

## Criar uma nova aplicação Java Spring Boot.

Link no [DONTPAD.com/poo3multiversa](https://DONTPAD.com/poo3multiversa)

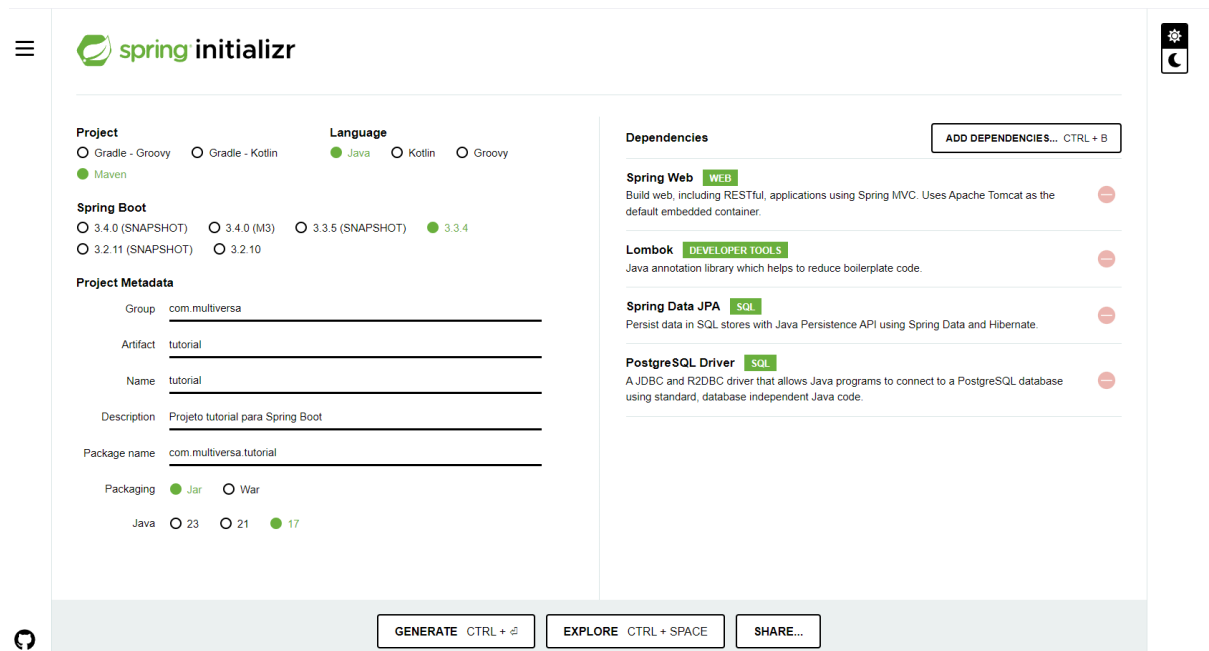
Vamos usar a IDE Java através do IntelliJ para construir a aplicação. Primeiro, vamos definir as camadas que serão criadas.

Camada Controller - ela lida com todas as solicitações dos usuários vindas dos clientes.

Camada Service - ela lida com a lógica de negócios.

Camada Repository - ela se comunica com o banco de dados e depois envia as respostas de volta para o cliente.

**Passo 1:** Vamos criar um projeto esqueleto Spring Boot usando o Spring Initializr. Naveguem até o link do Spring Initializr e preencham com os detalhes abaixo:



The screenshot shows the Spring Initializr web interface. On the left, there's a sidebar with a hamburger menu icon. The main area is divided into sections: 'Project' with radio buttons for 'Gradle - Groovy', 'Gradle - Kotlin', 'Maven' (selected), and 'Language' with radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'; 'Spring Boot' with radio buttons for versions '3.4.0 (SNAPSHOT)', '3.4.0 (M3)', '3.3.5 (SNAPSHOT)', '3.3.4' (selected), and '3.2.11 (SNAPSHOT)', '3.2.10'; 'Project Metadata' with input fields for 'Group' (com.multiversa), 'Artifact' (tutorial), 'Name' (tutorial), 'Description' (Projeto tutorial para Spring Boot), and 'Package name' (com.multiversa.tutorial); and 'Packaging' with radio buttons for 'Jar' (selected) and 'War', and 'Java' with radio buttons for '23', '21', and '17' (selected). On the right, there's a 'Dependencies' section with a button 'ADD DEPENDENCIES... CTRL + B' and three listed dependencies: 'Spring Web' (WEB), 'Lombok' (DEVELOPER TOOLS), and 'Spring Data JPA' (SQL), each with a red minus button. At the bottom, there are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'. A settings icon is visible in the top right corner.

Ou baixem através do seguinte [link](#).

