01 Requisições PUT

Transcrição

Agora vamos apresentar a funcionalidade de atualização.

Quando editarmos os perfis dos médicos, nossa *API* receberá uma nova requisição com os novos dados. Precisamos, porém, saber qual médico precisa ter seus dados atualizados no banco de dados.

A forma que temos para identificar os registros é o *ID*. O problema é que, por enquanto, o *ID* não está sendo devolvido na funcionalidade de listagem. Por isso, vamos alterar o *DTO* de listagem, para que ele inclua o *ID*.

Para isso, voltaremos à *IDE*. Vamos até a funcionalidade de listagem do arquivo "MedicoController.java". Dentro dele, abriremos o *DTO* "DadosListagemMedico.java". Dentro da `public record`, vamos adicionar `Long id` dentro dos parênteses.

Para que não haja erro de compilação no construtor, vamos passar o *ID* no `this`, passando `medico.getId()` entre parênteses:

```
package med.voll.api.medico;


public record DadosListagemMedico(Long id, String nome, String email,
String crm, Especialidade especialidade) {


    public DadosListagemMedico(Medico medico) {
```

```
        this(medico.getId(), medico.getNome(), medico.getEmail(),
medico.getCrm(), medico.getEspecialidade());
    }

} COPIAR CÓDIGO
```

Agora basta salvar. Se voltarmos ao *Insomnia*, receberemos a *ID* ao disparar a requisição. Ainda no *Insomnia*, vamos simular a nova requisição, de atualização.

Faremos isso clicando no ícone de "+ > HTTP Request". Passaremos a *URL* `http://localhost:8080/medicos`. Dessa vez, porém, a requisição será "PUT", a mais comum para atualizações.

Obs: Como o verbo é diferente do que já usamos, podemos continuar a utilizar a mesma *URL*.

No campo "Body", vamos selecionar a opção "JSON". Depois, vamos inserir os dados do médico. No vídeo, o instrutor insere o *ID* e telefone.

Quando ele dispara a requisição, recebemos como retorno o erro "405 Method Not Allowed". Isso acontece porque disparamos uma requisição para o endereço "/medicos", que até existe na nossa *API*, mas ainda não vai mapeado para requisições "PUT".

Para isso, vamos implementar um novo método no controller para atender a esse tipo de requisições.

De volta a "MedicoController.java" na *IDE*, vamos adicionar o método `public_void atualizar()`. Acima do método, passaremos as anotações `@PutMapping` e `@Transactional`.

Como método, já que precisamos receber o objeto *DTO*, passaremos `@RequestBody @Valid DadosCadastroMedico dados)`:

```
    @PutMapping
    @Transactional
    public void atualizar(@RequestBody @Valid DadosCadastroMedico
dados) {


    } COPIAR CÓDIGO
```

Porém, não poderemos usar `DadosCadastroMedico`. Precisaremos criar outro *DTO*. É isso que faremos no próximo vídeo.

02 Atualizando dados

Transcrição

Na atualização não poderemos utilizar o mesmo *DTO* do cadastro, por se tratarem de campos diferentes e com validações distintas.

Ao invés de `DadosCadastroMedico`, usaremos `DadosAtualizacaoMedico`. Com o atalho "Alt + Enter > Create record 'DadosAtualizacaoMedico'", vamos substituir o pacote para "med.voll.api.medico" e clicar no OK.

Receberemos apenas o *ID* e os dados atualizáveis, nome, telefone e endereço. Passaremos todas como strings, exceto o endereço, para o qual usaremos o *DTO* `DadosEndereco endereco`. As validações dela, como logradouro e CEP, permanecerão. Reaproveitamos o *DTO*.

No campo *ID*, adicionaremos a anotação `@NotNull`, para informar que ele é obrigatório:

```
package med.voll.api.medico;


import jakarta.validation.constraints.NotNull;
import med.voll.api.endereco.DadosEndereco;


public record DadosAtualizacaoMedico(
        @NotNull
        Long id,
        String nome,
        String telefone,
        DadosEndereco endereco) {
} COPIAR CÓDIGO
```

De volta a "MedicoController.java", vamos carregar o objeto no banco de dados, criando a variável `medico = repository.getReferenceById(dados.id())`.

Logo abaixo, vamos atualizar os dados criando o método `medico.atualizarInformacoes(dados)`. Vamos registrar o novo método clicando em "Alt + Enter > Create method". Seremos

redirecionados para a classe "Medico.java". Lá, será criada a assinatura. Vamos adicionar a implementação dentro das chaves.

