Construindo uma API REST para Cadastro de Produtos

1. Introdução aos Conceitos de REST

O que é REST?

REST (Representational State Transfer) é um estilo arquitetural para projetar APIs web, baseado em princípios que garantem simplicidade, escalabilidade e interoperabilidade. Ele utiliza o protocolo HTTP para realizar operações em **recursos**, que são entidades (como um "Produto") identificadas por URLs únicas.

Princípios fundamentais do REST:

- **Recursos**: Cada entidade é um recurso, acessado por uma URL (ex.: `/produtos/1` para o produto com ID 1).
- **Métodos HTTP**: Operações CRUD (Create, Read, Update, Delete) são mapeadas para métodos HTTP:
  - **GET**: Recupera dados (ex.: listar produtos).
  - **POST**: Cria um novo recurso (ex.: cadastrar um produto).
  - **PUT**: Atualiza um recurso existente (ex.: atualizar um produto).
  - **DELETE**: Remove um recurso (ex.: deletar um produto).
- **Stateless**: Cada requisição é independente, contendo todas as informações necessárias.
- **Formato de dados**: Geralmente JSON, devido à sua simplicidade e compatibilidade.
- **Códigos de status HTTP**: Respostas usam códigos padrão (ex.: 200 OK, 201 Created, 404 Not Found).
- **HATEOAS** (opcional): Respostas podem incluir links para outras ações.

Exemplo de URLs RESTful:

- `GET /produtos` : Lista todos os produtos.
- `GET /produtos/1` : Recupera o produto com ID 1.
- `POST /produtos` : Cria um novo produto.
- `PUT /produtos/1` : Atualiza o produto com ID 1.
- `DELETE /produtos/1` : Deleta o produto com ID 1.

Por que Spring Boot?

Spring Boot é um framework Java que simplifica o desenvolvimento de APIs REST, oferecendo:

- Configuração automática, reduzindo código boilerplate.
- Servidor embutido (ex.: Tomcat).
- Suporte a REST com anotações como `@RestController`.
- Integração com bancos de dados via Spring Data JPA.

Por que H2 com Persistência em Arquivo?

O **H2 Database** é um banco de dados leve, ideal para desenvolvimento e testes. Por padrão, ele opera em memória, mas pode ser configurado para persistir dados em um arquivo (ex.: `produtos.db`). Isso combina a simplicidade do H2 com a capacidade de manter dados entre execuções, sendo uma alternativa ao SQLite para este projeto.

2. Objetivo do Projeto

Criar uma API RESTful para gerenciar Produtos, com as seguintes funcionalidades:

- **Listar todos os produtos** (GET `/produtos`).
- **Buscar um produto por ID** (GET `/produtos/{id}`).
- **Cadastrar um novo produto** (POST `/produtos`).
- **Atualizar um produto existente** (PUT `/produtos/{id}`).
- **Deletar um produto** (DELETE `/produtos/{id}`).

Recurso: Produto, com atributos:

- `id` (Long): Identificador único, auto-incrementado.
- `nome` (String): Nome do produto.
- `preco` (Double): Preço do produto.

Tecnologias:

- Spring Boot 3.2.5.
- Java 17.
- H2 Database (persistência em arquivo).
- Maven.

3. Configurando o Ambiente

Pré-requisitos

- **Java 17** ou superior.
- **Maven**.
- IDE (ex.: IntelliJ IDEA, Eclipse, VS Code).
- Ferramenta para testar APIs (ex.: Postman, Insomnia, cURL).

Criar o Projeto Spring Boot

1. Acesse **Spring Initializr** (https://start.spring.io/) ou use sua IDE.
2. Configure:
    - **Project**: Maven.
    - **Language**: Java.
    - **Spring Boot**: 3.2.5 (ou versão estável mais recente).
    - **Group**: `com.example`.
    - **Artifact**: `produto-api`.
    - **Dependencies**:
        - **Spring Web**: Para APIs REST.
        - **Spring Data JPA**: Para integração com banco.
        - **H2 Database**: Para o banco H2.
        - **Lombok**: Para reduzir boilerplate.
3. Baixe, descompacte e importe na IDE.

