Api Rest

O que é Rest Afinal?

Trata-se de um meio de realizar a comunicação entre dois sistemas diferentes (independe as linguagens).

Rest no Spring:

Depois de estruturar o projeto de forma correta no Spring, devemos criar uma classe de controle eu ira receber as propriedades de uma Resta api.

Para que uma classe seja identificada pelo Spring como um serviço REST, é necessário eu ela esteja anotada com **@Service** e **@RestController**.

Métodos HTTP:

O protocolo HTTP possui nove métodos HTTP (7 podem ser usados pelo REST).

GET – para recuperação de dados.

POST – para criação de dados.

PUT e PATCH – para atualização de dados.

DELETE – para apagar os dados.

GET:

Para usarmos o get, utilizamos a anotação **@GetMapping**, depois temos que escolher o tipo de retorno por esta API, os mais famosos são **XML** e **JSON**.

Para informarmos que o retorno será um **JSON**, utilizamos a anotação **@RequestMapping(produces = MediaType.APPLICATION_JSON_VALUE)**, colocando essa anotação na classe informamos que o tipo de mídia que a variável produces será uma aplicação com um valor em JSON.

Devemos também mapear as requisições através de urls: *@GetMapping*("/drivers").

Podemos também utilizar um get através de uma variável especifica, exemplo com id:

@GetMapping(“/drivers/{id}”)

Public Driver findDriver(@PathVariable(“id”) Long id){

return driverRepository.findById(id).get();

}

A anotação @**PathVariable** vincula o texto “id” ao parâmetro Long id.

Post:

Este método é especial, pois permite eu o usuário utilize o corpo da requisição para passar parâmetros, ao contrário do GET, que só permite eu o servidor envie dados no corpo.

Usamos a anotação @**PostMapping** e o acrescentamos uma URL.

Podemos usar a mesma URL das outras requisições, pois em REST a URL de um recurso é apenas um dos aspectos da forma como interagimos com ele.

Devemos também utilizar a anotação @**RequestBody** para indicar que o parâmetro que será passado, faz parte de um objeto.

Codigo:

@PostMapping(“/drivers”)

public Driver createDriver(@RequestBody Driver driver){

return driverRepository.save(driver);

}

PUT e PATCH:

O PUT assim como o PATCH tratam-se de de formas de atualização de dados com idempotência, no PUT todos os atributos da coluna são atualizados, já o PATCH apenas um atributo da coluna será atualizado. Vale lembrar que o único atributo que não será atualizado em nenhum dos dois é a primary key.

Implementação:

Para implementar o método vamos chamar o método finddriver já usado no GET, para que caso o não exista uma coluna com esse id ele ira retornar o erro 404, vamos também atualizar os dados do objeto encontrado através do foundDriver para os padrões passados, e por fim usar o método save para persistir:

@PutMapping(“/drivers/{id}”)

public Driver fullUpdateDriver(@PathVariable(“id”) Long id, @RequestBody Driver driver){

Driver foundDriver = findDriver(id);

foundDriver.setBirthDate(driver.getBirthDate());

foundDriver.setName(driver.getName());

return driverRepository.save(foundDriver);

}

DELETE:

Existe uma polemica em relação ao DELETE se é permitido ou não enviar dados no corpo, o que pode gerar problemas de acordo com a ferramenta utilizada.

Codigo:

@DeleteMapping(“/drivers/{id}”)

public void deleteDriver(@PathVariable(“id”) Long id){

driverRepository.deleteById(id);

}

Erros HTTP:

O protocolo HTTP define alguns tipos de códigos de status, cada um condicionado a uma hierarquia diferente:

. (1xx) – São informacionais, mostram ao cliente eu a requisição foi recebida e que algo está sendo executado.

. (2xx) -São códigos de sucesso, mostram ao cliente que a requisição finalizou seu processamento com sucesso.

. (3xx) – São códigos de redirecionamento, mostram ao cliente que o resultado final da requisição depende de outras ações.

. (4xx) – São códigos de falha que foi originada por algo de errado que o cliente fez.

. (5xx) – São códigos de falha que foi originada por uma condição não tratada pelo servidor.

Modificando página de erro:

Ao buscarmos por um id não existente no nosso método de listar, o erro recebido será o 500, porem o erro ideal nesse caso seria o erro 404 Not Found, para mudarmos isso devemos utilizar a propriedade **orElseThrow** do spring, trata-se de uma exception do spring que possui os erros HTTP.

Codigo:

```
@GetMapping(“/drivers/{id}”)

Public Driver findDriver(@PathVariable(“id”) Long id){

return driverRepository.findById(id).orElseThrow(()-> new
ResponseStatusException(HttpStatus.NOT_found));

}
```

Invocações Sucessivas:

Existe uma propriedade HTTP que é a **idempotência**, uma requisição é idempotente quando o efeito da enésima requisição é igual ao da primeira, ou seja, o estado não muda em invocações repetitivas.