Passaremos `this.nome = dados.nome()`. Pegaremos o nome do médico atual e substitui-lo pelo nome que chega via *DTO*. Como queremos atualizar apenas se o campo for enviado, precisaremos adicionar um `if`.
Como parâmetros, passaremos `(dados.nome() != null)`. Faremos a mesma coisa com os outros campos. No endereço, porém, criaremos o método `atualizarInformacoes()` na classe `Endereco,`, passando dentro do `if` e selecionando "Alt + Enter > Create method":

```
}


  public void atualizarInformacoes(DadosAtualizacaoMedico dados) {
      if (dados.nome() != null) {
          this.nome = dados.nome();
      }
      if (dados.telefone() != null) {
          this.telefone = dados.telefone();
      }
      if (dados.endereco() != null) {
          this.endereco.atualizarInformacoes(dados.endereco());
      }

  } COPIAR CÓDIGO
```

Quando formos redirecionados ao método `Endereco.java`, vamos atualizar os dados. Criaremos um `if` com o

parâmetro `dados.logradouro() != null`. Faremos o mesmo para os outros campos:

```
 public void atualizarInformacoes(DadosEndereco dados) {
        if (dados.logradouro() != null) {
            this.logradouro = dados.logradouro();
        }
        if (dados.bairro() != null) {
            this.bairro = dados.bairro();
        }
        if (dados.cep() != null) {
            this.cep = dados.cep();
        }
        if (dados.uf() != null) {
            this.uf = dados.uf();
        }
        if (dados.cidade() != null) {
            this.cidade = dados.cidade();
        }
        if (dados.numero() != null) {
            this.numero = dados.numero();
        }
        if (dados.complemento() != null) {
            this.complemento = dados.complemento();
        }
    }
} COPIAR CÓDIGO
```

Agora vamos voltar para "MedicoController.java". Não precisamos fazer a atualização no banco de dados, isso é automático da *JPA*.

De volta ao *Insomnia*, podemos testar a atualização chamando a listagem de médicos. Removeremos os parâmetros de paginação para que todos sejam carregados.

Poderemos fazer as atualizações usando a requisição "PUT". Basta passar a informação do campo que queremos atualizar. No vídeo, o instrutor altera o nome do médico "João Carlos" para "João Carlos da Silva", informando os campos de *ID* e nome, assim:

```
        "id": 2,
        "nome": "João Carlos da Silva" COPIAR CÓDIGO
```

Se dispararmos a requisição de listagem de médicos depois disso, veremos que tivemos sucesso. Nossa requisição funcionou corretamente.

No próximo vídeo, implementaremos a requisição de exclusão.

03 Para saber mais: Mass Assignment Attack

Mass Assignment Attack ou Ataque de Atribuição em Massa, em português, ocorre quando um usuário é capaz de inicializar ou substituir parâmetros que não deveriam ser modificados na aplicação. Ao incluir parâmetros adicionais em uma requisição, sendo tais parâmetros válidos, um usuário mal-intencionado pode gerar um efeito colateral indesejado na aplicação.

O conceito desse ataque refere-se a quando você injeta um conjunto de valores diretamente em um objeto, daí o nome atribuição em massa, que sem a devida validação pode causar sérios problemas.

Vamos a um exemplo prático. Suponha que você tem o seguinte método, em uma classe Controller, utilizado para cadastrar um usuário na aplicação:

```
@PostMapping
@Transactional
public void cadastrar(@RequestBody @Valid Usuario usuario) {
    repository.save(usuario);
} COPIAR CÓDIGO
```

E a entidade JPA que representa o usuário:

```
@Getter
@Setter
@NoArgsConstructor
@EqualsAndHashCode(of = "id")
@Entity(name = "Usuario")
@Table(name = "usuarios")
public class Usuario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String email;
    private Boolean admin = false;
```

```
    //restante do código omitido…
} COPIAR CÓDIGO
```

Repare que o atributo `admin` da classe `Usuario` é inicializado como `false`, indicando que um usuário deve sempre ser cadastrado como não sendo um administrador. Porém, se na requisição for enviado o seguinte JSON:

```
{
    "nome" : "Rodrigo",
    "email" : "rodrigo@email.com",
    "admin" : true
} COPIAR CÓDIGO
```

O usuário será cadastrado com o atributo `admin` preenchido como `true`. Isso acontece porque o atributo `admin` enviado no JSON existe na classe que está sendo recebida no Controller, sendo considerado então um atributo válido e que será preenchido no objeto `Usuario` que será instanciado pelo Spring.

