02

Cadastrando tópicos

Transcrição

[00:00] Na aula de hoje, vamos aprender um pouco sobre como cadastrar informações no nosso projeto. Até então, só tínhamos implementado a parte da lista, só estou listando todos os tópicos que estão cadastradas no banco de dados. Hoje vamos aprender a fazer o cadastro, como eu recebo informações e como cadastro essas informações no banco de dados.

[00:25] No `TopicosController.java`, só temos um único método, que é o método para listar. Vou precisar de um novo método com a lógica para fazer o cadastro. Vamos criar um novo método, `public`. Ainda não sei o que vou devolver, então vou colocar como `void`, mas depois veremos melhor o que vai acontecer. Vou chamar o método de `cadastrar()`. Esse método vai ter que receber os parâmetros de alguma maneira, mas depois nós vemos o que vai chegar como parâmetro.

```
public void cadastrar() {

}
```

COPIAR CÓDIGO

[01:05] Em todo método preciso ter aquela anotação do `@RequestMapping("/topicos")` para determinar qual é a URL que vai cair nesse método. Na teoria eu poderia copiar `@RequestMapping("/topicos")` e colar em cima do método `cadatrar()`, mas isso daria problema, porque eu teria duas URLs iguais, uma no método `lista()` e outra no método `cadastrar()`. O

Spring vai reclamar dizendo que tem ambiguidade (A URL "/topicos" está mapeada em dois métodos e eu não sei qual dos dois chamar)

```
@RequestMapping("/topicos")
public List<TopicoDto> lista(String nomeCurso) {
    if (nomeCurso == null) {
            List<Topico> topicos =
topicoRepository.findAll();
                return TopicoDto.converter(topicos);
        } else {
            List<Topico> topicos =
topicoRepository.findByCursoNome(nomeCurso);
                return TopicoDto.converter(topicos);
        }
    }

    @RequestMapping("/topicos")
    public void cadastrar() {

    }
}
```

COPIAR CÓDIGO

[01:37] Só que, na verdade, vamos até fazer uma melhoria aqui. Como estou trazendo registros, carregando, a ideia é usar os verbos do HTTP para diferenciar essa operação. Eu poderia dizer para o Spring: o método `lista()` você vai chamar quando a URL for `"/tópicos"` e o método, o verbo HTTP for GET. E, na debaixo, a URL é a mesma, mas você vai chamar quando o método for POST. Se eu usar um método HTTP, consigo diferenciar um do outro.

[02:10] Tem duas maneiras de fazer isso. A primeira é: na anotação `@RequestMapping`, além de ter o valor da URL, existe outro parâmetro chamado `method`, onde posso dizer qual é o método HTTP, se é GET, POST, PUT, DELETE, dentre outros. Então, poderia colocar o atributo `value` e em seguida `, method = RequestMethod.GET`.

```
@RequestMapping("/topicos", method = RequestMethod.GET)
public List<TopicoDto> lista(String nomeCurso) {
//...
}
```

COPIAR CÓDIGO

[02:50] E aí, no método cadastrar eu faço a mesma coisa. Só que aí é POST ao invés de GET, porque estou postando uma informação, fazendo um cadastro. Dessa maneira, não teria mais conflito. O Spring sabe que a URL é a mesma, mas os métodos são diferentes.

```
@RequestMapping("/topicos", method = RequestMethod.POST)
    public void cadastrar() {
    }
```

COPIAR CÓDIGO

[03:05] Mas dá para deixar isso melhor. O problema é que em todos os métodos a URL vai se repetir. Se um dia eu quiser alterar, vou ter que alterar em todos os métodos. Podemos tirar a anotação `@RequestMapping(Value="/topicos", method = RequestMethod.GET)` de cima do método e colocar em cima da classe. Mas aí, na classe, não vou colocar o método, vou colocar só a URL. Então, é como se disséssemos ao Spring: o `TopicosController` responde às aquisições que começam com `"/tópicos"` .

