**Spring Boot API REST: Construa uma API**

Anotações Controller:

@ResponseBody

Por padrão, o Spring considera que o retorno do método é o nome da página que ele deve carregar, mas ao utilizar a anotação @ResponseBody, indicamos que o retorno do método deve ser *serializado* e devolvido no corpo da resposta.

DTO

Não é uma boa prática devolver entidades da JPA no seu controller, porque na entidade da JPA - na classe de domínio - você tem um monte de atributos (título, mensagem, data de criação e outros). E você tem atributos que são outras entidades, outras classes que têm outros atributos, sendo que um deles pode ser outra classe. O Jackson, por padrão, "serializa" todos os atributos que estiverem dentro da classe. E nem sempre, eu vou querer devolver no meu JSON todos os atributos da minha classe. Posso querer devolver só dois, três atributos ou ter uma flexibilidade. Por exemplo, em um endpoint devolver x atributos, e, em outro, y.

Se você sempre devolve a classe de domínio - a entidade da JPA - ele sempre vai devolver todos os atributos, você não tem flexibilidade. Por isso não é uma boa prática.

Ao invés de devolvermos uma lista de tópicos, List<Topico>, vamos criar outra classe que representa só os dados que quero devolver nesse endpoint. E essa classe é só uma classe de valor, que só tem aqueles atributos que quero devolver. Geralmente o pessoal utiliza o padrão "*DTO*", Data Transfer Object, que muita gente também chama de "*VO*", Value Object, para esse tipo de classe.

Dentro do código DTO, vou declarar somente os atributos, usando as classes primitivas do Java. Só vai ter "int", "long", ou "string" e "data". Não vai ter nenhuma classe de domínio aqui. Vai ser uma classe só com atributos primitivos.

Neste DTO, quais são os campos do tópico que quero devolver? Para esse exemplo, vou devolver apenas o "id" (private Long id;), o "título" (private String titulo;), a "mensagem" (private String mensagem;) e a "data de criação" (private LocalDateTime dataCriacao; e importar o localDateTime).

Nessa classe, vou criar um construtor, TopicoDto. Toda vez que eu der new no topicoDto, passo como parâmetro um objeto do tipo topico. Isto é, TopicoDto(Topico topico) Dentro do Topico, já tenho as informações: o "id", o "título" e a "mensagem". Recebo o tópico e, a partir desse tópico, preencho os atributos.

Eu vou preencher de onde o "id" vem, por exemplo e vai ficar topico, que foi passado como parâmetro construtor, mais getId(), ou seja, this.id = topico.getId();. A mesma coisa com os outros atributos.

 É isso. Quando eu der new nessa classe, passo o tópico e do tópico acesso as informações. Não precisa de setters, apenas de getters.

Agora, no meu Controller, ele está reclamando porque esse método tem que devolver uma lista de <TopicoDto>, e não mais de Topico. Um lado negativo, uma parte chata é que vou ter a classe <Topico>, mas tenho que devolver o "Dto". Então, vou ter que fazer essa conversão de <Topico> para <TopicoDto>.

Para não deixar essa lógica solta, vou criar um método que a encapsula dentro da própria classe DTO. Vou pegar o TopicoDto e, dentro dele, vou criar um método chamado ´converter. Passo para ele a lista de Topico e ele devolve a lista de TopicoDto.

return TopicoDto.converter(Arrays.asList(topico, topico, topico));

}

public static List<TopicoDto> converter(List<Topico> topicos) {

return null;

}

}

Aqui, posso usar o Java 8 (A API de Stream do Java 8) para não ter que fazer manualmente. O código ficará:

return topicos.stream().map(TopicoDto::new);

A função do mapeamento será TopicoDto::new, porque ele vai chamar o construtor que recebe o próprio tópico como parâmetro. No final, tenho que transformar isso em uma lista, então vou encadear a chamada para o método collect(), passando collectors.toList() para transformar numa lista.

Essa é a sintaxe do Java 8. Sem ele, teríamos que pegar essa lista de tópicos, fazer um for para cada tópico, dar new no topicoDto, guardar em uma lista de topicoDto e devolver essa lista de topicoDto no final. Desse jeito, ele faz tudo isso em uma linha só usando API de strings do Java 8.

Nesta aula, aprendemos:

* Sobre a API que desenvolveremos ao longo do curso e sobre as classes de domínio dela;
* Que, para um método no *controller* não encaminhar a requisição a uma página JSP, ou Thymeleaf, devemos utilizar a anotação @ResponseBody;
* Que o Spring, por padrão, converte os dados no formato JSON, utilizando a biblioteca Jackson;
* Que, para não repetir a anotação @ResponseBody em todos os métodos do *controller*, devemos utilizar a anotação @RestController;
* Que, para não precisar reiniciar manualmente o servidor a cada alteração feita no código, basta utilizar o módulo **Spring Boot DevTools**;
* Que não é uma boa prática retornar entidades JPA nos métodos dos *controllers*, sendo mais indicado retornar classes que seguem o padrão **DTO** (*Data Transfer Object*);
* Os principais conceitos sobre o modelo arquitetural **REST**, como **recursos**, **URIs**, **verbos HTTP**, **Representações** e comunicação ***stateless***.

Banco de dados Spring

Properties

# data source

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.url=jdbc:h2:mem:alura-forum

spring.datasource.username=sa

spring.datasource.password=

# jpa

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.jpa.hibernate.ddl-auto=update

# h2

spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

No primeiro bloco, onde consta o comentário datasource, temos as configurações do datasource, do banco de dados para o Spring. A primeira coisa que precisamos dizer é qual o driver de acesso ao banco de dados (o driver do JDBC). No caso, passamos a classe do H2. Você colocaria a classe do seu driver - do seu banco de dados.

Por padrão, o Spring Boot usa o "Hibernate" como implementação da JPA

A segunda propriedade é a "URL" de acesso ao banco de dados. No nosso caso vai ser jdbc:h2:mem. O database vai se chamar alura-forum. Isto é, ele vai criar um banco de dados em memória chamado alura-forum.

As duas próximas propriedades são do usuário e da senha do banco de dados. Geralmente com o H2, o pessoal coloca o usuário "sa" e a senha fica em branco.

No próximo bloco estão configurações específicas da JPA. Estamos usando o hibernate e ele precisa saber qual é o dialeto do banco de dados. Por isso, precisamos passar uma classe do hibernate que representa o dialeto do banco de dados. No caso do H2, a classe é org.hibernate.dialect.HSDialect

Continuando, eu quero que o hibernate crie automaticamente o banco de dados, e, sempre que tiver uma mudança, uma nova tabela, uma nova coluna, puxe das propriedades das nossas entidades e atualize o banco de dados automaticamente. Então, colocamos a propriedade jpa.hibernate.ddl-auto=update.

Embaixo, no bloco seguinte, vem algo interessante. Como vou fazer para acessar o H2 se não tenho nada instalado na máquina? O Spring Boot tem uma comodidade - se você estiver utilizando o H2 - que é a ideia de ter um console do H2. Nada mais é do que uma interface gráfica para o H2 que acessamos direto pelo navegador. Precisamos habilitar essa interface gráfica, colocando a propriedade spring.h2.console.enable=true.