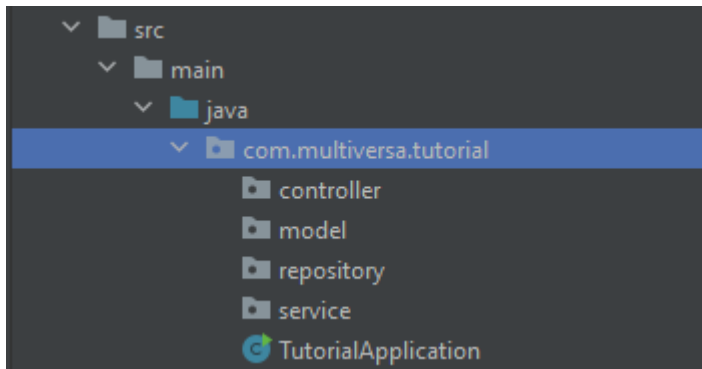
**Passo 2:** Criar os Pacotes no IntelliJ:

Naveguem até o diretório `src/main/java/com.multiversa.tutorial`. Cliquem com o botão direito, selecionem "novo" e procurem por "package"(pacote) para criar um pacote chamado "controller".

Façam o mesmo para criar os outros pacotes: chamados "service", "repository" e "model".

Assim, no total, teremos 4 pacotes criados.

No fim, deve ficar assim:



Para o pacote **model**: Criem uma classe clicando no pacote **model** e selecionando a opção de classe Java chamada **Product**. Insiram o código abaixo:

```
import jakarta.persistence.*;  
import lombok.Data;
```

```
@Entity
```

```
@Table(name = "product_inventory")
```

```
@Data
```

```
public class Product {
```

```
    @Id
```

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

```
    private long id;
```

```
    @Column(name = "product_name", nullable = false)
```

```
    private String productName;
```

```
    @Column(name = "color")
```

```
    private String color;
```

```
    @Column(name = "price")
```

```
    private int price;
```

```
}
```

O código acima define uma classe Java chamada **Product** que representa uma entidade em um banco de dados, com campos que correspondem a colunas em uma tabela (**id**, **productName**, **color**, **price**), anotados com as anotações de Jakarta Persistence (**@Entity**, **@Table**, **@Id**, **@GeneratedValue**, **@Column**), e utiliza a anotação **@Data** do

Lombok para gerar automaticamente o **código boilerplate**\*. A tabela associada a essa entidade no banco de dados é chamada **"product\_inventory"**.

*O que é boilerplate code HTML? É um código padrão que serve como ponto de partida para o desenvolvimento de páginas web. Ele inclui elementos essenciais como a estrutura básica do documento, metadados, links para folhas de estilo CSS e scripts JavaScript, facilitando a criação rápida e consistente de websites.*

Para o pacote **controller**: Criem duas classes chamadas HealthController e ProductController.

HealthController

Clique com o botão direito no pacote controller, selecione "novo" e crie a classe HealthController. Insiram o seguinte código:

### HealthController.java

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

@RestController

```
public class HealthController {
```

```
    @GetMapping("/")
```

```
    public ResponseEntity<?> ebHealth() {
        return new ResponseEntity<>(HttpStatus.OK);
    }
```

```
    @GetMapping("/health")
```

```
    public ResponseEntity<?> health() {
        return new ResponseEntity<>(HttpStatus.OK);
    }
```

```
}
```

-----

Descrição: O código acima define um controlador REST do Spring Boot chamado HealthController, com dois endpoints ("/" e "/health"), cada um respondendo com um status HTTP 200 OK quando acessado.

-----

### ProductController.java

```
import com.multiversa.tutorial.model.Product;
import com.multiversa.tutorial.service.ProductService; // vai dar erro por enquanto
import org.springframework.beans.factory.annotation.Autowired;
```

```

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/products")
public class ProductController {

    @Autowired
    private ProductService productService;

    @PostMapping("/product")
    public ResponseEntity<Product> saveProduct(@RequestBody Product product) {
        Product savedProduct = productService.saveProduct(product);
        return ResponseEntity.ok(savedProduct);
    }

    @GetMapping("/product")
    public ResponseEntity<Product> getProductById(@PathVariable long id) {
        Product product = productService.getProduct(id);
        return ResponseEntity.ok(product);
    }

    @GetMapping("/products")
    public List<Product> getAllProducts() {
        return productService.getProducts();
    }

    @PutMapping("/product")
    public ResponseEntity<Product> updateProduct(@PathVariable long id, @RequestBody
Product product) {
        Product updatedProduct = productService.updateProduct(id, product);
        return ResponseEntity.ok(updatedProduct);
    }

    @DeleteMapping("/product")
    public ResponseEntity<Void> deleteProduct(@PathVariable long id) {
        productService.deleteProduct(id);
        return ResponseEntity.noContent().build();
    }

    @GetMapping("/products-by-name")
    public List<Product> getProductsByName(@RequestParam String name) {
        return productService.getProductsByName(name);
    }
}

```

-----

Este código define um controlador REST chamado ProductController com operações CRUD para gerenciar produtos:

/product: Inserir um novo produto (método POST).  
/product: Obter um produto pelo seu ID (método GET).  
/products: Obter todos os produtos (método GET).  
/product: Atualizar um produto existente (método PATCH).  
/product: Excluir um produto existente (método DELETE).  
/products-by-name: Obter produtos pelo nome usando uma consulta SQL bruta.