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

import java.util.List;

@RestController
@RequestMapping("/topicos")
public class TopicosController {
//...
}
```

COPIAR CÓDIGO

[03:51] Em cima dos métodos, só preciso dizer então qual o verbo HTTP. Para isso, existe outra anotação do Spring, `@GetMapping`. No método cadastrar seria então `@PostMapping`. E pronto. Ficou mais simples. A URL só é definida uma vez em cima do método.

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

import java.util.List;

@RestController
@RequestMapping("/topicos")
public class TopicosController {

    @Autowired
        private TopicoRepository topicoRepository;

        @GetMapping
        public List<TopicoDto> list<TopicoDto> lista(String
nomeCurso) {
            if (nomeCurso == null) {
                List<Topico> topicos =
topicoRepository.findAll();
                return TopicoDto.converter(topicos);
              } else {
                  List<Topico> topicos =
topicoRepository.findByCursoNome(nomeCurso);
                      return
Topico.Dto.converter(topicos);
                }
            }

            @PostMapping
            public void cadastrar() {

            }
```

```
    }
```

COPIAR CÓDIGO

[04:05] Se eu tiver outro método que tem um complemento da URL, por exemplo, `"/topicos"` ou `"/id"`, posso passar esse complemento no método, e aí não preciso repetir o `"/topicos"`. O que está em cima da classe, `@RequestMapping("/topicos")`, é como se fosse o prefixo. Todos os métodos vão começar com `"/topicos"`. Em cada método eu posso ter um complemento na URL, normalmente.

[04:36] Voltando, o que quero fazer em sequência? Se eu chamar a URL `"/topicos"` via método POST, `@PostMapping`, o Spring já sabe que é para chamar o método `cadastrar()`. Agora preciso, de alguma maneira, receber as informações, receber o novo tópico que quero cadastrar. Na teoria, a ideia seria receber um objeto `topico`, `public void cadastrar (Topico topico)`. O cliente que está fazendo a chamada para a API vai mandar um objeto `topico`, os parâmetros, as informações do `topico`, e eu recebo diretamente o tópico com tudo preenchido.

[05:06] Porém, assim como no método de listagem, `public Lis<TopicoDto> lista(String nomeCurso)`, lembre-se que não era uma boa ficarmos trabalhando com a entidade JPA. 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`.

```
@PostMapping
public void cadastrar(TopicoForm topico) {

}
```

COPIAR CÓDIGO

[05:42] Então, quando eu estiver falando de `TopicoDto`, são dados que saem da API de volta para o cliente. `TopicoForm` são dados que chegam do cliente para a API. `Form` nada mais é que um `Dto`, uma classe que tem só os atributos, *Getters* e *Setters*. Não tem nenhuma entidade, nada do domínio da aplicação.

[06:05] Ele vai reclamar porque preciso criar a classe. Para isso, pressiono "Ctrl + 1", seleciono "Create Class 'TopicoForm'". Dentro do pacote "controller", vou criar um sub pacote chamado "form", isto é, "Package: br.com.alura.forum.controller.form" e, em seguida, apertar "Finish". Com isso, vou jogar a classe `TopicoForm` dentro do `TopicoForm.java`.

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

public class TopicoForm {


}
```

COPIAR CÓDIGO

[06:17] 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 usuário, na teoria, eu deveria receber. Mas a ideia é: o usuário logado que é o autor do tópico. Como ainda não implementamos essa parte de login e de segurança, vamos colocar um autor fixo. Depois vamos alterar isso, quando trabalharmos com segurança.

[07:33] Também temos o curso e a lista de respostas. A lista de respostas é vazia (acabei de criar o tópico, por isso não tenho resposta). Já 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.

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

public class TopicoForm {

    private String titulo;
        private String mensagem;
        private String nomeCurso;

    }
```

COPIAR CÓDIGO

[07:58] São só esses três campos que eu preciso na hora de cadastrar o tópico: `titulo`, `mensagem` e `nomeCurso`. Os outros ou já vem preenchidos por padrão ou vou pegar de alguma maneira. Por exemplo, o caso do usuário que já está logado.