ENUM / @Enumerated

Por exemplo, o atributo status é um enum, e quero que o hibernate grave no banco de dados o nome da constante do enum, ao invés da ordem de declaração. Então, preciso colocar a anotação @Enumerated(), que, por padrão, grava a ordem, mas, como eu não quero guardar a ordem, vou passar o valor STRING.

Relacionamentos

Preciso dizer a cardinalidade, de tópico para o usuário, para o autor, que vai ser @ManyToOne, muitos para um. Para o Curso, a mesma coisa (um curso pode ter vários tópicos, mas o tópico pertence a um único curso). E a lista de resposta, List<Resposta>. Como é uma lista, vai ser @OneToMany, um tópico pode ter várias respostas.

No caso do @ManyToOne, tem que passar uma propriedade, que é o mappedBy, para ele não achar que é um novo mapeamento, porque na classe, e nela, a resposta, estará mapeado o relacionamento com o tópico. Então, mappedBy = "topico" que é o nome do atributo lá na classe resposta.

@Entity

public class Topico {

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String título;

private String mensagem;

private LocalDateTime dataCriacao = localDateTime.now();

@Enumerated(EnumType.STRING)

private StatusTopico status = StatusTopico.NAO\_RESPONDIDO;

@ManyToOne

private Usuario autor;

@ManyToOne

private Curso curso;

@OneToMany(mappedBy = "topico")

private List<Resposta> respostas = new ArrayList<>();

//...

}

Data.SQL

Só para termos algumas informações já populadas no nosso banco de dados, uma coisa que podemos fazer no projeto é: lá no diretório "src/main/resources", se você tiver um arquivo chamado data.sql (no meu projeto eu até já copiei esse arquivo), o Spring automaticamente pega os comandos SQL que estiverem aqui dentro e executa no banco de dados.

Banco H2

Ele já vem com alguns valores preenchidos, só precisamos preencher a "JDBC URL" com o endereço "jdbc:h2:mem:alura-forum", que é o mesmo configurado no application.properties. Seguindo, vou selecionar "Connect" e ele vai acessar o H2.

Repository

Poderíamos simplesmente injetar o EntityManager da JPA e fazer a consulta, mas isso não é uma boa prática. Geralmente, isolamos esse código de acesso ao banco de dados em alguma outra classe e injetamos essa outra classe no TopicosController.java. Em um projeto tradicional com Java, geralmente criávamos a classe seguindo padrão "*DAO*", que é o Data Access Object. Por exemplo, se temos a entidade Topico, teríamos uma classe TopicoDAO, e aí nessa classe teria todo o acesso ao banco de dados usando JPA, tudo encapsulado, abstraído para quem fosse chamar essa classe.

Nessa classe, a lógica seria: teríamos que injetar o EntityManager, criar um método lista(), por exemplo, e dentro do método lista(), usar a API do EntityManager, chamar o EntityManager, depois, o createQuery, montar o JPQL, fazer toda a consulta manualmente.

Pensando nisso, o pessoal do Spring Boot criou uma facilidade. Você não precisa criar uma classe, e não precisa implementar aqueles métodos que são sempre iguais e repetitivos. Não vamos trabalhar com o padrão DAO, vamos utilizar outro padrão chamado Repository. No Spring Data, não vamos criar uma classe, vamos criar uma interface e ela vai herdar de outra interface do Spring Data que já tem alguns métodos prontos e abstraídos para nós.

Como estou herdando, toda vez que herdo ganho tudo da interface ou da classe que estou herdando. Essa interface JpaRepository já tem vários métodos comuns. Daí que vem a facilidade. Você não precisa implementar esses métodos, porque eles são comuns neste tipo de classe.

A JPA exige que todas as entidades tenham o construtor *defaul, que por padrão se não especificada na entidade já é defaul , caso crie um construtor com valores, tbm deve criar um vazio.*

Busca com relacionamento

O Spring também consegue filtrar isso. Só precisamos mudar o padrão para findbyCurso sendo que Curso é entidade e eu não quero filtrar pelo curso, e eu concateno na sequência Nome. Isto, é findByCursoNome, onde Curso é a entidade de relacionamento e Nome é o atributo dentro dessa entidade de relacionamento. Dessa maneira, ele vai conseguir encontrar.

public interface TopicoRepository extends JpaRepository<Topico, Long> {

List<Topico> findByCursoNome(String nomeCurso);

}

Quando existe nomes de atributos iguais nas classes que se relacionam pode haver conflito. Quando estou falando CursoNome, estou falando do atributo ou do relacionamento Curso e, lá dentro, do atributo nome? Nesse caso, como faço para diferenciar? Você pode colocar um "underline", findByCurso\_Nome(String nomeCurso).

Ex:

@Entity

public class Topico {

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String cursoNome;

private String mensagem;

private LocalDateTime dataCriacao = LocalDateTime.now();

@Enumerated(EnumType.STRING)

private StatusTopico status = StatusTopico.NAO\_RESPONDIDO;

//...

}

Então usamos o \_ isso irá especificar que o nome é da class curso.

public interface TopicoRepository extends JpaRepository<Topico, Long> {

List<Topico> findByCurso\_Nome(String nomeCurso);

}

Você pode navegar dentro dos relacionamentos quantos níveis for. Poderia ter findbyCursoCategoriaNome(String nomeCurso) (se categoria fosse uma classe). Ele sai navegando entre os relacionamentos. Dentro do tópico ele vai buscar o curso, dentro do curso buscar a categoria, e dentro da categoria buscar o nome. Funciona muito bem. Ele gera todos os JOINs e monta a Query certinho.

@Query

 Existe uma anotação que você coloca, o @Query. E aí você importa do pacote, org.springframework.data.jpa.repository. Nos parênteses, entre aspas, você vai ter que montar a Query na mão, usando o JPQL. Então, ficaria @Query("SELECT t FROM Topico t WHERE t.curso.nome = :nomeCurso")

public interface TopicoRepository extends JpaRepository<Topico, Long> {

List<Topico> findByCursoNome(String nomeCurso);

@Query("SELECT t FROM Topico t WHERE t.curso.nome = :nomeCurso")

List<Topico> carregarPorNomeDoCurso(String nomeCurso);

}

A vantagem dessa segunda abordagem é que você coloca o nome do método que você quiser, em português, no estilo que você quiser, só que a desvantagem é que você vai ter que montar a Query manualmente. Você vai ter que colocar o @Query e gerar a Query manualmente com JPQL.

Tem mais um problema. Como eu tenho um parâmetro nomeCurso, o Spring não assume que esse parâmetro é o que está no método. Você precisa colocar mais uma anotação, que é o @Param. No @Param, você coloca "nomeCurso" (sem os dois pontos ":"). Aí fica a critério do que você definir com a sua equipe, o que vocês vão usar.