Então, como fazemos para prevenir esse problema?

Prevenção

O uso do padrão DTO nos ajuda a evitar esse problema, pois ao criar um DTO definimos nele apenas os campos que podem ser recebidos na API, e no exemplo anterior o DTO não teria o atributo `admin`.

Novamente, vemos mais uma vantagem de se utilizar o padrão DTO para representar os dados que chegam e saem da API.

04 Para saber mais: PUT ou PATCH?

Escolher entre o método HTTP PUT ou PATCH é uma dúvida comum que surge quando estamos desenvolvendo APIs e precisamos criar um endpoint para atualização de recursos. Vamos entender as diferenças entre as duas opções e quando utilizar cada uma.

PUT

O método PUT substitui todos os atuais dados de um recurso pelos dados passados na requisição, ou seja, estamos falando de uma atualização integral. Então, com ele, fazemos a atualização total de um recurso em apenas uma requisição.

PATCH

O método PATCH, por sua vez, aplica modificações **parciais** em um recurso. Logo, é possível modificar apenas uma parte de um recurso. Com o PATCH, então, realizamos atualizações parciais, o que torna as opções de atualização mais flexíveis.

Qual escolher?

Na prática, é difícil saber qual método utilizar, pois nem sempre saberemos se um recurso será atualizado parcialmente ou totalmente

em uma requisição - a não ser que realizemos uma verificação quanto a isso, algo que não é recomendado.

O mais comum então nas aplicações é utilizar o método PUT para requisições de atualização de recursos em uma API, sendo essa a nossa escolha no projeto utilizado ao longo deste curso.

05 Requisições DELETE

Transcrição

Vamos criar a funcionalidade de exclusão.

De acordo com a definição, a exclusão não deve apagar os dados do médico, mas deixá-lo como inativo no sistema. Esse é o conceito de exclusão lógica.

Nesse vídeo, porém, faremos a exclusão tradicional.

Vamos acessar o Insomnia e clicar em "+ > HTTP Request". Vamos renomear a nova requisição para "Excluir Médico". Vamos selecionar o verbo "DELETE". A *URL* continuará a mesma.

No corpo, passaremos a informação do *ID*, para identificar qual médico será deletado do banco de dados.

Selecionaremos a opção "JSON" no campo "Body". Vamos passar um parâmetro dinâmico na barra de endereços. Vamos adicionar uma

barra e o *ID* do médico à *URL*, depois de `/medicos`. No vídeo, o instrutor passa a *URL* abaixo:

```
http://localhost:8080/medicos/3 COPIAR CÓDIGO
```

Se enviarmos a requisição, receberemos o erro "404 Not Found", porque a *URL* `/medicos/3` não está mapeada. Vamos resolver isso em "MedicoController.java", criando o novo método, para exclusão.

O método se chamará `excluir` e, acima dela, a anotação será `@DeleteMapping`. Para levar o *ID*, Vamos abrir parênteses e aspas após e anotação e passar o complemento da *URL*. Para que seja um parâmetro dinâmico, passaremos `("/{id}")`.

Também passaremos isso como parâmetro do método `excluir`. Faremos isso passando `Long id` como parâmetro e informando ao *Spring* que ele se trata do `("{/id}")`. Para isso, adicionaremos `@PathVariable` ao parâmetro do método.

Também adicionaremos o método `@Transactional` abaixo de `@DeleteMapping`. Com a ajuda de `repository`, também passaremos `.deleById(id)` para fazer o delete no banco de dados.

Vamos tentar excluir o médico de ID 3, disparando a requisição com a *URL* `http://localhost:8080/medicos/3`. Dará certo.

Fizemos a exclusão tradicional. No próximo vídeo, aprenderemos a fazer a exclusão lógica.

06 Exclusão lógica

Transcrição

Vamos fazer uma exclusão lógica. Em outras palavras, não vamos apagar o médico do banco de dados, mas marcá-lo como inativo.

Vamos criar uma nova *migration*. Antes disso, como de costume, vamos parar o projeto.

Vamos acessar "src > main > resources > db.migration". Nessa pasta, vamos criar a *migration* "V3__alter-table-medicos-add-column-ativo.sql". Agora vamos abrir a *migration*.

Dentro dela, passaremos o código abaixo:

```
alter table medicos add ativo tinyint;

update medicos set ativo =1;

COPIAR CÓDIGO
```

Essa nova migration será responsável por alterar a tabela de médicos, adicionando a coluna chamada "ativo", do tipo `tinyint`.