[08:11] Nesta classe, preciso gerar os *Getters* e *Setters*. Vou usar o atalho, pressionar "Generate Getters and Setters...". Na próxima tela, em "Select getters and setters to create", selecionarei tudo ("mensagem", "nomeCurso" e "título") pressionando "Select All", e, em seguida, "Generate". Agora basta formatar o código e temos nosso `TopicoForm`. Uma classe Java, um POJO, um Java Bean, que não tem anotação, dependência com nada, é só atributo *Getter* e *Setter*. F

no `TopicosController.java` é justamente o `TopicoForm` que recebo como parâmetro.

[08:32] Mas se deixarmos o parâmetro como `TopicoForm topico` no método, `cadastrar(TopicoForm topico)`, vai ser parecido com o parâmetro do `lista(String NomeCurso)`. No método `lista()` não colocamos para filtrar pelo nome do curso, `NomeCurso`? Dessa maneira, ele vê como parâmetro da URL. Já os parâmetros `TopicoForm topico`, na hora de cadastrar, eles não vêm na barra de endereços, não vêm na URL. Eles vêm no corpo da requisição.

A requisição é via método POST, não método GET. Então eu não mando os parâmetros via URL. Os parâmetros vão no corpo da requisição. Preciso avisar isso para o Spring. Temos que colocar uma anotação no parâmetro `TopicoForm topico`, que é o `@RequestBody`. É como se disséssemos ao Spring: Esse parâmetro - esse `TopicoForm` - é para pegar do corpo da requisição, e não das URLs, como parâmetro de URL.

```
@PostMapping
public void cadastrar(@RequestBody TopicoForm topico) {

}
```

COPIAR CÓDIGO

[09:23] Vai chegar o objeto `TopicoForm` com os dados preenchidos. Agora já posso implementar minha lógica do método `cadastrar()`. Eu preciso pegar esse `TopicoForm` e salvar no banco de dados. Para acessar o banco de dados, lembre-se que estamos usando o nosso `topicoRepository`. Inclusive, se eu chamar o `topicoRepository.save(topico)`, perceberemos que `save()` é um método pronto, porque se trata de operação comum nos `Repositories`. Então, posso chamar o `save()` e passar, como parâmetro, o `topico`. Mas se você olhar, você verá que não está compilando, porque o `save()` do `topicoRepository` está esperando um objeto `topico`, que é a nossa entidade Mas, o `topico` que temos não é a entidade, é um `form`.

```
@PostMapping
public void cadastrar(@RequestBody TopicoForm form) {

    topicoRepository.save(topico);
        }
```

COPIAR CÓDIGO

[10:18] Vamos ter um problema. Nós recebemos o `form` - classe que só tem os dados que estão vindo da requisição - mas na hora de salvar,não vou salvar um objeto `form` , preciso do objeto `topico` , da entidade. Vou ter que converter, da mesma maneira que convertemos na `lista()` , quando pegamos a lista de tópicos e convertemos para `topicoDto` . Aqui vou ter que fazer o caminho contrário: Vou receber o `form` , e tenho que converter para um `topico` .

[10:46] 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` .

```
@PostMapping
public void cadastrar(@RequestBody TopicoForm form) {
    Topico topico = form.converter();
        topicoRepository.save(topico);
 }
```

COPIAR CÓDIGO

[11:15] Agora, pressionando o comando "Ctrl + 1", Vou pedir para o Eclipse gerar para mim o método `converter()` ("Create method 'converter()' in type 'TopicoForm'). Terminado, vou retornar escrevendo `return new` e, em seguida `Topico()` , isto é, instancio o objeto `Topico` . Só que se eu der `new` dessa maneira, esse `Topico` não tem nenhum campo preenchido, vou precisar setar tudo na mão.

```
public Topico converter() {
    return new Topico();
}
```

COPIAR CÓDIGO

[11:33] 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.

[11:57] Então vou usar o atalho, gerar um construtor pressionando "Generate Getters and Setters". Na próxima tela, selecionaremos - para indicar o que esse construtor recebe - o "titulo", a "mensagem", e o "curso". Em seguida, pressionaremos "Generate".

```
public Topico(String titulo, String mensagem, Curso curso) {
        this.titulo = titulo;
        this.mensagem = mensagem;
        this.curso = curso;
}
```

COPIAR CÓDIGO

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.