@Query("SELECT t FROM Topico t WHERE t.curso.nome = :nomeCurso")

List<Topico> carregarPorNomeDoCurso(@Param("nomeCurso")(String nomeCurso);

Nesta aula, aprendemos que:

* Para utilizar o JPA no projeto, devemos incluir o módulo Spring Boot Data JPA, que utiliza o **Hibernate**, por padrão, como sua implementação;
* Para configurar o banco de dados da aplicação, devemos adicionar as propriedades do *datasource* e do JPA no arquivo **src/main/resources/application.properties**;
* Para acessar a página de gerenciamento do banco de dados H2, devemos configurar o *console* do H2 com propriedades no arquivo **src/main/resources/application.properties**;
* Para mapear as classes de domínio da aplicação como entidade JPA, devemos utilizar as anotações @Entity, @Id, @GeneratedValue, @ManyToOne, @OneToMany e @Enumerated;
* Para que o Spring Boot popule automaticamente o banco de dados da aplicação, devemos criar o arquivo **src/main/resources/data.sql**;
* Para criar um *Repository*, devemos criar uma interface, que herda da interface JPARepository do Spring Data JPA;
* Para criar consultas que filtram por atributos da entidade, devemos seguir o padrão de nomenclatura de métodos do Spring, como por exemplo findByCursoNome;
* Para criar manualmente a consulta com JPQL, devemos utilizar a anotação @Query;

DTO CADASTRO

Tanto no lista() quanto no cadastrar(), não vamos trabalhar com a entidade. Vamos ter que ter outra classe, outro DTO que recebe essas informações, e não vou trabalhar com a entidade. Mas só para diferenciar quando é um DTO em que estou mandando dados para o cliente de um em que estou recebendo, vamos usar outro padrão de nomenclatura, vamos chamar essa classe de Form.

Agora vou definir quais campos vão chegar do cliente. Toda vez que cadastro um tópico preciso de algumas informações. Por exemplo: preciso do título, private String titulo; e da mensagem, private String mensagem, qual a dúvida em si desse tópico que está sendo criado.

Não preciso da data, porque posso instanciar na hora. Por exemplo, se abrirmos a nossa classe Topico.java, perceberemos que a data já está sendo preenchida, private LocalDateTime dataCriacao = LocalDateTime.now(). Na hora que eu dou new no Topico, já pego a data atual.

O id também não é necessário, porque o banco de dados que vai gerar. O status também não vou receber, porque, por padrão, se acabei de criar um tópico, ele está com o status NAO\_RESPONDIDO.

O curso, preciso receber. Mas, a ideia não é ter um objeto Curso, porque é a classe de domínio. Então, vamos receber o nomeCurso. Vai chegar o nome do curso e vou ter que buscar no banco de dados qual o curso que tem esse nome.

Essa class DTO é uma classe Java, um POJO, um Java Bean, que não tem anotação, dependência com nada, é só atributo *Getter* e *Setter*.

Vou receber o form, e tenho que converter para um topico.

Para fazer isso, podemos encapsular essa lógica dentro do próprio form. Na classe TopicoForm, posso ter um método converter(), isto é, form.converter(). E esse método devolve um objeto do tipo Topico. Dentro do form já tem todas as informações que eu preciso do topico, então esse converter() já cria para mim o tópico.

 Ao invés de criar uma variável e, para cada campo, setar linha por linha, podemos usar aquela ideia do construtor. Mas lembre-se que a JPA precisa que a classe tenha um construtor *default*. Mas, não significa que eu não possa ter outros construtores. Eu preciso ter um construtor *default*', mas posso criar outros construtores que uso internamente na aplicação.

public Topico(String titulo, String mensagem, Curso curso) {

this.titulo = titulo;

this.mensagem = mensagem;

this.curso = curso;

}

Agora, no tópico TopicoForm.java, na hora de chamar o converter() não vou usar o construtor padrão, vou usar o construtor que recebe o título, a mensagem e o curso, ou seja, Topico(titulo, mensagem, curso).

O titulo e a mensagem já existem, são atributos do TopicoForm. O problema é o curso. Eu não tenho um objeto curso, só tenho a informação nome do curso, nomeCurso. Então, preciso carregar a entidade curso do banco de dados.

Mas, no TopicoForm, não consigo injetar o Repository, porque essa classe não é um componente do Spring. Para simplificar, na hora de chamar o método converter(), posso passar um parâmetro, o próprio repository. Desta forma, teremos, public Topico converter(TopicoRepository repository). Com o repository em mãos, consigo carregar o curso pelo banco de dados.

Só que aqui tem um problema. Não é o TopicoRepository repository que eu quero, porque não quero carregar o topico quero carregar o Curso. Então, na verdade, o que vou receber como parâmetro vai ser um CursoRepository.

 No TopicoForm.java, preciso falar para o Repository, dado o curso, buscar pelo nome. Ou seja, Curso curso = cursoRepository.findByNome(). E aí, o meu parâmetro é nomeCurso.

public Topico converter(CursoRepository cursoRepository) {

Curso curso = cursoRepository.findByNome(nomeCurso);

return new Topico(titulo, mensagem, curso);

}

public Topico converter(CursoRepository cursoRepository) {

Curso curso = cursoRepository.findByNome(nomeCurso);

return new Topico(titulo, mensagem, curso);

}

Boas práticas de cadastro.

Quando estou cadastrando uma informação, postando uma nova informação no sistema, o ideal seria devolver outro código da família de sucesso, que é o código 201, de quando tenho uma requisição processada com sucesso, mas um novo recurso foi criado no servidor.

O Spring tem uma classe chamada ResponseEntity<> que recebe um *generic*. Esse *generic* é o tipo de objeto que vou devolver no corpo da resposta, que no caso, seria o topico. Mas lembre-se, nós não devolvemos a entidade, as classes de domínio, então na verdade vai ser um topicoDto (vamos devolver no corpo da resposta um DTO representando um recurso que acabou de ser criado).

Essa classe ResponseEntity tem alguns métodos estáticos para você criar um objeto ResponseEntity. Então, no return, teremos ResponseEntity.created() que devolve o 201. Só que esse método created() recebe um parâmetro uri e que eu preciso passar para ele.

Isso acontece porque toda vez que devolvo 201 para o cliente, além de devolver o código, tenho que devolver mais duas coisas: uma delas é um cabeçalho HTTP, chamado "Location", com a URL desse novo recurso que acabou de ser criado; a segunda coisa é que, no corpo da resposta, eu tenho que devolver uma representação desse recurso que acabei de criar. Então, quando eu chamo o método created(), ele fica esperando a uri do recurso que criamos para adicioná-la no cabeçalho Location.

No método cadastrar(), estou recebendo o form e posso colocar uma vírgula e depois uma classe do Spring chamada UriComponentsBuilder. Seguindo, vou chamar esse parâmetro de uriBuilder. É só declarar o UriComponentsBuilder como parâmetro que o Spring vai injetar no método para você.

@PostMapping

public ResponseEntity<TopicoDto> cadastrar(@RequestBody TopicoForm form, UriComponentsBuilder uriBuilder) {

Topico topico = form.converter(cursoRepository);

topicoRepository.save(topico);

URI uri =

return ResponseEntity.created(uri)

}

Nós vamos usar o uriBuilder. depois de URI uri = . Existem alguns métodos para ele criar um objeto uri. O método que vamos chamar é o path(). E aqui está o segredo: não vou passar o caminho completo, o caminho do servidor. Só vou passar o caminho do recurso.

O caminho do recurso será "/topicos/". Mas, se eu passar só "/tópicos", é a lista de tópicos, sendo que aqui, na verdade, criei só um único tópico. No geral, costumamos adicionar um id, que é dinâmico - é o id do tópico que acabei de criar. Coloco ele entre chaves para dizer que é um parâmetro dinâmico {id}.

