

Notas - Curso Spring Boot Expert: JPA, REST, JWT, OAuth2 com Docker e AWS

INTRODUÇÃO

Quando criamos um projeto, o próprio Springboot cria uma classe especial chamada de `NomeDoProjetoApplication` onde se encontra o nosso método `main`. Dentro do `main` temos o `SpringApplication`.

Pra começar o projeto, precisamos criar as seguintes duas classes, uma representando o modelo de negócios e a outra a nossa API:

Classe Controller: é a classe que vai ser a nossa API, isso é, quem vai fazer a ligação com o banco de dados por meio de requisições HTTP.

Por conta disso, nós fazemos a anotação dessa classe com o `@RestController`.

Também usamos a anotação `@RequestMapping("produtos")`. Essa anotação tem por objetivo definir qual que é a nossa url básica (produtos).

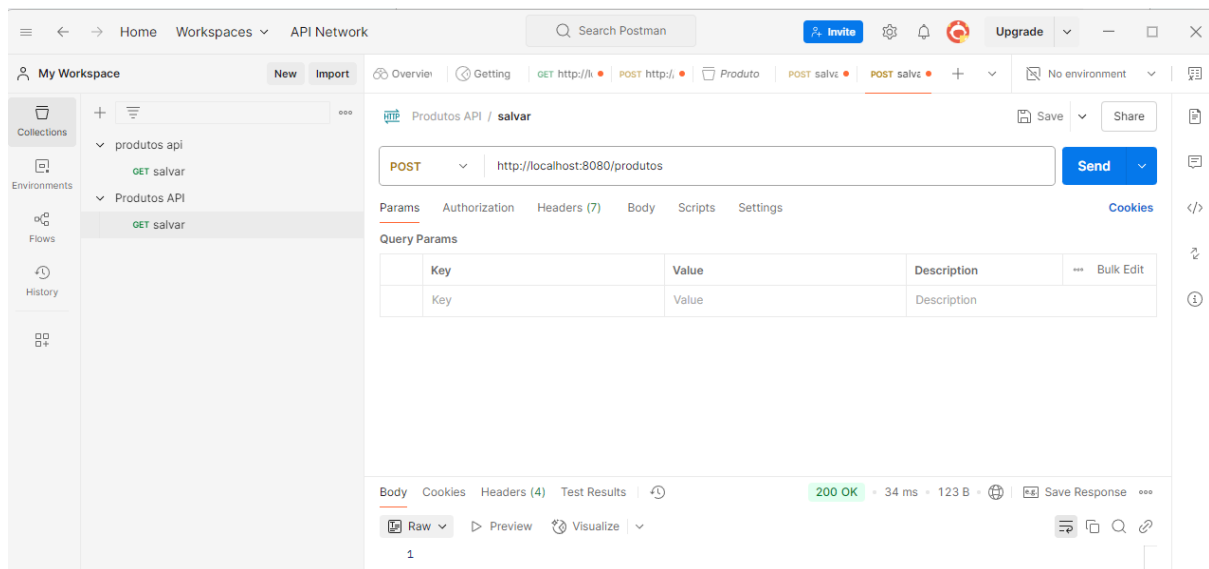
Classe Model: representa o nosso negócio (um usuário e seus atributos)

Tudo o que você envia ou recebe de um servidor rest é chamado de **recurso**

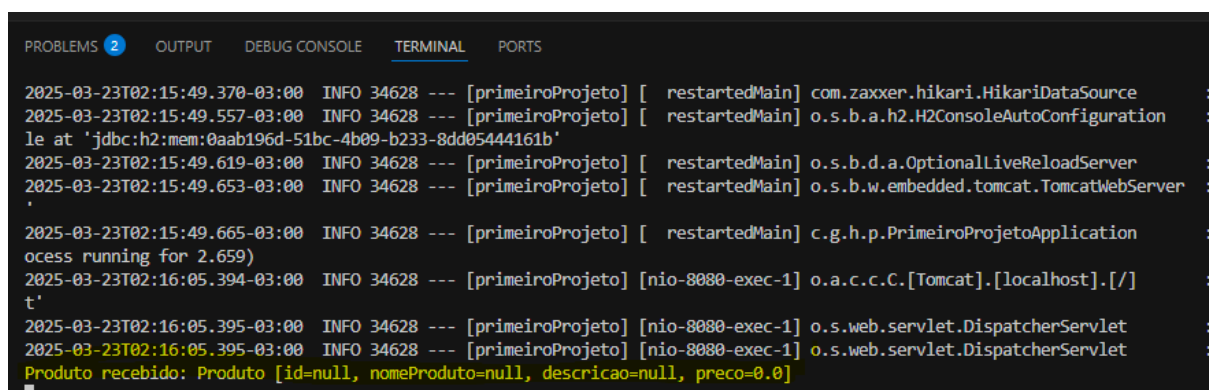
Quando queremos enviar alguma coisa para um servidor, nós usamos o método HTTP `@PostMapping`.

```
8
9  @RestController
10 @RequestMapping("produtos")
11 public class produtoController {
12
13
14     @PostMapping
15     public void salvar(Produto produto){
16         System.out.println("Produto recebido: " + produto.toString());
17     }
18
19
20 }
21
```

