



Detalhando tópicos

Transcrição

[00:00] Na última aula nós implementamos a parte de validação no cadastro de tópicos e personalizamos o JSON que é devolvido como resposta caso tenha algum erro de validação, usando a classe que funciona como um *Controller Advice* do Spring.

[00:22] Na aula de hoje, nossa API já está com a lógica para fazer a listagem, o cadastro com validação. Agora vamos completar a API. A próxima funcionalidade que vamos colocar é para detalhar um tópico. A ideia é: chega uma requisição para uma URL, passando um tópico específico e eu quero devolver os detalhes apenas desse tópico. Diferente do `lista()`, que vai devolver os detalhes de todos os tópicos.

[00:49] Para fazer essa lógica, vou precisar de um novo método no `Topicos.Controller.java`. Vou criar um novo método, `public`, que devolve um `TopicoDto` - então agora não é mais uma lista, apenas um tópico - e vou chamar esse método de `detalhar()`.

```
public TopicoDto detalhar() {  
  
}
```

[COPIAR CÓDIGO](#)

Preciso colocar qual é o mapeamento - quando esse método vai ser chamado, qual método HTTP. Como estou consultando dados, a requisição vai ser do tipo GET,

```
@GetMapping .
```

```
@GetMapping
public TopicoDto detalhar() {

}
```

[COPIAR CÓDIGO](#)

Só que não posso simplesmente deixar `@GetMapping` , porque o método `lista()` já tem o `@GetMapping` , e ambos usam a URL `"/topicos"` que está declarada em cima da classe. Daria conflito. Eu teria duas URLs `"/tópicos"` com o método GET.

[01:42] Preciso especificar qual tópico quero detalhar. Vou abrir os parênteses, a URL vai ser `"/topicos"` , que é a que está acima da classe, e preciso receber um `{id}` que é dinâmico, então fica `("/{id}")` . Ou seja, para dizer que parte da minha URL é dinâmica, coloco o "id" entre chaves e dou um nome para o parâmetro dinâmico que chamei de `{id}` .

```
@GetMapping("/{id}")
public TopicoDto detalhar() {

}
```

[COPIAR CÓDIGO](#)

[02:08] Obviamente, no meu método `detalhar()` , preciso receber um parâmetro, que é esse `{id}` que virá na barra de endereços. Então, vou receber um `Long id` . Só que se recebermos o parâmetro dessa maneira, o Spring vai achar que ele vem como parâmetro de URL (com interrogação), mas aqui ele vem como parte da URL, ou seja, ele vem como `"/topicos"` seguido de `("/{id})"` .

Isto é, preciso avisar para o Spring que o parâmetro `Long id` não virá numa interrogação e sim no barra ("/"), na URL. Para dizer isso pro Spring, existe uma anotação, `@PathVariable`, que se trata de uma variável do *Path*, da URL. E aí o Spring por padrão perceberá que o nome do parâmetro do método se chama `id` (`@PathVariable Long id`) e a parte dinâmica da URL se chama `id` (`"/{id}"`), então ele vai associar, saberá que é para pegar o que veio na URL e jogar no parâmetro.

```
@GetMapping("/{id}")  
public TopicoDto detalhar(@PathVariable Long id) {  
  
}
```

[COPIAR CÓDIGO](#)

[03:17] 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"`.

```
@GetMapping("/{id}")  
public TopicoDto detalhar(@PathVariable("id") Long codigo) {
```

[COPIAR CÓDIGO](#)

É como se dissesse ao Spring: eu chamei de `codigo`, mas na verdade é para pegar a variável `id` do *Path* e jogar no `codigo`. No nosso caso, não faremos essa alteração, vamos deixar igual, do jeito mais comum, mais simples de utilizar.

[03:50] Agora vem a implementação. Vai chegar a requisição, o `id` do `TopicoDto`, preciso carregar todos os dados do tópico do banco de dados e transformar em um DTO para devolver como resposta. Para carregar do banco de dados, usamos o `topicoRepository`.