Por último .buildAndExpand(), método que temos que chamar, passando como parâmetro um valor a ser substituído no espaço do {id}, que é dinâmico. No caso, vou puxar o id do tópico que acabei de criar no banco de dados e ele vai substituir esse {id} e jogar na uri.

No final, tem um método .toUri(), que converte e transforma na URL completa, com endereço do servidor e com valores dinâmicos que posso passar como parâmetro no buildAndExpand.

@PostMapping

public ResponseEntity<TopicoDto> cadastrar(@RequestBody TopicoForm form, UriComponentsBuilder uriBuilder) {

Topico topico = form.converter(cursoRepository);

topicoRepository.save(topico);

URI uri = uriBuilder.path("/topicos/{id}").buildAndExpand(topico.getId()).toUri();

return ResponseEntity.created(uri).body(new TopicoDto(topico));

}

Postman / Content- Type

Para simular no POST, lembre-se que a URL é a mesma, "/topicos". Também precisamos trocar o método de "GET" para "POST". Mas aí, antes de disparar a requisição, preciso configurar duas coisas. A primeira é que na aba "Body", preciso levar o corpo da requisição, os parâmetros, as informações que preciso enviar para fazer o cadastro.

Na aba "Body", vamos selecionar a opção "raw", porque vamos mandar o JSON diretamente. Seguindo, na caixa de texto, vou abrir chaves e, no JSON, preciso passar, entre aspas, o nome do campo. Por exemplo, título "titulo":"Dúvida Postman",. Também precisamos mandar a mensagem, então, "mensagem":"Texto da mensagem",. A terceira informação é o nome do curso, isto é, "nomeCurso":"Spring Boot" (precisamos indicar um valor que existe no nosso banco de dados e o que temos lá é o "Spring Boot").

{

"titulo":"Dúvida Postman",

"mensagem":"Texto da mensagem",

"nomeCurso":"Spring Boot"

}

Esse é o corpo da requisição (quero testar uma requisição do tipo POST levando esse JSON como parâmetro). Estamos trabalhando com JSON, só que, no nosso código, TopicosController.java, recebemos o Form, que é um DTO, e aí o Spring pega o JSON que está vindo na requisição - no @RequestBody da requisição - e converte para um objeto TopicoForm.

Além disso, ainda preciso definir um cabeçalho. Ppara isso, existe a aba chamada "Headers". Como estou mandando um JSON, preciso avisar para o servidor que o corpo da requisição está no formato JSON. Preciso indicar isso por um cabeçalho chamado "Content-Type", que eu seleciono abrindo a aba "Headers". Desta forma, na chave, "Key", do "Headers" eu selecionarei "Content-Type" e no "Value", selecionarei "application/json".

O cabeçalho "Content- Type" é para o cliente dizer qual o conteúdo que está sendo enviado. E aí, no nosso caso, passamos "JSON".

Se olharmos um pouco acima do código, na parte centro direita da tela, encontraremos "Status" marcando "201 Created". Ou seja, ele realmente devolveu 201. Na mesma altura, lado esquerdo, temos a aba "Headers" onde encontraremos os cabeçalhos que foram devolvidos do servidor.

"**Location** -> <http://localhost:8080/topicos/4>

**Content-Type** -> application/json:charset=UFT-8

**Transfer-Encoding** -> chunked

**Date** -> Tue, 14 May 2019 17:29:01 GMT"

Ao testar o cadastro de um novo tópico pelo Postman, foi necessário adicionar um cabeçalho na requisição, chamado Content-Type. Qual o objetivo desse cabeçalho?

Indicar o formato das informações enviadas no corpo da requisição

O cabeçalho Content-Type serve exatamente para indicar o tipo de conteúdo sendo enviado ao servidor.

Nesta aula, aprendemos:

* Que para evitar repetir a URL em todos os métodos, devemos utilizar a anotação @RequestMapping em cima da classe *controller*;
* Que para mapear requisições do tipo POST, devemos utilizar a anotação @PostMapping;
* Que para receber dados enviados no corpo da requisição, a boa prática é criar uma classe que também siga o padrão **DTO** (*Data Transfer Object*);
* Que a boa prática para métodos que cadastram informações é devolver o código HTTP 201, ao invés do código 200;
* Que para montar uma resposta a ser devolvida ao cliente da API, devemos utilizar a classe ResponseEntity do Spring;
* Que para testar requisições do tipo POST, precisamos utilizar alguma ferramenta de testes de API Rest;
* Como utilizar o Postman para testar uma API Rest;

Validações com Bean Validation

Para validar os dados recebidos, como esses dados são um DTO, as anotações de validação vão no DTO.

Essas anotações fazem parte do Bean Validation que posso utilizar para fazer a validação. Posso colocar, por exemplo, que a mensagem tem que ter no mínimo 10 caracteres, @NotNull @NotEmpty @Length(min = 10), o título no mínimo 5 caracteres, @NotNull @NotEmpty @Length(min = 5) . Tudo como @NotNull,não nulo, e @NotEmpty, não vazio.

O Bean Validation também é flexível. Você pode criar uma nova anotação, caso não tenha. Por exemplo, você quer validar um campo CPF. Você pode criar uma anotação @CPF e ensinar como é essa validação. É uma especificação bem bacana e simples de trabalhar.

No nosso TopicosController.java, no Topicoform, além de estar anotado o @RequestBody - para dizer para o Spring que o TopicoForm está vindo no corpo da requisição - temos que colocar mais uma anotação, o @Valid - que é do próprio Bean Validation - para avisar para o Spring: quando você for injetar o TopicoForm, puxando os dados que estão vindo na requisição, rode as validações, @Valid, do Bean Validation. Nessa classe tem anotações e eu quero que você verifique se o objeto chegando é válido, conforme as anotações que eu coloquei.

*Controller Advice*.  (Tratamento de erros)

 O Spring tem uma solução para esse tipo de cenário. A solução é criar uma espécie de interceptador.

Toda vez que acontecer uma *exception*, em qualquer método de qualquer Controller, o Spring automaticamente vai chamar esse interceptador, e nós faremos o tratamento apropriado. Esse interceptador é chamado de *Controller Advice*. A ideia é criarmos uma classe e transformá-la em um *Controller Advice*, onde vamos fazer o tratamento do erro.

"ErroDeValidacaoHandler".java, preciso dizer para o Spring que a classe ErroDeValidacaoHandler é um *Controller Advice*. Existe a anotação @ControllerAdvice, mas, no nosso caso, usaremos @RestControllerAdvice, porque estamos usando RestController da API REST.

Agora, precisamos ensinar para o Spring que esse *Controller Advice* vai fazer tratamento de erros, para quando aparecer uma exceção. Por exemplo, invalidação de formulários.

Para isso, vamos criar um método usando void por enquanto, public void handle(), sendo que é o método handle() que fará o tratamento do erro. Preciso dizer para o Spring que esse método deve ser chamado quando houver uma exceção dentro de algum Controller. Para falar isso para o Spring, em cima do método, vamos colocar uma anotação @ExceptionHandler, que é do próprio Spring.