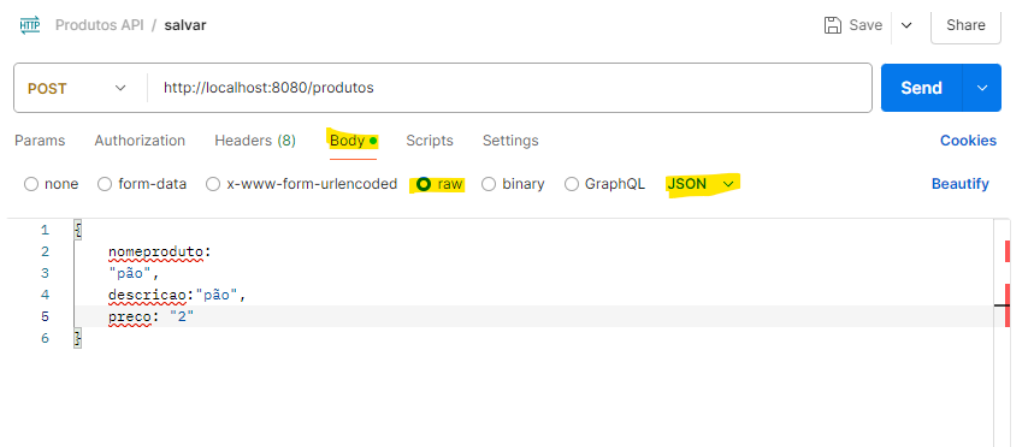
Podemos fazer uma requisição Post de HTML usando o Postman:



Observer conforme o nosso produto que toda vez que recebemos uma requisição soltamos uma mensagem no terminal:



Note que os dados do produto vieram todos nulos. Isso não é por acaso. Com exceção da id do produto, os atributos do vão ser definidos pela gente na requisição http:



Nota: o nome dos atributos devem estar entre aspas:

none form-data x-www-form-urlencoded raw

```
1
2 "nomeProduto":
3 "pão",
4 "descricao": "pão",
5 "preco": 20
6
```

Nós queremos que o texto no body do http seja direcionado ao objeto Produto que nós criamos. Para fazer isso, usamos a anotação `@RequestBody`:

```
public class produtoController {

    @PostMapping
    public void salvar(@RequestBody Produto produto){
        System.out.println("Produto recebido: " + produto.toString());
    }

}
```

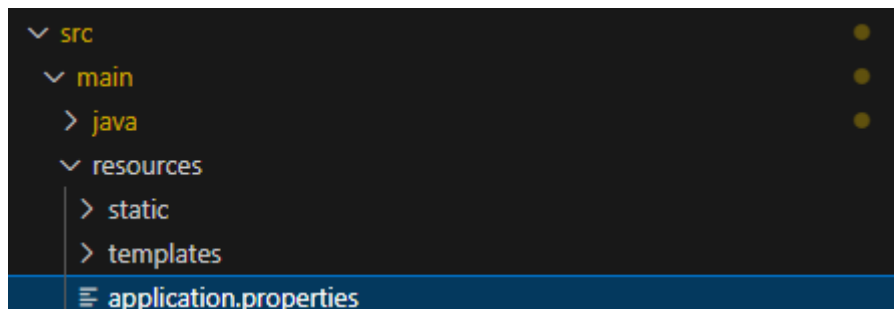
Com isso nós temos agora que o Produto foi criado com os métodos passados no http:

```
Produto recebido: Produto [id=null, nomeProduto=pão, descricao=pão, preco=20.0]
```

BANCO DE DADOS

Configurando o banco de dados: como fazemos isso? Criamos uma nova classe no Java? A resposta é não.

