Projeto API RestFul com Spring

API CRUD de Products utilizando:

Spring Boot 3 e Java 17.

Spring Web MVC (Arquitetura).

Spring Data JPA (Ligação e mapeamento com o BD).

Spring Validation (DTO’s).

Spring Hateoas.  
  
Configuração do projeto feita pelo Spring Initializr:  
Project: **Maven**, Language: **Java 17**, Springboot: **3.3.1**, Packaging: **Jar**.

* **Conexão com o Banco de Dados:**  
  O banco de dados utilizando no projeto foi o PostgreSQL.

Foi criada uma tabela chamada “tb products” e vamos criar as colunas e nosso mapeamento por meio do JPA.

---------------------------------------------------------------------------------------------------------

* **Aplication.properties**

Já com o projeto aberto na IDE, vamos no arquivo chamado “aplication.properties”, localizado no package “resources”, e fazemos o seguintes passos:

spring.datasource.url=jdbc:postgresql://localhost:5432/bdrestful

- Passamos a nossa conexão da nossa máquina local via URL para o banco de dados, passando a porta default e logo em seguida o nome da tabela/base de dados.

Logo em seguida definimos as nossas credenciais (credenciais do admin).

spring.datasource.username=postgres  
spring.datasource.password=victormacedo

- Passamos o nome e a senha de administrador para a conexão com o banco de dados.

spring.jpa.hibernate.ddl-auto=update

- Todo o mapeamento que fizermos para a nossa base de dados, por exemplo o Model, ele é criado através de tabelas de forma automática, logo quando rodamos a aplicação, caso ele ainda não exista, ele passa a existir. Caso haja alguma alteração, ele fará as alterações necessárias.

spring.jpa.properties.hibernate.jdbc.lob.non\_contextual\_creation=true

- Para evitar que o Hibernate gere logs relacionados à busca de metadados do Postgres durante a inicialização e execução da aplicação, você pode ajustar a configuração de logging do Hibernate para controlar o nível de detalhe dos logs exibidos.

O Hibernate, por padrão, pode gerar mensagens de log para mostrar detalhes sobre o mapeamento de entidades (classes Java) para tabelas do banco de dados, entre outros processos. Isso inclui consultas que ele executa internamente para obter informações sobre a estrutura do banco de dados.

Para controlar esses logs, você pode configurar o nível de logging do Hibernate.

---------------------------------------------------------------------------------------------------------

Após essa configuração com a base de dados, podemos partir para a modelagem da nossa API Restful, onde começamos criando nossos packages, models e mapeamento de entidades.

[Seguindo sempre o modelo de maturidade de Richardson para API’s Rest.](https://rivaildojunior.medium.com/modelo-de-maturidade-de-richardson-para-apis-rest-8845f93b288)

- Package: models

* **ProductModel**

Ela vai ser responsável modelar nossos produtos, tendo seu próprio ID, nome e valor.

Iniciando a nova classe e para que ela seja uma entidade na base de dados, vamos ter que importar algumas notations.

**@Entity** – declarar que vai ser uma entidade.  
**@Table(name = “TB PRODUCTS”)** – o nome da nossa tabela.

public class ProductModel implements Serializable {  
 private static final long *serialVersionUID* = 1L;  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*AUTO*)  
 private UUID idProduct;

- Implementamos a interface **Serializable** para a classe ficar habilitada para passar por serializações, ou seja, serialização é o processo de transformar objetos em uma forma que pode ser facilmente armazenada, transmitida ou recuperada, e ao implementar a interface Serializable, você habilita essa funcionalidade para objetos de sua classe, auxiliando na persistência de dados.

* Definimos também o **serialVersionUUID**, que é o número de controle de versão de cada uma dessas classes que forem serializadas quando houver a necessidade.

Inicializamos os atributos que foram citados acima, porém com atenção ao ID, que vai ter algumas notations específicas para ele.

**@Id** – Demonstrar o nosso ID na base de dados

**@GeneratedValue(strategy = GenerationType.AUTO)** – Como vai ser gerado esse tipo de ID, ou seja, eu não vou precisar importar um valor para o ID, pois ele já vai ser gerado de forma aleatória e única.