Estrutura inicial:

```
produto-api/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/example/produtoapi/
│   │   │       └── ProdutoApiApplication.java
│   │   ├── resources/
│   │   │   └── application.properties
├── pom.xml
```

4. Configurando o H2 com Persistência em Arquivo

Teoria: Bancos de Dados e Spring Data JPA

O Spring Data JPA mapeia entidades Java para tabelas no banco. O H2, quando configurado para persistência em arquivo, armazena dados em um arquivo local (ex.: `produtos.mv.db`), permitindo que os dados sejam mantidos entre execuções da aplicação. Isso é útil para desenvolvimento e testes, oferecendo uma alternativa ao SQLite.

Configurar o `application.properties`

Edite `src/main/resources/application.properties` para configurar o H2 em modo de arquivo:

```
spring.datasource.url=jdbc:h2:file:./produtos;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Explicação:

- `spring.datasource.url` : Configura o H2 para salvar dados no arquivo `./produtos` (gerará `produtos.mv.db` na raiz do projeto).
- `DB_CLOSE_ON_EXIT=FALSE` : Mantém o banco aberto durante a execução.
- `spring.datasource.username=sa` e `password=` : Credenciais padrão do H2.
- `spring.jpa.database-platform` : Usa o dialeto nativo do H2.
- `spring.jpa.hibernate.ddl-auto=update` : Cria/atualiza tabelas automaticamente.
- `spring.h2.console.enabled=true` : Habilita o console web do H2, acessível em `http://localhost:8080/h2-console`.

5. Modelando o Recurso Produto

Teoria: Representação de Recursos

Em REST, um recurso é uma entidade (ex.: `Produto`) serializada em JSON para comunicação com o cliente. A entidade Java é mapeada para uma tabela no banco usando anotações JPA.

Criar a Entidade Produto

Crie a classe `Produto` no pacote `com.example.produtoapi.model`:

```
package com.example.produtoapi.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.Data;

@Entity
@Data
public class Produto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nome;
    private Double preco;
}
```

Explicação:

- `@Entity`: Mapeia a classe para a tabela `PRODUTO` no H2.
- `@Id` e `@GeneratedValue`: Define `id` como chave primária auto-incrementada.
- `@Data` (Lombok): Gera getters, setters, `toString`, `equals` e `hashCode`.
- `nome` e `preco` são mapeados como colunas VARCHAR e DOUBLE.

6. Criando o Repositório

Teoria: Camada de Persistência

O repositório gerencia interações com o banco de dados. O Spring Data JPA fornece a interface `JpaRepository`, que implementa operações CRUD automaticamente.

Criar o Repositório

Crie a interface `ProdutoRepository` no pacote `com.example.produtoapi.repository`:

```
package com.example.produtoapi.repository;

import com.example.produtoapi.model.Produto;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ProdutoRepository extends JpaRepository<Produto, Long> {
}
```

Explicação:

- `JpaRepository<Produto, Long>`: Fornece métodos como `save`, `findById`, `findAll` e `delete`.
- O tipo `Long` corresponde ao tipo do campo `id`.

7. Implementando a Camada de Serviço

Teoria: Lógica de Negócio

A camada de serviço contém a lógica de negócio, coordenando chamadas ao repositório e aplicando validações ou regras. Ela atua como intermediária entre o controlador e o repositório.

Criar o Serviço

Crie a classe `ProdutoService` no pacote `com.example.produtoapi.service`:

```java
package com.example.produtoapi.service;

import com.example.produtoapi.model.Produto;
import com.example.produtoapi.repository.ProdutoRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

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

@Service
public class ProdutoService {

    @Autowired
    private ProdutoRepository produtoRepository;

    public List<Produto> listarTodos() {
        return produtoRepository.findAll();
    }

    public Optional<Produto> buscarPorId(Long id) {
        return produtoRepository.findById(id);
    }

    public Produto cadastrar(Produto produto) {
        return produtoRepository.save(produto);
    }

    public Optional<Produto> atualizar(Long id, Produto produto) {
        if (produtoRepository.existsById(id)) {
            produto.setId(id);
            return Optional.of(produtoRepository.save(produto));
        }
        return Optional.empty();
    }

    public boolean deletar(Long id) {
        if (produtoRepository.existsById(id)) {
            produtoRepository.deleteById(id);
            return true;
        }
        return false;
```