Dentro do nosso projeto temos a pasta Resources, que é usada para guardar todos os nossos recursos externos, dentre eles o nosso banco de dados:



O que nos interessa aqui é o `application.properties`, que é onde vamos configurar o nosso banco de dados. Esse arquivo é responsável pelas propriedades do nosso projeto e pode ter duas extensões: `.properties`, como você pode ver na imagem, e `.yaml`. O segundo formato é o que nós vamos usar porque ele é o mais usado no mercado.

O `yaml` é uma abreviação de `yaml` (iãôl) e é parecida com o `JSON`, sendo usado como um arquivo de configuração:

YAML Ain't Markup Language is a data serialization language that matches user's expectations about data.

Um detalhe aqui: o arquivo `yaml` que vamos criar deve ter nome de `application`.

O arquivo `yaml` que vamos criar vai ter a seguinte estrutura:

```
src > main > resources > ! application.yaml > {} spring > {}  
1  spring:  
2    datasource:  
3      url: jdbc:h2:mem:produtos  
4      driver-class-name: org.h2.Driver  
5      username: sa  
6      password:  
7    h2:  
8      console:  
9        enabled: true  
10     path: /h2-console
```

A primeira parte funciona como uma interface entre a aplicação e o banco de dados e a segunda diz respeito à interface gráfica.

JDBC: Java DataBase Connectivity.

Aqui é interessante trazer a documentação a respeito do `JDBC` no `Spring`:

Why Spring Data JDBC?

The main persistence API for relational databases in the Java world is certainly JPA, which has its own Spring Data module. Why is there another one?

JPA does a lot of things in order to help the developer. Among other things, it tracks changes to entities. It does lazy loading for you. It lets you map a wide array of object constructs to an equally wide array of database designs.

This is great and makes a lot of things really easy. Just take a look at a basic JPA tutorial. But it often gets really confusing as to why JPA does a certain thing. Also, things that are really simple conceptually get rather difficult with JPA.

Spring Data JDBC aims to be much simpler conceptually, by embracing the following design decisions:

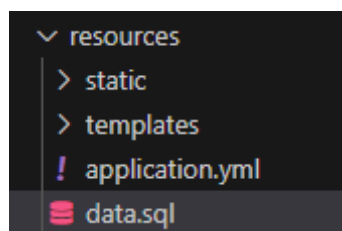
- If you load an entity, SQL statements get run. Once this is done, you have a completely loaded entity. No lazy loading or caching is done.
- If you save an entity, it gets saved. If you do not, it does not. There is no dirty tracking and no session.
- There is a simple model of how to map entities to tables. It probably only works for rather simple cases. If you do not like that, you should code your own strategy. Spring Data JDBC offers only very limited support for customizing the strategy with annotations.

fonte:

<https://docs.spring.io/spring-data/relational/reference/jdbc/why.html>

Assim temos no nosso código que JDBC é a API que estamos usando pra fazer a conexão com o nosso banco e dados e `h2:mem:produtos` indica que o nosso banco de dados é o h2 que tem persistência em memória (mem) e com nome de produtos.

Porque o nosso banco de dados é feito em memória, *on run*, ele acaba caindo toda vez que nós derrubamos a aplicação. Ele não é persistente. Por conta disso nós vamos criar um arquivo dentro do `resources` para guardar os dados. Esse arquivo vai ter o formato `sql`:



Dentro desse arquivo vamos criar os comandos SQL que vão gerar a nossa tabela:

```
create table Produto (
    id varchar(255) not null primary key,
    nome varchar(50) not null,
    descricao varchar(300),
    preco numeric(18,2)
);
```

Com isso, se rodarmos a aplicação e formos verificar o nosso banco de dados, veremos que foi criada uma tabela com o nome produtos:

The screenshot shows a database management interface. On the left, a tree view displays the database structure: 'jdbc:h2:mem:produtos' contains a table 'PRODUTO' with columns 'ID', 'NOME', 'DESCRICAO', and 'PRECO'. Below the table are 'Indexes', 'INFORMATION_SCHEMA', and 'Users'. The version 'H2 2.3.232 (2024-08-11)' is shown at the bottom. On the right, a query editor shows the command 'SELECT * FROM PRODUTO'. Below the editor, the results are displayed: 'SELECT * FROM PRODUTO;' followed by a table with four columns: 'ID', 'NOME', 'DESCRICAO', and 'PRECO'. Below the table, it says '(no rows, 3 ms)'. An 'Edit' button is at the bottom left of the results area.