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. Vou selecionar `getOne(id)` com o comando "Ctrl + 1" e, em seguida, apertar "Assign statement to new local variable" para que ele guarde isso numa variável que vou chamar de `Topico`. Esse `getOne()` não precisamos declarar no nosso `Repository`, ele já vem da interface que nós herdamos, *JpaRepository*.

```
@GetMapping("/{id}")
public TopicoDto detalhar(@PathVariable Long id) {
    Topico topico = topicoRepository.getOne(id);
}
}
```

[COPIAR CÓDIGO](#)

[04:38] 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);
}
}
```

[COPIAR CÓDIGO](#)

[04:58] Está implementada a lógica para detalhar um tópico. Vamos testar se está tudo funcionando no Postman. Só que agora a requisição será do tipo "GET" e preciso passar o `id` de algum tópico cadastrado no banco de dados. Lembrando que toda vez que ele reinicia, ele cria OS tópicos que deixei no `data.sql`. Ele está sempre

criando três tópicos. Tem os tópicos de `id` 1,2 e 3. Vou colocar o três, por exemplo: <http://localhost:8080/topicos/3> (<http://localhost:8080/topicos/3>). Apertando "Send" disparei a requisição, voltou o código "200 OK", e no corpo da resposta veio o JSON do `TopicoDto.java`.

```
{
  "id": 3
  "titulo": "Dúvida 3",
  "mensagem": "Tag HTML",
  "dataCriacao": "2019-05-05T20:00:00"
}
```

[COPIAR CÓDIGO](#)

[05:40] Só para dar uma variada, vamos voltar ao `TopicosController.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.

[06:18] 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`.

```
@GetMapping("/{id}")
public DetalhesDoTopicoDto detalhar(@PathVariable Long id) {
    Topico topico = topicoRepository.getOne(id);
    return new TopicoDto(topico);
}
```

```
}  
}
```

[COPIAR CÓDIGO](#)

Sendo assim, vou criar outro DTO que é específico para essa lógica de detalhar. Vou criar essa classe no pacote DTO que é um subpacote do `Controller` (pelo atalho, seleciono "Create class 'DetalhesDoTopicoDto'". Na próxima tela, pressiono "Browser" e, em "Choose a folder", seleciono "br.com.alura.forum.controller.dto" seguido de "Ok" para finalizar). Agora, em `DetalhesDoTopicoDto.java`, vou colocar quais campos que esse DTO tem que serializar na hora de gerar o JSON.

[07:00] Vou abrir o `TopicoDto.java` e copiar:

```
private Long id;  
private String titulo;  
private String mensagem;  
private LocalDateTime dataCriacao;
```

[COPIAR CÓDIGO](#)

Agora colarei no `DetalhesDoTopicoDto.java`. Mas, além do `id`, `titulo`, `mensagem` e `dataCriacao`, também quero pegar o nome do usuário que criou esse tópico, então `private String nomeAutor`. E também quero pegar o status do tópico. Como o status é um *enum*, posso devolver diretamente, isto é, `private StatusTopico status`. Lembrando que no DTO só devolvemos coisas primitivas: *string*, *int*, *data* e *enum*. Não estou devolvendo nenhuma entidade da JPA em si.

```
package br.com.alura.forum.controller.dto;  
  
import java.time.LocalDateTime;  
  
import br.com.alura.forum.modelo.StatusTopico;
```

```
public class DetalhesDoTopicoDto {  
  
    private Long id;  
    private String titulo;  
    private String mensagem;  
    private LocalDateTime dataCriacao;  
    private String nomeAutor;  
    private StatusTopico status;  
  
}
```

[COPIAR CÓDIGO](#)

[07:49] 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 `RespostaDto.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.`

```
package br.com.alura.forum.controller.dto;
```

```
import java.time.LocalDateTime;
```

```
public class RespostaDto {  
  
    private Long id;  
    private String mensagem;  
    private LocalDateTime dataCriacao;
```

```
private String nomeAutor;  
  
}
```

[COPIAR CÓDIGO](#)

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.
```

```
package br.com.alura.forum.controller.dto;
```

```
import java.time.LocalDateTime;
```

```
import br.com.alura.forum.modelo.Resposta;
```

```
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();  
    }  
}
```

[COPIAR CÓDIGO](#)

[09:57] 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.