**Identificador do tipo UUID** – É um tipo de identificador utilizado na arquitetura de microsservices, pois ele evita diversos conflitos, já que eles são universais e únicos, ou seja, não vamos correr o risco de terem UUID’s semelhantes.

E no fim nós podemos gerar os métodos Getters e Setters, porém podemos utilizar as notations **@Getter** e **@Setter** da Dependência Lombok, sendo essa responsável por nos ajudar com alguns boilerplates.

Rodamos a aplicação e vamos para a criação do Repository.

OBS: Ele já inicia nossa aplicação no TOMCAT (localhost) com a porta default 8080.

---------------------------------------------------------------------------------------------------------

- package: repository

* **ProductRepository**

Aqui vamos abstrair qual vai ser a entidade que vai ter acesso aos dados da aplicação, fornecendo os métodos para realizar operações básicas de persistência (Get, Post, Update, Delete, etc).

@Repository  
public interface ProductRepository extends JpaRepository<ProductModel, UUID> {}

- Vamos extender o **JpaRespository** a classe, passando a entidade/module que ele vai contemplar e qual que é o tipo de identificador que ele utiliza.  
**@Repository** – é um Bean do próprio Spring indicando que vai ser gerenciado por ele de acordo com a notation.

---------------------------------------------------------------------------------------------------------

-package: controller

* **ProductController**

É considerado um ponto de injeção para a base de dados, para iniciarmos isso, usamos:

@Autowired  
ProductRepository productRepository;

- Criado para que nós tenhamos acesso a todos os métodos JPA quando houver necessidade.

Tendo nossos métodos do JPA importados em nosso controller, nós podemos implementar o nosso CRUD, mas antes de recebermos, por exemplo, um Put na nossa aplicação, é de extrema importância que mapeemos estes campos, utilizando os Records.

* Records: é uma forma de declarar classes simples imutáveis para armazenar dados de semelhante as structs, de forma clara e concisa.

Ela é muito utilizada na criação de DTO’s, pois é nela que passamos como queremos receber nosso objeto para a injeção na base de dados, colocando, por exemplo, que o campo de nome e de valor não podem estar vazios ou sem nenhuma atribuição de valor, respectivamente.

**@RestController** – é um Bean do próprio Spring indicando que vai ser gerenciado por ele de acordo com a notation.

-package: dtos

* **ProductRecordDto** (especificamos como Record na hora da criação)

public record ProductRecordDto(@NotBlank String name, @NotNull BigDecimal value) {}

Pensando assim: O que o cliente vai enviar para a base de dados para ser armazenado?

O ID você não precisa se preocupar pois ele vai ser gerado de forma automática por conta do UUID, então vamos precisar do nome do produto e do seu preço.

Logo, o nome e o preço do produto eles precisam estar preenchidos para que ocorra tudo bem na hora da injeção desse objeto na base de dados, por isso utilizamos as notations de validação **@NotBlank**, para que quando for dar o POST desse objeto com o campo “name” vazio/nulo ocorra um erro, a mesma coisa com a notation **@NotNull**, para que quando for da o POST desse objeto com o campo “value” vazio/nulo ocorra um erro.

Assim os DTO’s eles são de extrema importância para criarmos “regras” na hora da injeção de dados na nossa base.

Implementação dos métodos POST, GET (getAll e getOne), PUT e DELETE

OBS: Para fazermos as chamadas desses métodos, utilizei os Postman para fazer o teste da minha API.

// POST: Vai ser o método de criar/salvar produtos na base de dados.

**@PostMapping(“/products”)** – Declaramos o método POST juntamente com o protocolo HTTP de forma semântica e com sua devida URI.

@Autowired  
ProductRepository productRepository;

- É o ponto de injeção da base de dados.

public ResponseEntity<ProductModel> saveProduct(@RequestBody @Valid ProductRecordDto productRecordDto) {}

- ResponseEntity<ProductModel> - é usado para encapsular a resposta HTTP completa que um método de controle Spring pode retornar, incluindo status, cabeçalhos e o corpo da resposta que é do tipo ProductModel. Seria o retorno do método saveProduct

- saveProduct – é o nome do método.