MAPEANDO A APLICAÇÃO - PERSISTÊNCIA

Resta agora fazer a ligação das requisições HTTP com essa tabela. Para isso, vamos fazer o uso de annotations para fazer esse mapeamento na nossa entidade produto.

Na nossa classe produto, vamos usar a annotation `@Entity` e nos atributos vamos usar a annotation `@Column` pra indicar que cada elemento dali é referência à uma coluna em nosso banco de dados:

```
import jakarta.persistence.Entity;

@Entity
public class Produto {

    @Column
    private String id;
    @Column
    private String nomeProduto;
    @Column
    private String descricao;
    @Column
    private double preco;
}
```

Temos um detalhe: nós não precisamos botar a annotation `@Column` em cima dos atributos porque no nosso banco de dados o nome das colunas são os mesmos do que no atributo. E se o nome dos atributos fossem diferentes dos nomes das colunas no nosso banco de dados? Nesse caso, dentro da annotation nós faríamos essa indicação:

```
@Entity
public class Produto {

    @Column(name = "id")
    private String codigo;
}
```

Para identificar um atributo como id (chave primária), usamos a annotation `@Id`:

```
@Id
@Column
private String id;
```

Apesar de termos feito todas essas configurações, ainda precisamos criar mais uma class: repository.

Aqui vale a pena recapitular o que foi temos:

Temos uma classe `Controller`, que é a nossa `API`.

Temos uma classe `model` chamada de `Produto` que é o nosso `modelo de negócio, os dados de negócio`.

E temos a classe `repository` que representa as classes que farão as `operações no banco de dados` (isso é, as operações SQL, sem que tenhamos que escrevê-las nós mesmos).

O `JpaRepository` **não representa diretamente o banco de dados**, mas sim um **repositório genérico** que interage com ele através do

EntityManager do JPA. Ele estende outras interfaces (`CrudRepository` e `PagingAndSortingRepository`), adicionando métodos para operações CRUD, paginação, ordenação e outras funcionalidades avançadas.

```
1 package com.github.hac1997.primeiroProjeto.repository;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5 public interface ProdutoRepository extends JpaRepository<T, ID> {
6
7 }
8
```

Essa classe, na verdade interface, é uma extensão do `JpaRepository`. O `JpaRepository` deve levar dois parâmetros: a classe que representa a nossa entidade (no caso, `Produto`) e o “tipo” no qual o ID do nosso produto é representado (`String`).

Aqui nós não precisamos fazer nenhuma annotation, porque fizemos um extend da classe `JpaRepository`.

Como estamos trabalhando com o JPA, devemos incluir no `Application.yml` a seguinte configuração:

```
1 spring:
2   datasource:
3     url: jdbc:h2:mem:produtos
4     driver-class-name: org.h2.Driver
5     username: sa
6     password:
7   h2:
8     console:
9       enabled: true
10      path: /h2-console
11   jpa:
12     database-platform: org.hibernate.dialect.H2Dialect
```

Como dito, o `Repository` vai ser responsável pelas operações que serão feitas no banco de dados e por isso nós vamos usar ela dentro do `Controller`:


```

11 @RestController
12 @RequestMapping("produtos")
13 public class ProdutoController {
14
15     private ProdutoRepository produtoRepository;
16
17
18     public ProdutoController(ProdutoRepository produtoRepository) {
19         this.produtoRepository = produtoRepository;
20     }
21

```

A Interface JpaRepository é extremamente útil pra gente porque ela já vem com uma coleção de métodos que usaremos para trabalharmos no nosso banco de dados. Abaixo uma lista dos mais importantes métodos do JpaRepository (segundo o nosso amigo chatGPT):

save(S entity) – Salva ou atualiza uma entidade.
saveAll(Iterable<S> entities) – Salva ou atualiza uma coleção de entidades.
findById(ID id) – Busca uma entidade pelo ID.
existsById(ID id) – Verifica se uma entidade existe pelo ID.
findAll() – Retorna todas as entidades.
findAllById(Iterable<ID> ids) – Retorna todas as entidades correspondentes aos IDs informados.
count() – Retorna a quantidade total de registros.
deleteById(ID id) – Exclui uma entidade pelo ID.
delete(T entity) – Exclui uma entidade específica.
deleteAll(Iterable<? extends T> entities) – Exclui múltiplas entidades.
deleteAll() – Exclui todas as entidades da tabela.