[10:08] Voltando no `DetalhesDoTopicoDto.java`, minha lista não é de `Resposta`, é de `RespostaDto`, isto é, `private List<RespostaDto> respostas;`. Assim como no `TopicoDto.java`, vou receber como parâmetro o `Topico`:

```
public TopicoDto(Topico topico) {  
    this.id = topico.getId();  
    this.titulo = topico.getTitulo();  
    this.mensagem = topico.getMensagem();  
    this.dataCriacao = topico.getDataCriacao();  
}
```

[COPIAR CÓDIGO](#)

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 serializar 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));  
}
```

[COPIAR CÓDIGO](#)

[12:14] Preciso gerar os *Getters* desse meu DTO, vou gerar pelo atalho (Seleciono "Generate Getters and Setters". Na próxima tela, pressiono "Select Getters" com todos os itens selecionados: `dataCriacao`, `id`, `mensagem`, `nomeAutor`, `respostas`, `status` e `titulo`. Para finalizar, basta apertar "Generate"). E aí pronto, tenho outro DTO que tem mais detalhes: a funcionalidade de listagem que usa o `TopicoDto.java` ;o `id`; o `título`; a `mensagem`; a `data de criação`; e o `detalhe do tópico` vem com mais informações, já que estou detalhando um tópico em específico.

[12:37] Vamos voltar ao Postman para testar se vai funcionar. Vou testar primeiro o lista, então, seleciono "GET" no primeiro campo. No segundo campo, o endereço será: <http://localhost:8080/topicos> (<http://localhost:8080/topicos>). Pressiono "Send" e observo que continua devolvendo os três tópicos, 'só vem "id": 1 , titulo": "Dúvida" , "mensagem": "Erro ao criar projeto" e "dataCriacao": "2019-05-05T18:00:00" .

Agora quero detalhar um específico, então, seleciono "GET" no primeiro campo e, no segundo, o endereço será: <http://localhost:8080/topicos/3> (<http://localhost:8080/topicos/3>).

```
{
  "id": 3,
  "titulo": "Dúvida 3",
  "mensagem": "Tag HTML",
  "dataCriacao": "2019-05-05T20:00:00"
}
```

[COPIAR CÓDIGO](#)

Não veio o que estávamos esperando. Vamos voltar no `TopicosController.java` . Substituiremos o `return new TopicoDto(topico)` por `return new DetalhesDoTopicoDto(topico)` , sendo que passei `topico` como parâmetro.

```
@GetMapping("/{id}")
public DetalhesDoTopicoDto detahar(@PathVariable Long id) {
    Topico topico = topicoRepository.getOne(id);
    return new DetalhesDoTopicoDto(topico);
}
```

[COPIAR CÓDIGO](#)

Salvei e vou voltar no Postman para testar de novo. Voltou certinho.

```
{  
  "id": 3,  
  "titulo": "Dúvida 3",  
  "mensagem": "Tag HTML",  
  "dataCriacao": "2019-05-05T20:00:00",  
  "nomeAutor": "Aluno",  
  "status": NAO_RESPONDIDO",  
  "repostas": []  
}
```

[COPIAR CÓDIGO](#)

Como nenhum tópico tem resposta cadastrada no banco de dados, veio vazio o parâmetro do JSON, "repostas":[] . Mas se tivesse resposta, ele iria serializar normalmente. Então funcionou, já temos a funcionalidade de detalhar um tópico e mostramos a vantagem de usar um DTO que é sua flexibilidade: em um endereço, devolvo um DTO, no outro, outro DTO. Isto é, cada DTO com seus campos específicos.

[13:52] No próximo vídeo vamos implementar a próxima funcionalidade, que é para atualizar um tópico. Imagine que cadastrei alguma informação errada e quero atualizar.