E temos que passar como parâmetro que tipo de exceção que, quando acontecer dentro do Controller, o Spring vai direcionar para o método handler(). No nosso caso, é exceção de validação de formulário. Quando dá um erro de validação de formulário, usando Bean Validation, que exceção que o Spring lança? Ele manda uma *exception* chamada MethodArgumentNotValidException. Temos que passar o nome da classe, isto é, MethodArgumentNotValidException.class. Agora o Spring sabe que se acontecer essa *execption* em qualquer RestController, ele vai cair no método @ExceptionHandler().

No método handle(), preciso pegar a exceção que aconteceu, para pegar as mensagens, fazer o tratamento. Esse método recebe como parâmetro um objeto do tipo MethodArgumentNotValidException, que é o mesmo *Handler* que estou utilizando no@ExceptionHandler(). Chamarei o parâmetro de exception.

Toda vez que acontecer alguma *exception* desse tipo em qualquer RestController do projeto, o Spring vai chamar esse método handle() passando como parâmetro a *exception* que aconteceu. Tendo esse interceptador, o Spring considera que deu um erro, mas que não vai devolver o código 400 para o cliente. Ele vai chamar o interceptador e, aqui dentro, você faz o tratamento. Ele considera que fizemos o tratamento e, por padrão, ele vai devolver 200 para o cliente. Mas não quero que ele devolva 200, quero que ele devolva 400. Então, em cima do método, escreverei @ResponseStatus() para falar para ele qual o status que ele vai devolver. O parâmetro será code = HttpStatus.BAD\_REQUEST

@RestControllerAdvice

public class ErroDeValidacaoHandler {

@ResponseStatus(code = HttpStatus.BAD\_REQUEST)

@ExceptionHandler(MethodArgumentoNotValidException.class)

public void handle(MethodArgumentNotValidException exception) {

}

}

 Por mais que eu tenha tratado o erro, não é para devolver 200. É para continuar devolvendo 400, porque meu tratamento é só para mudar as mensagens. Agora, dentro do código, eu faço o tratamento.

Só que o Spring, além de devolver o @ResponseStatus, devolve o que o método estiver retornando. Então, nosso método não pode ser void. Ele tem que devolver alguma coisa, que, no nosso caso, vai ser uma lista com as mensagens de erro. Mas ao invés de ser aquela mensagem bizarra, enorme, do Spring, vai ser uma mensagem que nós vamos personalizar.

Esse método vai devolver um List<>. Para representar o erro, vou criar um novo DTO que serve para representar um erro de formulário. Vou criar uma classe chamada ErroDeFormularioDto.

Essa classe não existe, então vou criá-la selecionando, no atalho, "Create class 'ErroDeFormularioDto". Depois, em "Create a New Java Class" - considerando que essa classe faz parte da configuração de validação - a colocarei no mesmo pacote "br.com.alura.forum.config.validacao". Essa classe ErroDeFormularioDto que vai representar um erro de validação. O JSON, que vai ser devolvido para o cliente, não vai ser mais aquele imenso do Spring. Vai ser o JSON representado por essa classe.

Nessa classe, o que preciso ter de informações para mandar para o JSON? Só duas: primeiro, qual foi o campo que deu erro (se foi título, mensagem, nome do curso), então vou criar um atributo chamado campo; e o segundo, é uma String também, correspondente à mensagem - qual o erro - logo, chamarei de erro.

E aí vou declarar um construtor. Para isso selecionarei "Generate Construtor using Fields" usando o atalho. Na próxima tela, selecionados "campo" e "erro", basta apertar "Generate". Agora, quando der new nessa classe, já passarei campo e mensagem de erro. Usarei de novo o atalho para gerar os *Getters* e *Setters*, selecionando "Generate Getters and Setters". Só que no caso, como já estou recebendo os parâmetros do construtor, só vou gerar os *Getters*. Sendo assim, seleciono "campo" e "erro" e, em seguida, "Select Getters".

O DTO, ErroDeFormularioDto, representa o erro de algum campo. Ele tem o nome do campo, private String campo, e uma mensagem de erro, private String erro. Era só o que eu queria: para cada campo, o nome do campo e a mensagem de erro.

public class ErroDeFormularioDto {

private String campo;

private String erro;

public ErroDeFormularioDto(String campo, String erro) {

this.campo = campo;

this.erro = erro;

}

public String getCampo() {

return campo;

}

public String getErro() {

return erro;

}

}

Voltando ao nosso ErroDeValidacaoHandler.java. Esse que vai ser o retorno: List<ErroDeFormularioDto> handle(MethodArgumentNotValidException exception). Isto é, ele vai devolver uma lista com cada um dos erros que aconteceram.

Como eu descubro que erros aconteceram? Por isso o método handle() recebe o tal do (MethodArgumentNotValidException exception). Dentro desse objeto tem todos os erros que aconteceram.

Se pegarmos o parâmetro exception, seguido do método getBindingResult(), com o resultado das validações, e do método getfieldErrors() que contém todos erros "*Field*", de formulário.

com o comando "Ctrl + 1", usarei o atalho para criar uma variável local que chamarei de fieldErrors. Essa variável tem os erros de formulário. Só que não quero devolver essa lista, List<>, de FieldError. Quero devolver uma lista de ErroDeFormulárioDto. Então:

O que vou devolver nesse método é justamente o DTO, return dto. Estou devolvendo uma lista de erro de formulário DTO. Só que no momento essa lista está vazia.

Agora que peguei os fieldErrors, vou ter que percorrer essa lista para cada *field error* criar um objeto *erro dto* e guardar nessa lista representada pela minha variável dto (List<ErroDeFormularioDto> dto = new ArrayList<>()).

Agora vou usar os recursos de *stream*, de interação, do Java8. Então, vou chamar direto o forEach(), ( fieldErrors.forEach()) e passar um lâmbda, indicando: para cada erro, o que quero fazer, isto é, fieldErrors.forEach(e -> {. Neste caso, quero criar um erro de formulário DTO, então, ErroDeFormularioDto erro = new ErroDeFormularioDto(campo, erro).

Lembre-se que, quando eu dou new no ErroDeFormularioDto(), tenho que passar uma mensagem e qual o campo que deu erro. O campo pego pelo e.getField(). O próprio erro do FieldError já tem qual é o nome do campo.

Agora, para pegar a mensagem, começaremos esrevendo String mensagem =. Como podemos ter aquele esquema de internacionalização, isto é, de mensagens em vários idiomas, o Spring nos ajuda com uma classe chamada MessageSource. Vou injetar com o @Autowired, nessa classe Handler um atributo do tipo messageSource. Essa classe MessageSource te ajuda a pegar mensagens de erro, de acordo com o idioma que o cliente requisitar.

@RestControllerAdvice

public class ErroDeValidacaoHandler {

@Autowired

private MessageSource messageSource;

@ResponseStatus(code = HttpStatus.BAD\_REQUEST)

@ExceptionHandler(MethodArgumentNotValidException.class)

public List<ErroDeFormularioDto> dto = new ArrayList<>();

List<FiledError> fieldErrors = exception.getBindingResult().getFieldErrors():

fieldErrors.forEach(e -> {

String mensagem =

ErroDeFormularioDto erro = new ErroDeFormularioDto(e.getField(), mensagem);

});

return dto;

}

Para pegar mensagens de erro, String mensagem = messageSource.getMessage() e vou passar como parâmetro o erro e, e uma classe do Spring chamada localeContextHolder.getLocaLe(), para ele descobrir qual o "*locale*", qual o local atual para pegar a mensagem no idioma correto.