No nosso caso aqui nós vamos usar inicialmente o método **save**.

Dentro do ProdutoController, nós havíamos criado o método chamado salvar:

```

8
9  @RestController
10 @RequestMapping("produtos")
11 public class produtoController {
12
13
14     @PostMapping
15     public void salvar(Produto produto){
16         System.out.println("Produto recebido: " + produto.toString());
17
18     }
19
20 }
21

```

Como instanciamos um `ProdutoRepository`, vamos usar o método `save` dessa classe para salvar a entidade que recebemos no método Post do HTTP:

```

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

    private ProdutoRepository produtoRepository;

    public ProdutoController(ProdutoRepository produtoRepository) {
        this.produtoRepository = produtoRepository;
    }

    @PostMapping
    public void salvar(@RequestBody Produto produto){
        System.out.println("Produto recebido: " + produto.toString());
        produtoRepository.save(produto);
    }

}

```

That's it. Tá feito.

Usando os métodos do `JpaRepository` podemos fazer um **CRUD** (**Create**, **Read**, **Update** e **Delete**).

Vamos criar os seguintes métodos:

ObterPorId

(deve retornar um `Produto` e deve fazer uso do método `Get` do HTTP):

```
@GetMapping("/{id}")
public Produto obterPorId(@PathVariable("id") String id){
    return produtoRepository.findById(id).orElse(other:null);
}
```

O código é bastante simples: usamos a annotation `@GetMapping` e dentro dela definimos um caminho url. Esse caminho terá a forma `"/{id}"`. O `id` entre as chaves é uma abstração no nosso código e basicamente o que estamos fazendo aqui é dizer para o programa interpretar o caminho depois de `"/"` como uma variável que indicamos por `id`. Essa variável é incorporada pelo annotation `@PathVariable`.

DeletePorId

(deve retornar void e é basicamente um copia e cola do código anterior, com exceção de que estaremos usando o método `Delete` do HTTP):

```
@DeleteMapping("/{id}")
public void DeletePorId(@PathVariable("id") String id){
    produtoRepository.deleteById(id);
}
```

Atualizar

(deve retornar void e faremos uso do método `Put` do HTTP)

O código não tem muito segredo. Vamos usar as annotations `@PathVariable` e `@RequestBody` (vamos atualizar um produto, logo, precisamos receber os dados dentro do corpo da mensagem HTTP):

```
@PutMapping("/{id}")
public void Atualizar(@PathVariable("id") String id, @RequestBody Produto produto){
    produto.setId(id);
    produtoRepository.save(produto);
}
```

O Código é bem simples: setamos o `Id` do produto como o `Id` indicado pelo `{id}` e salvamos o produto no `produtoRepository`. Se não houver produto com o `id` indicado, é setado de qualquer maneira o `id` para o indicado no path `{id}`.

buscar

Esse vai ser o método mais complexo dos que iremos trabalhar. O objetivo é buscarmos os produtos no banco de dados com base nos