```
    }
}
```

Explicação:

- `@Service` : Marca a classe como um componente de serviço.
- `@Autowired` : Injeta o `ProdutoRepository` .
- Métodos implementam operações CRUD, com verificações para `atualizar` e `deletar` .

8. Criando o Controlador REST

Teoria: Camada de Apresentação

O controlador processa requisições HTTP, mapeando URLs e métodos HTTP para ações no serviço. Ele retorna respostas em JSON com códigos de status apropriados.

Criar o Controlador

Crie a classe `ProdutoController` no pacote `com.example.produtoapi.controller` :

```java
package com.example.produtoapi.controller;

import com.example.produtoapi.model.Produto;
import com.example.produtoapi.service.ProdutoService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/produtos")
public class ProdutoController {

    @Autowired
    private ProdutoService produtoService;

    @GetMapping
    public List<Produto> listarTodos() {
        return produtoService.listarTodos();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Produto> buscarPorId(@PathVariable Long id) {
        return produtoService.buscarPorId(id)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping
    public Produto cadastrar(@RequestBody Produto produto) {
        return produtoService.cadastrar(produto);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Produto> atualizar(@PathVariable Long id, @RequestBody Produto produto
        return produtoService.atualizar(id, produto)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deletar(@PathVariable Long id) {
```

```
        if (produtoService.deletar(id)) {
            return ResponseEntity.noContent().build();
        }
        return ResponseEntity.notFound().build();
    }
}
```

Explicação:

- `@RestController` : Define a classe como um controlador REST, serializando respostas em JSON.
- `@RequestMapping("/produtos")` : Prefixo para todas as URLs.
- `@GetMapping` , `@PostMapping` , etc.: Mapeiam métodos HTTP.
- `@PathVariable` : Extrai o `id` da URL.
- `@RequestBody` : Converte o JSON do corpo da requisição em um objeto `Produto` .
- `ResponseEntity` : Controla o status HTTP (ex.: 200, 404, 204).

Exemplo de JSON para POST ou PUT:

```
{
    "nome": "Notebook",
    "preco": 3500.00
}
```

9. Testando a API

Executar a Aplicação

1. Execute a classe `ProdutoApiApplication` na IDE ou com:

   ```
   mvn spring-boot:run
   ```

2. A aplicação estará disponível em `http://localhost:8080` .
3. O arquivo `produtos.mv.db` será criado na raiz do projeto, persistindo os dados.

Acessar o Console H2

1. Acesse `http://localhost:8080/h2-console` .
2. Configure:

- **JDBC URL**: `jdbc:h2:file:./produtos`.
- **User Name**: `sa`.
- **Password**: (em branco).

3. Conecte e verifique a tabela `PRODUTO`.

Testar os Endpoints

Use Thunder Client (VSCode), Postman, Insomnia ou cURL para testar as rotas.

Exemplos de Requisições:

1. **Listar todos os produtos**:

   ```
   GET http://localhost:8080/produtos
   ```

   Resposta esperada (exemplo):

   ```
   [
       {"id": 1, "nome": "Smartphone", "preco": 2000.00},
       {"id": 2, "nome": "Notebook", "preco": 3500.00}
   ]
   ```

2. **Cadastrar um produto**:

   ```
   POST http://localhost:8080/produtos
   Content-Type: application/json

   {
       "nome": "Smartphone",
       "preco": 2000.00
   }
   ```

   Resposta esperada:

   ```
   {"id": 1, "nome": "Smartphone", "preco": 2000.00}
   ```

3. **Buscar um produto por ID**:

   ```
   GET http://localhost:8080/produtos/1
   ```

   Resposta esperada:

   ```
   {"id": 1, "nome": "Smartphone", "preco": 2000.00}
   ```

4. **Atualizar um produto**:

```
PUT http://localhost:8080/produtos/1
Content-Type: application/json

{
    "nome": "Smartphone Pro",
    "preco": 2500.00
}
```

   Resposta esperada:

```
{"id": 1, "nome": "Smartphone Pro", "preco": 2500.00}
```

5. **Deletar um produto**:

```
DELETE http://localhost:8080/produtos/1
```

   Resposta esperada: Status 204 (No Content).

Códigos de status HTTP:

- 200 (OK): GET ou PUT bem-sucedido.
- 201 (Created): POST bem-sucedido (pode ser ajustado no controlador).
- 204 (No Content): DELETE bem-sucedido.
- 404 (Not Found): Recurso não encontrado.