Cada uma dessas operações interage com o serviço ProductService para manipular os dados no banco.

-----

Aqui está o código para a classe ProductService no pacote **service**.

### **ProductService.java**

```
import com.multiversa.tutorial.model.Product;
import com.multiversa.tutorial.repository.ProductRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

    // Método para salvar um produto no banco de dados
    public Product saveProduct(Product product) {
        return productRepository.save(product);
    }

    // Método para obter um produto específico pelo seu ID
    public Product getProduct(long productId) {
        return productRepository.findById(productId).orElseThrow(() -> new
        RuntimeException("Produto não encontrado"));
    }

    // Método para obter todos os produtos no banco de dados
    public List<Product> getProducts() {
        return productRepository.findAll();
    }
}
```

```

    }

    // Método para atualizar um produto existente no banco de dados
    public Product updateProduct(long productId, Product product) {
        Product existingProduct = productRepository.findById(productId).orElseThrow(() -> new
        RuntimeException("Produto não encontrado"));
        existingProduct.setProductName(product.getProductName());
        existingProduct.setColor(product.getColor());
        existingProduct.setPrice(product.getPrice());
        productRepository.save(existingProduct);
        return existingProduct;
    }

    // Método para excluir um produto existente no banco de dados
    public Product deleteProduct(long productId) {
        Product existingProduct = productRepository.findById(productId).orElseThrow(() -> new
        RuntimeException("Produto não encontrado"));
        productRepository.deleteById(productId);
        return existingProduct;
    }

    // Método para obter produtos pelo nome usando uma consulta personalizada
    public List<Product> getProductsByName(String productName) {
        return productRepository.getProductsByName(productName);
    }
}

```

-----

O código acima define uma classe de serviço do Spring Boot chamada ProductService que encapsula a lógica de negócios para gerenciar produtos. Ela utiliza um ProductRepository para interagir com o banco de dados, fornecendo métodos para:

Salvar um produto.

Recuperar um produto pelo seu ID.

Listar todos os produtos.

Atualizar um produto existente.

Excluir um produto.

Recuperar produtos pelo nome usando uma consulta SQL personalizada.

Além disso, o serviço lida com exceções quando um produto com um ID específico não é encontrado, lançando uma exceção RuntimeException com a mensagem adequada.

-----

Em seguida, crie o pacote **repository**: Crie uma **interface**, clique com o botão direito no pacote repository, selecione interface e nomeie-a como ProductRepository. É uma estrutura básica para interagir com o banco de dados.

## ProductRepository.java

```
import org.springframework.data.jpa.repository.JpaRepository;
import com.multiversa.tutorial.model.Product;
import org.springframework.data.jpa.repository.Query;

import java.util.List;

public interface ProductRepository extends JpaRepository<Product, Long> {

    // Consulta personalizada para obter produtos pelo nome
    @Query(value = "SELECT * FROM product_inventory WHERE product_name = ?",
nativeQuery = true)
    List<Product> getProductsByName(String productName);
}
```

-----

Este código define uma interface de repositório do Spring Data JPA chamada `ProductRepository`, que estende `JpaRepository`. Isso fornece as operações CRUD básicas para a entidade `Product`, como salvar, atualizar, excluir e buscar registros no banco de dados. Além disso, a interface inclui um método personalizado `getProductsByName`, anotado com `@Query`, que permite a recuperação de uma lista de produtos pelo nome usando uma consulta SQL nativa.

Com isso, o repositório será capaz de se comunicar diretamente com o banco de dados e buscar produtos com base no nome, facilitando a implementação de consultas personalizadas quando necessário.