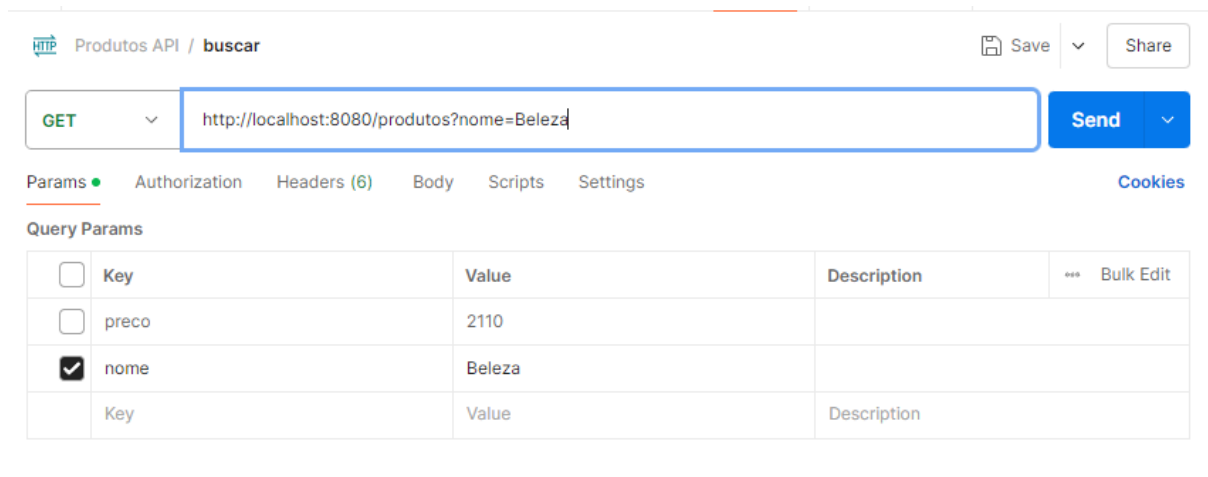
parâmetros como nome, valor e descrição. Nessa parte vamos ver um pouco da mágica do JpaRepository com a criação dos métodos que usaremos para filtrar.

O código é o seguinte:

```
@GetMapping
public List<Produto> buscar(@RequestParam("nome") String nome){
    return produtoRepository.findByNome(nome);
}
```

Usamos as annotations @GetMapping e @RequestParam.

A annotation @RequestParam diz respeito ao Query Parameter que estamos usando no nosso método Get do HTTP:



Produtos API / buscar

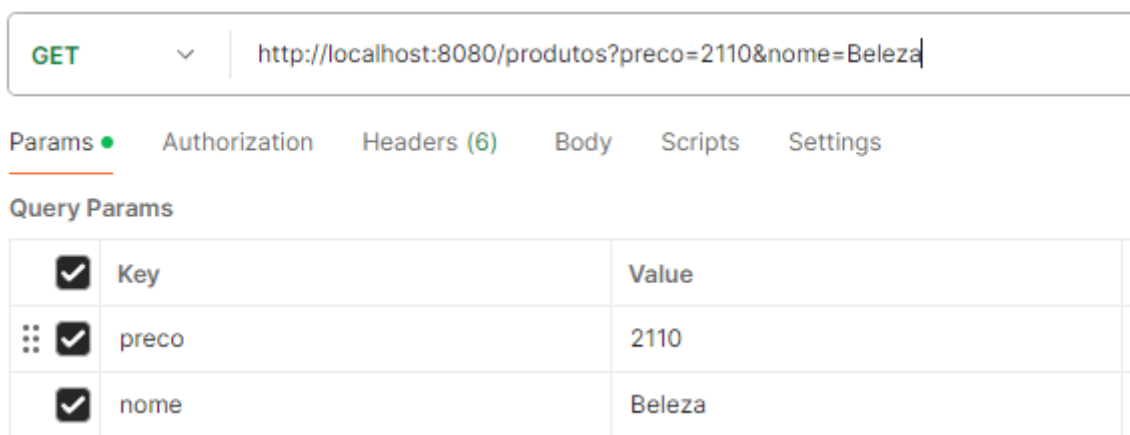
GET Send

Params • Authorization Headers (6) Body Scripts Settings Cookies

Query Params

<input type="checkbox"/>	Key	Value	Description	...	Bulk Edit
<input type="checkbox"/>	preco	2110			
<input checked="" type="checkbox"/>	nome	Beleza			
	Key	Value	Description		

Segue daí que poderíamos ter mais de um filtro no nosso código. O detalhe é que ele seria um filtro do tipo lógico AND:



GET

Params • Authorization Headers (6) Body Scripts Settings

Query Params

<input checked="" type="checkbox"/>	Key	Value
<input checked="" type="checkbox"/>	preco	2110
<input checked="" type="checkbox"/>	nome	Beleza

O nosso código ficaria assim se quiséssemos fazer a sua implementação:

```
@GetMapping
public List<Produto> buscar(@RequestParam("nome") String nome, @RequestParam("preco") double preco){
    List<Produto> produtos = new ArrayList<>();
    produtos.addAll(produtorepository.findByNome(nome));
    produtos.addAll(produtorepository.findByPreco(preco));
    return produtos;
}
```

Note que o tipo de retorno é List porque estamos retornando uma lista de Produtos.