Agora vamos voltar a rodar o projeto. É hora de atualizar a entidade *JPA*. Vamos declarar o atribuito `private Boolean ativo`.

Atualizaremos, também, o construtor que recebe `DadosCadastroMedico`. Nele, adicionaremos `this.ativo = true;`:

```
    private Boolean ativo;


    public Medico(DadosCadastroMedico dados) {
        this.ativo = true;
        this.nome = dados.nome();
        this.email = dados.email();
        this.telefone = dados.telefone();
        this.crm = dados.crm();
        this.especialidade = dados.especialidade();
        this.endereco = new Endereco(dados.endereco());
    }
```

COPIAR CÓDIGO

Vamos voltar à funcionalidade de excluir de "MedicoController.java". Nela, vamos carregar e entidade, inativá-lo, configurar o atributo como "false" e disparar o update no banco de dados. Vamos substituir `depository.deleteById(id);` por `var medico :Medico = repository.getReferenceById(id);`.

Vamos configurar o atributo como inativo, chamando `medico.excluir()`:

```
    @DeleteMapping("/{id}")
    @Transactional
    public void excluir(@PathVariable Long id) {
        var medico = repository.getReferenceById(id);
        medico.excluir();
    }
```

COPIAR CÓDIGO

Com a ajuda de "Alt + Enter > Create method", criaremos o método. Entre as chaves do método, em "Medico.java", passaremos `this.ativo = false`:

```
    }



    public void excluir() {
        this.ativo = false;
    }
}
```

COPIAR CÓDIGO

Agora vamos salvar e testar no *Insomnia*. Se dispararmos a requisição de exclusão para inativar o médico de *ID* 2, passando a *URL* "http://localhost:8080/medicos/2", a requisição será processada.

A listagem, porém, continua apresentando médicos ativos e inativos. Vamos alterar isso, para exibir apenas os ativos. Vamos voltar ao método `lista`, em "MedicoController.java", para atualizar a listagem.

Nesse método, vamos substituir `findAll` por `finAllByAtivoTrue`:

```
    public Page<DadosListagemMedico> listar(@PageableDefault(size =
10, sort = {"nome"}) Pageable paginacao) {
        return
repository.findAllByAtivoTrue(paginacao).map(DadosListagemMedico::new)
;
    }
```

```
COPIAR CÓDIGO
```

Com a ajuda do "Alt + Enter > Create method", vamos criá-lo no nosso *repository*. Nele, passaremos o código abaixo:

```
package med.voll.api.medico;


import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.data.jpa.repository.JpaRepository;


public interface MedicoRepository extends JpaRepository<Medico, Long>

{

    Page<Medico> findAllByAtivoTrue(Pageable paginacao);

}

COPIAR CÓDIGO
```

Se salvarmos e dispararmos a requisição novamente, veremos que tivemos sucesso.

07 Parâmetros dinâmicos

Você está trabalhando em uma aplicação com Spring Boot e se depara com o seguinte método:

```
@DeleteMapping("/id")
public void apagar(@PathVariable Long id) {
    repository.deleteById(id);
} COPIAR CÓDIGO
```

Considerando que a classe Controller a qual esse método pertence está anotada com `@RequestMapping("/produtos")`, o que acontecerá

se uma requisição DELETE for disparada para a API com a url **/produtos/0**?

- O produto não será apagado do banco de dados, pois faltou anotar o método com `@Transactional`.

  Mesmo faltando a anotação `@Transactional` no método, esse não é o problema que vai ocorrer.

- Alternativa correta

  Ocorrerá um erro por conta do id ter sido passado com o valor 0 (zero), sendo que no banco de dados os ids devem começar com 1.

- Alternativa correta

  Ocorrerá um erro *404 - not found*.

  O parâmetro dinâmico `id`, adicionado na anotação `@DeleteMapping`, foi declarado sem estar entre chaves({}). Com isso, o Spring vai considerar que a URL para chamar esse método deve ser **/produtos/id**, ou seja, ele vai considerar que a palavra **id** faz parte da URL, e não que se trata de um parâmetro dinâmico.

Parabéns, você acertou!

08 Faça como eu fiz: atualização e exclusão

Agora é com você! Faça o mesmo procedimento que eu fiz na aula, porém, para as funcionalidades de **atualização e exclusão de pacientes**.