-----

**Passo 3:** Abra o pgAdmin em sua máquina local e insira o código abaixo para criar um banco de dados:

```
CREATE DATABASE store_database;
```

Em seguida, crie a tabela `product_inventory` e defina os campos necessários conforme mostrado abaixo:

```
\c store_database;
```

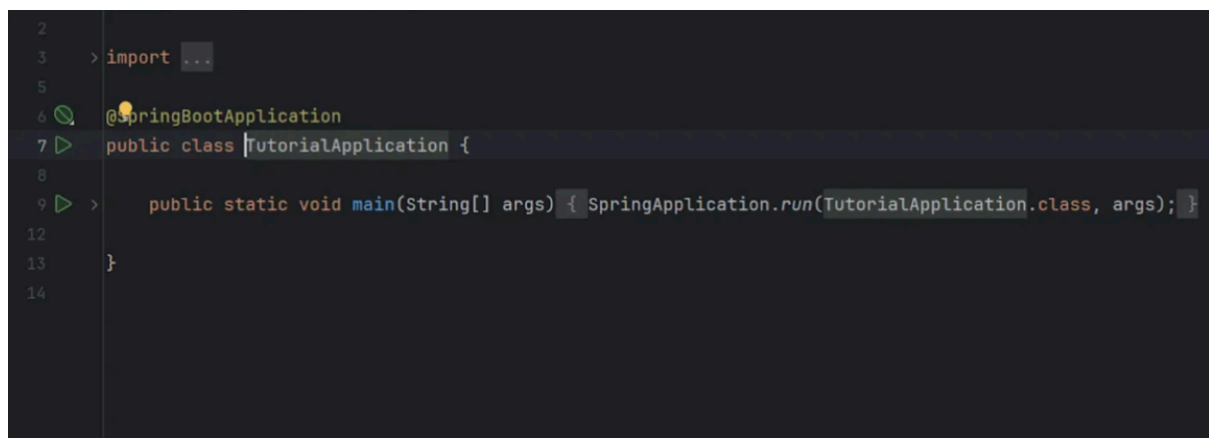
```
CREATE TABLE IF NOT EXISTS product_inventory (
    id BIGSERIAL NOT NULL PRIMARY KEY,
    product_name VARCHAR(32) NOT NULL,
    color VARCHAR(10),
    price INT
);
```

Agora, especifique o nome do banco de dados e o nome da tabela em nosso código Java. Navegue até a pasta resources, procure pelo arquivo application.properties e insira o código abaixo. Lembre-se de inserir seu nome de usuário e senha do banco de dados:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/store_database
spring.datasource.username=postgres
spring.datasource.password=@insirasenhadoPostgresqlAqui#(costuma ser host)
```

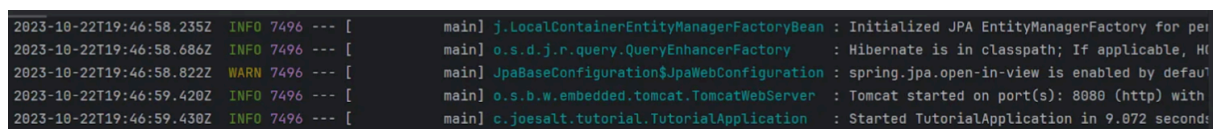
#### Passo 4: Teste a aplicação

Volte para o IntelliJ e selecione o arquivo TutorialApplication.java. Em seguida, clique no botão de play à esquerda (linha 7), conforme mostrado abaixo:



```
2
3 > import ...
4
5
6 @SpringBootApplication
7 public class TutorialApplication {
8
9 > public static void main(String[] args) { SpringApplication.run(TutorialApplication.class, args); }
10
11
12
13 }
14
```

Se aparecer isso daqui, então está tudo correto:



```
2023-10-22T19:46:58.235Z INFO 7496 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-10-22T19:46:58.686Z INFO 7496 --- [main] o.s.d.j.r.query.QueryEnhancerFactory : Hibernate is in classpath; If applicable, HQL parser will be used.
2023-10-22T19:46:58.822Z WARN 7496 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default; This will result in memory being used by the view layer, and may cause issues on multi-user systems with pagination.
2023-10-22T19:46:59.420Z INFO 7496 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path: /
2023-10-22T19:46:59.430Z INFO 7496 --- [main] c.joesalt.tutorial.TutorialApplication : Started TutorialApplication in 9.072 seconds
```

Utilizando o Postman WEB, teste os caminhos.  
Teste o caminho health: insira o seguinte.



HTTP <http://localhost:8080/health> Save

GET <http://localhost:8080/health> Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Bulk Edit
Key	Value	

Body Cookies Headers Test Results 200 OK 274 ms 123 B Save Response

Pretty Raw Preview Visualize Text

1

**Crie um novo produto:** Adicione um novo produto ao banco de dados fazendo uma solicitação POST para localhost:8080/product com o corpo abaixo como body da solicitação:

HTTP <http://localhost:8080/product>

POST <http://localhost:8080/product>

Params Authorization Headers (7) Body  Scripts Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON

```
1 {
2   "productName": "Tinta Guache",
3   "color": "azul",
4   "price": 2500
5 }
```