String mensagem = messageSource.getMessage(e, localeContextHolder.getLocaLe());

Agora falta só pegar o dto, que é minha lista de erros, e adicionar uma mensagem de erro add(erro).

List<FieldError> fieldErrors = exception.getBindingResult().getFieldErrors();

fieldErrors.forEach(e -> {

String mensagem = messageSource.getMessage(e, localeContextHolder.getLocaLe());

ErroDeFormularioDto erro = new ErroDeFormularioDto(e.getField(), mensagem);

dto.add(erro);

});

return dto;

}

Essa vai ser a lógica do nosso ErroDeValidacaoHandler.java. É um pouco estranha, mas a ideia do *Handler* é justamente isso. Pense que o *Handler* é um interceptador, em que estou configurando o Spring para, sempre que houver um erro - alguma *exception* em algum método de qualquer Controller - ele chamar automaticamente o interceptador, passando a lista com todos os erros que aconteceram. Sendo assim, eu pego essa lista e transformo no meu objeto ErroDeFormularioDto, que só tem o nome do campo e a mensagem para simplificar.

Pronto. É só isso. Você só precisa criar esse *Handler*. Não precisa mexer nos Controllers. Automaticamente o Spring sabe que é para chamar esse erro em qualquer Controller quando houver um erro de validação.

Nesta aula, aprendemos que:

* Para fazer validações das informações enviadas pelos clientes da API, podemos utilizar a especificação ***Bean Validation***, com as anotações @NotNull, @NotEmpty, @Size, dentre outras;
* Para o Spring disparar as validações do *Bean Validation* e devolver um erro 400, caso alguma informação enviada pelo cliente esteja inválida, devemos utilizar a anotação @Valid;
* Para interceptar as *exceptions* que forem lançadas nos métodos das classes *controller*, devemos criar uma classe anotada com @RestControllerAdvice;
* Para tratar os erros de validação do *Bean Validation* e personalizar o JSON, que será devolvido ao cliente da API, com as mensagens de erro, devemos criar um método na classe @RestControllerAdvice e anotá-lo com @ExceptionHandler e @ResponseStatus.

Get by ID @GetOne

@GetMapping("/{id}")

public TopicoDto detalhar(@PathVariable("id") Long codigo) {

Mas se você não quiser chamar o parâmetro de id, mas, sim, de código, o Spring já não iria fazer essa assimilação, ele não iria saber que código é o id do @GetMapping("/{id}"). Nesse caso, no @PathVariable() eu deveria passar o "id" entre aspas, "id".

Como buscar por id é uma ação comum, já tem um método no Spring chamado getOne(id), em que você passa um id e ele te devolve o objeto tópico que é nossa entidade.

@GetMapping("/{id}")

public TopicoDto detalhar(@PathVariable Long id) {

Topico topico = topicoRepository;getOne(id);

}

}

Mas está faltando o return. Então: return new TopicoDto(topico). Lembrando que na hora em que dou new no TopicoDto posso passar um tópico como parâmetro, e ele converte para um DTO. E pronto, ele vai compilar tudo certinho.

@GetMapping("/{id}")

public TopicoDto detalhar(@PathVariable Long id) {

Topico topico = topicoRepository;getOne(id);

return new TopicoDto(topico);

}

}

Só para dar uma variada, vamos voltar ao Topicos.Controller.java, não quero mais devolver o TopicoDto. O TopicoDto que criamos na hora de fazer a lógica de listar todos os tópicos, só traz quatro informações: id, titulo, mensagem e dataCriacao. Imagine que quero novas informações. Por exemplo, quero saber o status do tópico, as respostas, quem foi o usuário. Eu até poderia ir no TopicoDto.java e adicionar novos atributos. Só que isso ia influenciar também na lógica de listar todos os tópicos.

Aí que entra a vantagem de ter um DTO: a flexibilidade. No método lista(), posso usar o topicoDto, que só tem aquelas quatro informações, mas no detalhar() posso usar um outro DTO que traz mais informações. Por exemplo, posso usar um DetalhesDoTopicoDto. Sendo assim, vou criar outro DTO que é específico para essa lógica de detalhar.

public class DetalhesDoTopicoDto {

private Long id;

private String titulo;

private String mensagem;

private LocalDateTime dataCriacao;

private String nomeAutor;

private StatusTopico status;

}

Além dos campos do DTO acima, também quero uma lista com as respostas do tópico, private List<Resposta> respostas;. Só que resposta é uma entidade. Não quero devolver o objeto resposta inteiro, então vou criar um DTO para a resposta, private List<RespostaDto> respostas. Abrindo o atalho, selecionarei "Create class 'RespostaDto'". Na próxima tela, criarei a classe RespostaDto.java no pacote DTO.

Na classe RepostaDto.java, quero devolver da resposta: o id, private Long id; a mensagem, private String mensagem; a data de quando foi criada a resposta, private LocalDateTime dataCriacao; e o nome do autor, quem postou a resposta, private String nomeAutor.

Vou usar aquele mesmo procedimento de criar um construtor que já recebe todos esses parâmetros (para isso, com o atalho, seleciono "Generate Constructor using Fields". Na próxima tela, pressiono "Select All" com todos os itens selecionados - id, mensagem, dataCriacao, nomeAutor - seguido de "Generate" para finalizar).

Aliás, nesse caso não era um construtor recebendo os parâmetros, na verdade o construtor recebe a Resposta, o objeto resposta, e a partir dele que pegamos os parâmetros this.id = resposta.getId(). Mesma coisa para os outros campos: this.mensagem = resposta.getMensagem(); this.dataCriacao = resposta.getDataCriacao(); sobre o nome do autor, this.nomeAutor = resposta.getAutor().getNome(), não existe um campo "nomeAutor", mas tem o relacionamento para autor e pelo autor eu consigo chegar no nome do autor.

public class RespostaDto {

private Long id;

private String mensagem;

private LocalDateTime dataCriacao;

private String nomeAutor;

public RespostaDto(Resposta resposta) {

this.id = resposta.getId();

this.mensagem = resposta.getMensagem();

this.dataCriacao = resposta.getDataCriacao();

this.nomeAutor = resposta.getAutor().getNome();

}

}

 É parecido com o que fizemos no TopicoDto.java. Só preciso criar os *Getters* (Pelo atalho, seleciono "Generate Getters and Setters". Na próxima tela, pressiono "Select Getters" com todos os itens selecionados - dataCriacao, id, mensagem e nomeAutor. Para finalizar, basta apertar "Generate") lembrando que no DTO não precisa dos *Setters* porque no construtor já recebo todos os parâmetros.

Agora, vou apenas copiar o construtor que está no Topico.Dto.java para o DetalhesDoTopicoDto.java e mudar o nome da classe para DetalhesDoTopicoDto. Preenchi tudo que já tinha no outro.

