**”, teremos acesso a funções básicas de um banco de dados, tais como: cadastrar, selecionar, alterar, excluir e filtrar registros.

Neste tutorial, iremos utilizar o “**ProdutoRepository**”. 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 “repository”. Dentro desta pasta, criaremos o arquivo “**ProdutoRepository.java**”.



Agora podemos desenvolver nosso arquivo de repositório:

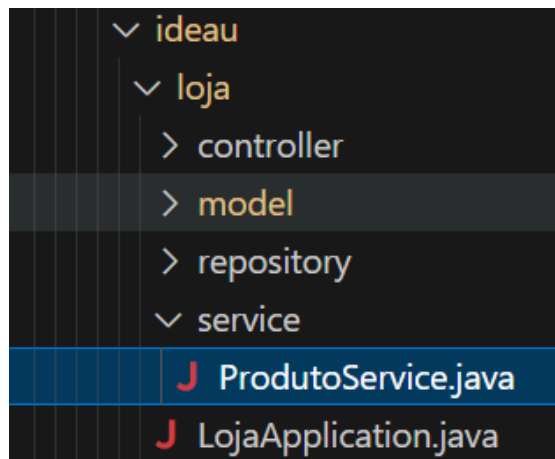
```
1 import org.springframework.data.jpa.repository.JpaRepository;
2 import org.springframework.stereotype.Repository;
3
4 import com.ideau.loja.model.Produto;
5
6 @Repository
7 public interface ProdutoRepository extends JpaRepository<Produto, Long> {
8 }
```

15. 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 “service” e dentro dela um arquivo chamado “ProdutoService.java”:



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

```
1 @Service
2 public class ProdutoService {
3
```

16. 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 “ProdutoService.java” e crie um atributo do tipo “produtoRepository”. Através desse atributo, conseguiremos efetuar as ações com a nossa tabela de produtos.

O próximo passo é criarmos um método para realizarmos o cadastro de produtos.

```
1 @Service
2 public class ProdutoService {
3
4     @Autowired
5     private ProdutoRepository produtoRepository;
6
7     public Produto salvarProduto(Produto produto) {
8         return produtoRepository.save(produto);
9     }
}
```

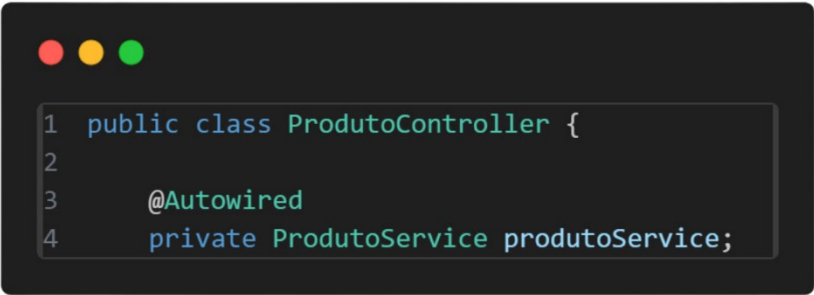
17. Autowired:

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 **ProdutoControle.java** e adicione o seguinte código antes das definições das rotas:



```
1 public class ProdutoController {
2
3     @Autowired
4     private ProdutoService produtoService;
```

18. Cadastrar um produto:

O método `salvarProduto()` 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 `salvarProduto()` já estará disponível para realizar operações de cadastro ou atualização.

Para utilizar o método `salvarProduto()`, 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 `salvarProduto()` 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:

```

1 @RestController
2 @RequestMapping("/api/produtos")
3 public class ProdutoController {
4
5     @Autowired
6     private ProdutoService produtoService;
7
8     // Adicionar novo produto
9     @PostMapping
10    public Produto adicionarProduto(@RequestBody Produto produto) {
11        return produtoService.salvarProduto(produto);
12    }

```

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 **ProdutoService.java** e faça a seguinte implementação:

```

1 public List<Produto> listarProdutos() {
2     return produtoRepository.findAll();
3 }

```

Agora podemos ir no arquivo de **ProdutoControle.java** e criar uma rota para efetuar a seleção:

```

1 // Listar todos os produtos
2 @GetMapping
3 public List<Produto> listarProdutos() {
4     return produtoService.listarProdutos();
5 }
6

```

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.

Agora, mãos à obra! Vamos ao nosso arquivo **ProdutoService.java** e criar o método **buscarProdutoPorId()**.

```

1 // Adicionando o método para buscar produto por ID
2 public Optional<Produto> buscarProdutoPorId(Long id) {
3     return produtoRepository.findById(id);
4 }

```

Agora vamos criar nossa rota no arquivo de **ProdutoControle.java**:

```

1 // Buscar produto por ID
2 @GetMapping("/{id}")
3 public Optional<Produto> buscarProdutoPorId(@PathVariable Long id) {
4     return produtoService.buscarProdutoPorId(id);
5 }

```

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 no arquivo **ProdutoService.java**:

```

1 // Adicionando o método para atualizar o produto
2 public Produto atualizarProduto(Long id, Produto produtoAtualizado) {
3     Optional<Produto> produtoExistente = produtoRepository.findById(id);
4
5     if (produtoExistente.isPresent()) {
6         Produto produto = produtoExistente.get();
7         produto.setNome(produtoAtualizado.getNome());
8         produto.setPreco(produtoAtualizado.getPreco());
9         produto.setCategoria(produtoAtualizado.getCategoria());
10        return produtoRepository.save(produto);
11    } else {
12        throw new RuntimeException("Produto não encontrado com o ID: " + id);
13    }
14 }

```

Em seguida, vamos abrir o arquivo **ProdutoController.java** e adicionar a rota para a função do service.

```

1 // Atualizar produto existente
2 @PutMapping("/{id}")
3 public Produto atualizarProduto(@PathVariable Long id, @RequestBody Produto produto) {
4     return produtoService.atualizarProduto(id, produto);
5 }

```

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 **ProdutoService.java**, adicione o seguinte trecho de código:

```
1     public void removerProduto(Long id) {  
2         produtoRepository.deleteById(id);  
3     }
```

Em seguida, chamamos essa função no arquivo **ProdutoController.java**

```
1     // Remover produto por ID  
2     @DeleteMapping("/{id}")  
3     public void removerProduto(@PathVariable Long id) {  
4         produtoService.removerProduto(id);  
5     }
```