Esse método ele receber, como corpo da solicitação (**@RequestBody**, ele recebe no formato JSON de lá do Postman) um objeto que passou pelo DTO e que vai aplicar os métodos de validação que são colocados nos campos do DTO (**@Valid**).

Então quando recebemos um productRecordDto, ou melhor, o “objeto DTO”, nós inicializamos um productModel e copiamos todas as suas propriedades e transformamos esse JSON em um “objeto Java/model”, ou melhor, assim para que seja salvo em nossa base de dados.

var productModel = new ProductModel();  
BeanUtils.*copyProperties*(productRecordDto, productModel);  
return ResponseEntity.*status*(HttpStatus.*CREATED*)

.body(productRepository.save(productModel));

- Resumindo, nós recebemos um objeto (JSON) que passa pelo DTO e passa a ser um “objeto DTO”, nele é aplicado sua validações, e após isso ele é convertido em um “objeto Model”, pois ele só se mantém como um “objeto DTO” para fazer essas verificações, e subimos esse model para a base de dados.

Nós retornamos o ResponseEntity, onde ele é construído em duas partes (status (retorno do HTTP) e body(nós podemos retornar o que foi salvo(ID, nome, valor) e aplicar na nossa base de dados via productRepository.save(productModel)).

// GET (GetAll e GetOne)

- GetAll: Vai listar todos os produtos que temos na base de dados.

**@GetMapping(“/products”)** – vai ser acionado com essa URI.

public ResponseEntity<List<ProductModel>> getAllProducts(){  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(productRepository.findAll());  
}

- Seu retorno vai ser um ResponseEntity, porém seu corpo vai ser uma lista de ProductModel.

- Ele também vai retornar o status (HTTP status OK, 200) e o body vai ser pegar todos esses produtos pela injeção de dados (productRepository) direto da base de dados.

- GetOne: Vai listar um produto específico por meio de seu ID, por exemplo.

**@GetMapping(“/products/{id}”)** – vai ser acionado por essa URI.

public ResponseEntity<Object> getOneProduct(@PathVariable(value="id") UUID id){  
 Optional<ProductModel> product0 = productRepository.findById(id);  
 if(!product0.isEmpty()){  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(product0.get());  
 }  
 return ResponseEntity.*status*(HttpStatus.*NOT\_FOUND*).body("Product not found");  
}

- Ele vai retornar um ResponseEntity, porém vai ser Object, pois vamos ter dois tipos de resposta/retorno (se existe esse produto ou não)

- Dentro do método nós vamos utilizar da notation @PathVariable para que a gente consiga obter o ID do produto passando o nome correto da URI e vamos ter também o tipo desse ID.

- Antes de tudo, vamos na base de dados encontrar esse produto por meio de seu id e que se encontrarmos, nós vamos criar um objeto do tipo Optional.

E o retorno vai ser bem parecido com o que já vimos até aqui, claro, cada um dentro da sua condicional.

// PUT – Vai fazer o update do objeto.

**@PutMapping(“products/{id}”)**

- Vai ter um retorno do ResponseEntity do tipo Object, pois pode ter retornos diferentes para retornos diferentes.

- Vai usar das notations @PathVariable para pegar seu ID do tipo UUID e ele também vai receber o corpo da requisição, pois sem ele não tem como fazer o update(name e value).

- Antes de atualizarmos, vamos verificar, por meio do ID, na base de dados se esse objeto existe, ou seja, criamos um objeto do tipo Optional(product0) que vai fazer um findById na nossa base de dados por meio do ponto de injeção (productRepository) e vai cair nas condicionais.

- Caso exista, vai se criar uma variável local (productModel) que vai estar relacionado ao product0 que foi solicitado lá em cima para fazer a pesquisa na base de dados.  
Com isso, vamos fazer a atualização em si:  
- Pegamos os campos atualizados (productRecordDto) e passamos para o objeto já existente (productModel), lembrando que nesse momento ele já está pronto para entrar na base de dados, pois ele não está mais como corpo da requisição por conta do BeanUtils.copyProperties.