Falta só preencher algumas coisinhas: nome do autor, this.nomeAutor = topico.getAutor().getNome(); status, que vem diretamente do tópico,this.status = topico.getStatus(), sendo que o Jackson, na hora de serilizar vai colocar o nome da constante; respostas, this.respostas = new ArrayList<>(), como a lista está nula, preciso instanciar; agora, dada a resposta, quero uma resposta DTO e coletar ela numa lista, portanto, this.respostas.addAll(topico.getRespostas(). Só que o getRespostas() me devolve uma lista de respostas e eu preciso de uma lista de RespostaDto. Sendo assim, seguirei escrevendo .stream.map(RespostaDto::new) e, depois, vou coletar isso numa lista, .collect(collectors.toList())). Estamos utilizando o recurso do Java8, da API de Strings, só para simplificar.

public DetalhesDoTopicoDto(Topico topico) {

this.id = topico.getId();

this.titulo = topico.getTitulo();

this.mensagem = topico.getMensagem();

this.dataCriacao = topico.getDataCriacao();

this.nomeAutor = topico.getAutor().getNome();

this.status = topico.getStatus();

this.respostas = new ArrayList<>();

this.respostas.addAll(topico.getRespostas().stream.map(RespostaDto::new).collect(collectors.toList()))

}

Controller

@GetMapping("/{id}")

public DetalhesDoTopicoDto detahar(@PathVariable Long id) {

Topico topico = topicoRepository.getOne(id);

return new DetalhesDoTopicoDto(topico);

}

Atualização

nesse método atualizar(), além do id, vou precisar dos dados do tópico que quero atualizar. Ou seja, vou precisar receber um DTO com os dados, parecido com o que fizemos no cadastrar(). Só que no cadastrar() não estávamos usando o padrão de nomenclatura DTO e, sim, o padrão *Form*. No atualizar() também preciso receber oTopicoForm form, ele vem no corpo da requisição, e tem que ter validação, porque não posso atualizar com algo inválido. Então, vou copiar o TopicoForm form que vem do método cadastrar().

@PutMapping("/{id}")

public ReponseEntity<TopicoDto> atualizar(@PathVariable Long id, @RequestBody @Valid TopicoForm form) {

}

Mas tem um detalhe. Eu não vou receber um TopicoForm, porque também posso querer ter flexibilidade. Tem algumas situações em que no cadastro preencho algumas informações, mas na hora de atualizar não posso atualizar todas as informações. Tem um ou outro campo que é somente leitura que não posso atualizar. Se eu receber o mesmo *Form*, nele poderiam vir campos que não deveria atualizar.

No geral, o ideal seria ter outra classe Form que representa a atualização.

Agora, no AtualizacaoTopicoForm.java, colocarei só os campos que eu quero de fato atualizar. Vai ser um pouco parecido com o TopicoForm.java.

public class AtualizacaoTopicoForm {

@NotNull @NotEmpty @Length(min = 5)

private String titulo;

@NotNull @NotEmpty @Length(min = 10)

private String mensagem;

//Getters e Setters.

}

Nessa lógica, tenho que fazer o seguinte: está chegando como parâmetro o id do tópico que quero atualizar, então preciso carregá-lo do banco de dados. Depois preciso sobrescrever com as informações que foram enviadas pelo usuário, pelo cliente no AtualizacaoTopicoForm. Para não ficar com essa lógica no Controller, vou encapsular essa lógica dentro da classe *Form*. Eu vou chamar de form.atualizar().

Vou ter um método em que passo as informações que não consigo ter dentro do *Form*, por exemplo, o id, que vai chegar no Controller. Vou precisar também do topicoRepository, porque no *Form* não consigo fazer injeção de dependência, e o método form.atualizar() vai atualizar e vai me devolver o topico como parâmetro, como resposta. Sendo que o tópico que vai ser devolvido é o tópico com as informações atualizadas.

@PutMapping("/{id}")

public ResponseEntity<TopicoDto> atualizar(@PathVariable Long id, @RequestBody @Valid AtualizacaoTopicoForm form) {

Topico topico = form.atualizar(id, topicoRepository);

}

Ele está reclamando, porque o método atualizar() não existe. Vou mandar ele criar selecionando atualizar() e, com o atalho, apertar "Create method 'atualizar(Long, TopicoRepository)'". Depois vou colocar a lógica de buscar o tópico pelo id no banco de dados e atualizar as informações que foram modificadas.

Para buscar um tópico pelo id, já vimos como fazer, topicoRepository.getOne(id), passo o id, que chegou como parâmetro do método. Pego o tópico, que veio do banco de dados com as informações desatualizadas. Agora vou sobrescrever tudo que eu quiser sobrescrever topico.setTitulo(), e passo o this.titulo, que é o que veio no JSON que o usuário enviou. Mesma coisa com a mensagem, topico.setmensagem(this.mensagem). E aí eu retorno o tópico que foi atualizado return topico.

public Topico atualizar(Long id, TopicoRepository topicoRepository) {

Topico topico = topicoRepository.getOne(id);

topico.setTitulo(this.titulo);

topico.setMensagem(this.mensagem);

return topico;

}

Para atualizar no banco de dados, não precisamos chamar nenhum método do Repository, porque a partir do momento em que carreguei ele do banco de dados pelo id, pela JPA ele já está sendo gerenciado. Qualquer atributo que eu setar, no final do méotodo, o Spring roda dentro de uma transação. Então eu vou carregar o tópico do banco de dados, no final do método ele vai commitar a transação, a JPA vai detectar que foram alterados os atributos e ela vai disparar o update no banco de dados automaticamente, não preciso chamar.

Quando chegar na linha Topico topico = form.atualizar(id, topicoRepository), ele já atualizou em memória e, quando acabar o método, ele atualiza no banco de dados.

Preciso devolver uma resposta. No caso, vou usar a classe ResponseEntity para montar uma resposta. Eu não estou criando um novo recurso, então não vou chamar o método created(), vou chamar só o ok(), e aí passo como parâmetro um new TopicoDto(topico), com esse tópico que foi atualizado. Esse parâmetro do ok() é o corpo que vai ser devolvido como resposta pelo servidor.

return ResponseEntity.ok(new TopicoDto(topico));

Vamos precisar colocar, no TopicosController.java, o @Transactional, que é para avisar para o Spring que é para commitar a transação no final do método.

@PutMapping("/{id}")

@Transactional

public ResponseEntity<TopicoDto> atualizar(@PathVariable Long id, @RequestBody @Valid AtualizacaoTopicoForm form) {

Topico topico = form.atualizar(id, topicoRepository);

return ResponseEntity.ok(new TopicoDto(topico));

}

@Transactional

No último vídeo, foi mostrado como implementar a funcionalidade para atualizar um tópico, sendo que no método do *controller* foi necessário adicionar a anotação @Transactional. Quais os objetivos dessa anotação?

Efetuar o *commit* automático da transação, caso não ocorra uma *exception*

Alternativa correta! Ao finalizar o método, o Spring efetuará o *commit* automático da transação, caso nenhuma *exception* tenha sido lançada.

Executar o método dentro de um contexto transacional

Alternativa correta! Métodos anotados com @Transactional serão executados dentro de um contexto transacional.

Exclusão

Quero remover um tópico específico, então na URL vai ter o parâmetro do id que vai chegar no método. Neste caso, como é exclusão, vamos mapear esse método com o @DeleteMapping (em lógica de exclusão, usamos o método delete() no REST). Na URL tem que ter o "/{id}", para assim dizer, com a URL, qual o tópico que quero excluir de fato.