[12:42] 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.

```
public Topico converter(TopicoRepository repository) {

    return new Topico(titulo, mensagem, curso);
}
```

COPIAR CÓDIGO

[13:07] 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`.

```
public Topico converter(CursoRepository cursoRepository) {

    return new Topico(título, mensagem, curso);
}
```

COPIAR CÓDIGO

[13:22] Mas nós não temos um `CursoRepository` até então. Só tínhamos o `TopicoRepository`. Eu vou criar um novo abrindo o atalho, pressionando "Create class 'CursoRepository'". Na próxima tela, vou trocar "br.com.alura.forum.controller.form" para "br.com.alura.forum.repository" e apertar "Finish", criando, a partir de agora, o `CursoRepository.java`.

Vou abrir nosso `TopicoRepository.java`, lembrando que temos que herdar o `extends JpaRepository<Topico, Long>` passando o *generic* (a entidade e o tipo da chave primária). Vou copiar, trocando a entidade para `Curso`. E pronto. Acabei de criar um `CursoRepository.java`, parecido com o nosso `TopicoRepository.java`.

[14:05] Agora, o que eu preciso fazer? 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);
}
```

COPIAR CÓDIGO

Lembre-se, o `findByNome`, como é específico do nosso projeto, não existe como método. Vou ter que pedir para o Eclipse criar para mim, acessando o atalho e selecionando "'Created method 'findByNome(String)' in type 'CursoRepository'" e ele criará o `findByNome`.

```
package br.com.alura.forum.repository;

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

public interface CursoRepository extends
JpaRepository<Curso, Long> {

    Curso findByNome(String nomeCurso);

}
```

COPIAR CÓDIGO

[14:58] Estou usando o padrão de nomenclatura do Spring Data, não preciso montar a Query, pois ele já sabe que é para filtrar pelo atributo `nomeCurso`. Eu vou importar o `Curso`, em seguida tenho o `findByNome`, passei o `nomeCurso`, ele devolveu o objeto `curso` completo. E aí sim eu pego e passo esse objeto na hora de criar o tópico. Agora, no meu `TopicoForm.java`, no meu método

`converter()`, já devolvo um objeto `Topico(titulo, mensagem, curso)` completo, com título, mensagem e curso que carreguei pelo `Repository`.

```
public Topico converter(CursoRepository cursoRepository) {
    Curso curso = cursoRepository.findByNome(nomeCurso);
        return new Topico(titulo, mensagem, curso);
}
```

COPIAR CÓDIGO

[15:22] Voltando para o `TopicoController.java`, vai dar erro de compilação, porque preciso passar como parâmetro um `CursoRepository`. Só que não tenho, então vou injetar. Para isso, vou declarar mais um atributo, e aí ao invés de `TopicoRepository` será `CursoRepository`. Vou renomear também a variável para `CursoRepository` e importá-la.

```
@Autowired
private TopicoRepository topicoRepository;

@Autowired
private CursoRepository cursoRepository;
//...
```

COPIAR CÓDIGO

[15:45] Desse jeito fica um pouco mais simples. Eu chamo o `form`, chamo o `converter`, passo as informações que não tem, que o `form` não consegue recuperar, mas que aqui no `Controller` eu tenho e ele já cria o objeto `Topico`, que mando gravar no banco de dados. Com isso, já temos a lógica de `cadastrar()` funcionando.

```
@PostMapping
public void cadastrar(@RequestBody TopicoForm form) {
    Topico topico = form.converter(cursoRepository);
```

```
        topicoRepository.save(topico);
    }
```

COPIAR CÓDIGO

[16:05] Só que o método ainda está `void`. O Spring até aceita, ele vai devolver - se der tudo certo - o código 200. Se der alguma *exception* no meio do caminho, ele devolve 500 para o cliente, como resposta. Mas no próximo vídeo vamos aprender que não deveríamos devolver 200 nesse cenário. Vamos dar uma melhorada para seguir as boas práticas do mundo REST.