O método GET é idempotente e o POST não, por isso quando utilizamos os browsers e tentamos reenviar um POST o próprio browser nos pergunta se queremos mesmo enviar.

Subconjuntos de URL:

Um recurso pode ter subcoleções de recursos, Exemplo:

“/Solicitações/{Solicitações ID}/Viagens/{Viagens ID}”

Ou seja, podemos interpretar as URLs como conjuntos, e cada um dos trechos seguintes (separados por /) pode ser interpretado como subconjunto do conjunto principal.

HATEOAS:

HATEOAS é uma abreviação para **Hypermedia As The Engine Of Application State**, ou hipermídia como o motor de estado da aplicação. Este é

um nome grande para indicar a noção de que REST deve utilizar links para relacionar recursos conectados entre si.

Em REST utilizamos esses conceitos para demonstrar a relação entre recursos que estão conectados entre si, mas não necessariamente pertencem um ao outro, quando queremos demonstrar essa relação, utilizamos links.

O Spring facilita a criação desses links com o subprojeto Spring HATEOAS. Temos que adicionar a dependência:

```
<!-- https://mvnrepository.com/artifact/org.springframework.hateoas/spring-hateoas -->
<dependency>
    <groupId>org.springframework.hateoas</groupId>
    <artifactId>spring-hateoas</artifactId>
    <version>1.12.11</version>
</dependency>
```

O Spring HATEOAS nos fornece uma hierarquia de classes para lidar com recursos de vários tipos:

. Para trabalhar com links de uma **única entidade**, a melhor classe a ser utilizada é a **org.springframework.hateoas.EntityModel**;

. Para trabalhar com links sobre uma **coleção de entidades**, a melhor classe a ser utilizada é a **org.springframework.hateoas.CollectionModel**;

. Para trabalhar com links sobre uma **coleção com suporte a paginação**, a melhor classe a ser utilizada é a **org.springframework.hateoas.PagedModel**;

. Caso nenhuma das classes acima atenda às necessidades do projeto, é possível criar uma extensão da classe **org.springframework.hateoas.RepresentationModel**;

Para adicionarmos o atributo links em uma classe devemos utilizar o **método EDGE**, esse método recebe como parâmetro um objeto link e esse objeto link que tem os dois atributos **rel** e **href**.

Para aplicarmos o HATEOAS em uma classe devemos extender uma das classes do HATEOAS.

Exemplo:

```
public class ProdutoModel extends RepresentationModel<ProdutoModel>{

//variáveis e métodos

}
```

Depois de termos a classe extendendo classes HATEOAS devemos aplicar o HATEOAS na pratica através dos métodos de requisição.

Exemplo de aplicação do método:

```
 public ResponseEntity<List<ProdutoModel>> getAllProdutos(){

List<ProdutoModel> produtoList = produtoRepository.findAll();

If(produtosList.isEmpty()){

        return new ResponseEntity<>(HttpStatus.NOT_FOUND);

}else{

        for(ProdutoModel produto : produtoList){

                long id = produto.getIdProduto();

                //aqui ira o HATEOAS


        produto.add(linkTo(methodOn(ProdutoController.class).getOneProduto(i
d)).withSelfRel());

        //O método add pertence as classes do HATEOAS eu foram extendidas

        //Devemos criar e passar o link para o método add

        //Para gerar o link no add usamos o linkTo

        //Para utilizarmos o linkTo devemos usar o methodOn que ira mapear o
link
```

```
}
        return new ResponseEntity<List<ProdutoModel>>(produtosList,
HttpStatus.OK);
}
}
```

Testes Automatizados

Estratégias:

Testes unitários: Os Testes Unitários restringem-se a um único componente, e suas dependências são fornecidas como mocks, ou seja, retornam respostas pré-programadas.

Testes de Integração: Os testes de integração abrangem escopos mais amplos, atingindo vários componentes, algumas vezes também requerendo mocks, mas procurando depender o mínimo possível.

Testes de contrato: Os testes de contratos são como os testes de integração, mas são feitos inclusive testando a interface da API.

O spring não possui suporte a esses testes, vamos usar a framework REST Assured. Ele atua em conjunto com o spring para fornecer testes das APIs muito poderosos.

Criando testes de API com REST Assured:

Como sempre vamos adicionar a dependência do REST Assured no pom.xml. Devemos utilizar também o framework JUnit(que é o padrão de fato para testes em Java).

Devemos criar uma classe com o seguinte nome: nomedaclassetestadaTestIT, esse padrão é o padrão maven para testes de integração.

A tarefa dessa classe é inicializar todo o contexto spring, assim como o web server associado. Para isso só precisamos usar a anotação **@SpringBootTest**.

Devemos entender que o teste deve ser realizado em uma porta especifica visto que teremos diversos testes ocorrendo em paralelo. Isso pode ser feito utilizando o parâmetro **webEnvironment** da anotação @SpringBootTest, passando como parâmetro a constante RANDOM_PORT:

```
@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
```