Neste caso, qual vai ser meu retorno? Tenho que devolver um código "200 OK" (excluí um registro e a requisição foi processada com sucesso). Mas como eu o excluí do banco de dados, o recurso não existe mais. Eu não vou devolver nenhum corpo na resposta. Vou copiar o return ResponseEntity.ok(new TopicoDto(topico)), do método atualizar(), mas vou deixar o ok()vazio, já que só quero devolver o "200 OK".

@DeleteMapping("/{id}")

public ResponseEntity<TopicoDto> remover(@PathVariable Long id) {

topicoRepository.deleteById(id);

return ResponseEntity.ok();

}

Ele vai reclamar, porque o retorno do meu método é ResponseEntity e tem um <TopicoDto> nos *generics*, então vou apagá-lo. Não quero devolver nada no *generics*. Ele continua reclamando, porque falta o método build(). Ele seguirá reclamando porque o ResponseEntity tem um *generics*. Posso até colocar uma interrogação ("?"), só para dizer que é um *generics*, mas não sei qual é o tipo, e não especifiquei nada.

@DeleteMapping("/{id}")

public ResponseEntity<?> remover(@PathVariable Long id) {

topicoRepository.deleteById(id);

return ResponseEntity.ok().build();

}

Tratando o erro 404

Mas e se eu passar por exemplo <http://localhost:8080/topicos/999>? A URL é válida, mas não existe um tópico com esse id no banco de dados. Disparando a requisição, ele vai dar um erro (deu um erro no servidor e esse erro voltou para o cliente) e voltará uma *stack trace*. Não deveríamos devolver esse erro para o usuário.

[01:10] Isso aconteceu porque no exemplo anterior estávamos chamando o método detalhar(). O erro aconteceu exatamente na linha Topico topico = topicoRepository.getOne(id). Quando usamos o topicoRepository para buscar pelo id, estávamos utilizando o método getOne(). O getOne() funciona assim: passamos um id como parâmetro e ele considera que existe um registro, um tópico com esse id no banco de dados. Se ele não encontrar esse registro, ele não devolve nulo. Ele joga uma *exception*. Aconteceu uma exception no meio do caminho e essa *exception* vazou para o cliente.

@GetMapping("/{id}")

public DetalhesDoTopicoDto detalhar(@PathVariable Long id) {

Topico topico = topicoRepository.getOne(id);

return new DetalhesDoTopicoDto(topico);

}

O ideal aqui seria fazermos um tratamento. Antes de detalhar(), atualizar() e remover(), primeiro vou verificar se existe um tópico com esse id. Se existir, ok, executo. Se não existir, ao invés de voltar um erro imenso, devolvo "404", que é o código de "Not Found" - e que faz sentido neste caso. O cliente mandou uma URL com o id de um recurso que não existe, que não foi encontrado.

Para fazer isso, vamos fazer a seguinte alteração. No detalhar(), não vou mais usar o método getOne(), porque ele, por padrão, se não achar o elemento vai dar erro. Até poderia fazer um *Try/Catch*, se cair no *catch* devolver o "404", mas tem outro jeito. Existe outro método nos *Repositorys* chamado findById(). Ele também tem a mesma lógica. Você passa um id e ele vai fazer uma consulta filtrando pelo id no banco de dados, só que se ele não encontrar, ele não joga *exception*.

@GetMapping("/{id}")

public DetalhesDoTopicoDto detalhar(@PathVariable Long id) {

Topico topico = topicoRepository.findById(id);

return new DetalhesDoTopicoDto(topico);

}

Está até dando erro de compilação, porque o retorno do findById() não é a entidade topico. Ele devolve um Optional, que é uma classe que veio na API do Java 8. Tenho que mudar meu retorno para ser um Optional<topico> (isto é, para ser um *Optional* de tópico). Vou importar o Optional (abrindo o atalho, em "Choose type to import", seleciono "java.util.Optional" e ele devolverá um Optional).

@GetMapping("/{id}")

public DetalhesDoTopicoDto detalhar(@PathVariable Long id) {

Optional<Topico> topico = topicoRepository.findById(id);

return new DetalhesDoTopicoDto(topico);

}

Como o próprio nome já diz, o Optional é opcional. Tenho que verificar se nesse Optional tem um registro. Se não tiver, devolvo "404". Se tiver, eu retorno esse TopicoDto, conforme estava funcionando antes. Então vou fazer um if. Ou seja, if (topico.isPresent()). Se existe um registro de fato presente, vou retornar um return new DetalhesDoTopicoDto(topico), passando como parâmetro topico.

Vai dar erro de compilação, porque o parâmetro topico agora é o optional, mas, nesse caso, não quero o optional. Então, tenho que chamar o método get(), que é para pegar o topico de fato que está dentro do Optional. Enão, eu só vou chamar o get() dentro do if (topico.isPresent()), isto é, se estiver presente.

@GetMapping("/{id}")

public DetalhesDoTopicoDto detalhar(@PathVariable Long id) {

Optional<Topico> topico = topicoRepository.findById(id);

if (topico.isPresent()) {

return new DetalhesDoTopicoDto(topico.get());

}

}

Se entrou no if, if (topico.isPresent()), vai devolver o DTO. Se não entrou, é porque ele não está presente. Ou seja, não encontrei um tópico com esse id. Se chegou, preciso devolver o "404". Para fazer isso, vou retornar o ResponseEntity.notFound().build para montar o objeto ResponseEntity.

return ResponseEntity.notFound().build();

Só que agora vai dar um problema, porque o método detalhar() está devolvendo um DetalhesDoTopicoDto, não o ResponseEntity. Para eu devolver um notFound(), preciso que o retorno do método seja o ResponseEntity. Então precisamos alterar para que ele devolva um ResponseEntityde <DetalhesDoTopicoDto>.

@GetMapping("/{id}")

public ResponseEntity<DetalhesDoTopicoDto> detalhar(@PathVariable Long id) {

Optional<Topico> topico = topicoRepository.findById(id);

if (topico.isPresent()) {

return new DetalhesDoTopicoDto(topico.get());

}

}

Essa parte funcionou, mas agora está dando problema no if, porque dentro do if não estou devolvendo um ResponseEntity. Então, vou devolver um ResponseEntity e, no caso do detalhar(), vou devolver um ok(), e como parâmetro do ok(), no corpo da requisição é que mando o DTO.

if (topico.isPresent()) {

return ResponseEntity.ok(new DetalhesDoTopicoDto(topiico.get());

}

Nesta aula, aprendemos que:

* Para receber parâmetros dinâmicos no *path* da URL, devemos utilizar a anotação @PathVariable;
* Para mapear requisições do tipo PUT, devemos utilizar a anotação @PutMapping;
* Para fazer o controle transacional automático, devemos utilizar a anotação @Transactional nos métodos do *controller*;
* Para mapear requisições do tipo DELETE, devemos utilizar a anotação @DeleteMapping;
* Para tratar o erro 404 na classe *controller*, devemos utilizar o método findById, ao invés do método getOne, e utilizar a classe ResponseEntity para montar a resposta de *not found*;
* O método getOne lança uma *exception* quando o id passado como parâmetro não existir no banco de dados;
* O método findById retorna um objeto Optional<>, que pode ou não conter um objeto.