E como resposta vamos seguir os padrões anteriores (status e body).

// DELETE – Vai deletar um objeto por meio de seu ID.

A estrutura do código vai ser bastante parecida com o que nós vimos anteriormente.

**@UpdateMapping(“products/{id}”)** – Vai ser a URI de acesso.

Vamos criar uma variável para procurar o produto na base de dados, caso ele não exista emitimos uma mensagem (status e body) e caso ele exista, pegamos esse objeto com o ID e deletamos direto na base de dados e retornamos uma mensagem (status e body).

**Implementação de HATEOAS e criação de hipermídias**

Vamos implementar o Hateoas para que terminemos de seguir os níveis de maturidade de Richardson para que nossa API seja considerado Restful.

As hipermídias auxiliam o usuário na navegabilidade dos recursos, mas sempre tendo uma API bem documentada.  
Lembrando que o Hateoas não é muito utilizado no dia a dia, pois dependendo da complexidade ele pode ser limitado, porém para o foco do projeto vamos implementar ele.

E agora para tornar essa API de fato Restful, vamos ter que fazer algumas implementações e modificações.

* Verifico nas minhas dependências se o starter hateoas está presente no meu pom.xml.
* Modificamos nosso ProductModel.

public class ProductModel extends RepresentationModel<ProductModel> implements Serializable {}

- Agora quando formos acionar o nosso ProductModel, internamente ele já vai ter alguns métodos para construirmos esses links.

* Alteração nos métodos do ProductController.

// getAllProducts

Agora na nossa API, quando um cliente solicitar a listagem de produtos, vai os atributos mais um atributos, os links.

Mas o que seria esses links?

Esse link vai ser responsável para redirecionar o cliente para os detalhes daquele produto, que nada mais é o nosso método getOneProduct, assim então ele vai analisar e ser redirecionado para a URI que especifica um único produto de acordo com seu UUID.

@GetMapping("/products")  
public ResponseEntity<List<ProductModel>> getAllProducts(){  
 List<ProductModel> productsList = productRepository.findAll();  
 if (!productsList.isEmpty()){  
 for (ProductModel product : productsList) {  
 UUID id = product.getIdProduct();  
 product.add(*linkTo*(*methodOn*(ProductController.class).getOneProduct(id)).withSelfRel());  
 }  
 }  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(productsList);  
}

- Primeiro, a gente cria uma lista de ProductModel separadamente, armazenando ela em uma variável, puxando seus produtos direto da base de dados (productRepository.findAll()) e fazemos algumas verificações.

- Se essa lista de produtos não estiver vazia:

Fazemos um for percorrendo toda essa lista.

Pegamos os id’s de cada determinado produto e armazenamos em uma variável (UUID id).

E utilizamos os métodos do Hateous para começarmos a criar esses links de cada produto listado.

product.add(*linkTo*(

*methodOn*(ProductController.class)

.getOneProduct(id))

.withSelfRel());

**linkTo()** – Para qual método que eu vou redirecionar o cliente quando ele clicar nesse link.

**methodOn(ProductController.class).getOneProduct(id)** – Qual que é o controller que está o método de especificar o produto e qual que esse método em si que vai fazer esse redirecionamento (que vai estar acompanhado do seu id (que é um atributo do método em si), que foi a variável que declaramos dentro do for para ficar mais sugestivo).

Também poderia ser feito dessa forma:

**methodOn(ProductController.class).getOneProduct(product.getIdProduct())**.

**withSelRel()** – Ele vai mostrar que ele então vai ser redirecionado para cada um de seus produtos em si.

E no final nós construímos nossa resposta passando o HttpStatus.OK e nossa lista de produtos.

// getOneProduct

Fazemos a mesma coisa com o método getOneProduct, porém só mostramos a listagem dos produtos caso o produto requisitado não exista.

product0.get().add(*linkTo*(*methodOn*(ProductController.class).getAllProducts()).withSelfRel());