Opinião do instrutor

-
-

Você precisará adicionar novos métodos no Controller de paciente:

```
@PutMapping

@Transactional

public void atualizar(@RequestBody @Valid DadosAtualizacaoPaciente

dados) {
```

```
    var paciente = repository.getReferenceById(dados.id());
    paciente.atualizarInformacoes(dados);
}

@DeleteMapping("/{id}")
@Transactional
public void remover(@PathVariable Long id) {
    var paciente = repository.getReferenceById(id);
    paciente.inativar();
} COPIAR CÓDIGO
```

Também precisará criar um atributo e novos métodos na entidade `Paciente`, além de modificar o construtor dela:

```
private Boolean ativo;

public Paciente(DadosCadastroPaciente dados) {
    this.ativo = true;
    this.nome = dados.nome();
    this.email = dados.email();
    this.telefone = dados.telefone();
    this.cpf = dados.cpf();
    this.endereco = new Endereco(dados.endereco());
}

public void atualizarInformacoes(DadosAtualizacaoPaciente dados) {
    if (dados.nome() != null)
        this.nome = dados.nome();
```

```
    if (dados.telefone() != null)
        this.telefone = dados.telefone();

    if (dados.endereco() != null)
        endereco.atualizarInformacoes(dados.endereco());
}

public void inativar() {
    this.ativo = false;
} COPIAR CÓDIGO
```

Na sequência, será necessário criar o DTO `DadosAtualizacaoPaciente` e modificar o `DadosListagemPaciente`:

```
public record DadosAtualizacaoPaciente(
    Long id,
    String nome,
    String telefone,
    @Valid DadosEndereco endereco
) {
} COPIAR CÓDIGO
public record DadosListagemPaciente(Long id, String nome, String
email, String cpf) {
    public DadosListagemPaciente(Paciente paciente) {
        this(paciente.getId(), paciente.getNome(),
paciente.getEmail(), paciente.getCpf());
    }
} COPIAR CÓDIGO
```

Vai precisar também criar uma migration (**Atenção!** Lembre-se de parar o projeto antes de criar a migration!):

```
alter table pacientes add column ativo tinyint;

update pacientes set ativo = 1;

alter table pacientes modify ativo tinyint not null; COPIAR CÓDIGO
```

E, por fim, vai precisar atualizar o método da listagem na classe `PacienteController`, para trazer somente os pacientes ativos, além de também criar o novo método na interface `PacienteRepository`:

```
@GetMapping

public Page<DadosListagemPaciente> listar(@PageableDefault(page = 0,

size = 10, sort = { "nome" }) Pageable paginacao) {

    return

repository.findAllByAtivoTrue(paginacao).map(DadosListagemPaciente::ne

w);

} COPIAR CÓDIGO

Page<Paciente> findAllByAtivoTrue(Pageable paginacao); COPIAR CÓDIGO
```

09 O que aprendemos?

Nessa aula, você aprendeu como:

- Mapear requisições PUT com a anotação `@PutMapping`;
- Escrever um código para atualizar informações de um registro no banco de dados;
- Mapear requisições DELETE com a anotação `@DeleteMapping`;
- Mapear parâmetros dinâmicos em URL com a anotação `@PathVariable`;
- Implementar o conceito de exclusão lógica com o uso de um atributo booleano.

10 Projeto final do curso

Caso queira, você pode baixar aqui o projeto completo implementado neste curso.

11 Conclusão

Transcrição

Parabéns, você chegou ao final desse treinamento de *Spring Boot*!

Criamos um *API REST* utilizando *Spring Boot* e também desenvolvemos um *CRUD*.

Aprendemos o que é o *Spring Boot* e quais suas principais diferenças quando comparado ao *Spring* tradicional. Usamos o *Spring Initializr* para fazer a construção do nosso projeto.

Entendemos como funciona a estrutura de diretórios e como funciona o arquivo "pom.xml". Aprendemos a utilizar alguns de seus módulos, como Web, Validação e *Spring Data JPA*.

Usamos outras bibliotecas, como o driver do *MySQL, o Flyway e o Lombok*. Também aprendemos como funcionam as configurações em um projeto com *Spring Boot*.

Aprendemos a fazer migrations para ter controle do histórico de evolução do banco de dados e a implementar a *API Rest* a partir da criação dos controllers. Aprendemos a usar repositories, para simplificar o acesso ao banco de dados e ao *JPA*.

Fizemos o mapeamento das entidades *JPA*. Implementamos, também, o *CRUD* com validações.

Até a próxima!