07

Utilizando Repository

Transcrição

[00:00] Na última aula, nós configuramos a JPA do nosso projeto, mapeamos as entidades, configuramos o banco de dados H2 e fizemos um acesso pelo console do H2 (disponibilizado pelo Spring Boot). Ficou faltando a parte da listagem no `TopicosController.java`, que ainda estamos devolvendo uma lista em memória. Nesta aula vamos trocar essa lista para trazer os registros do banco de dados de verdade, lá do H2.

[00:33] Para fazer isso, 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.

[01:18] 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. Mas, se você parar para pensar, todas as classes DAO são muito parecidas. Pelo menos nos métodos do "*CRUD*", Cadastro, Listagem, Exclusão e Alteração, é sempre igual (faz a consulta, o insert, o update e o delete). A única coisa que muda é a entidade (por exemplo, em uma DAO estou trabalhando com o "tópico", na outra com "resposta", na outra com "curso"), mas o código é sempre igual.

[01:57] 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.

[02:35] Para implementar esse sistema, no nosso pacote principal "br.com.alura.forum", vou criar uma nova interface selecionando "Interface" e depois o comando "Ctrl + N". Na próxima tela, seleciono "Intarface" e "Next". Agora, vou só trocar o pacote, para colocar essa interface dentro do pacote chamado `Repository`, para não ficar misturado. Então, "Package: br.com.alura.forum.repository". E o nome da interface vai ser "TopicoRepository".

Como no `Topico.Repository.java` temos uma interface e não uma classe, não preciso colocar nenhuma anotação em cima dela. Normalmente, as classes que são gerenciadas pelo Spring, temos que colocar um `@controller`, `@service`, `@Repository`, `@Component`. Esse, por ser interface, não precisa. O Spring já encontra a classe automaticamente.

[03:22] Essa interface, eu preciso herdar de alguma interface do Spring data. O Spring data tem algumas interfaces que você pode utilizar na herança. No nosso caso, vamos herdar de uma interface chamada `JpaRepository`. Quando você herda dessa interface, percebe que ela tem um *generics* que você tem que passar dois tipos. O primeiro é a entidade com que o `JpaRepository` vai trabalhar (no nosso caso é `Topico`). E o segundo é qual o tipo do atributo do `ID`, da chave primária dessa entidade. No nosso caso, estamos usando o `Long`.

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

import
```

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

import br.com.alura.forum.modelo.Topico;

public interface TopicoRepository extends
JpaRepository<Topico, Long> {

}
```

COPIAR CÓDIGO

[04:10] Só precisamos criar essa interface herdando `JpaRepository`, os *generics*, e ela fica vazia. Na verdade, 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.

[04:41] Agora, no meu `Topicos.Controller.java`, preciso injetar o `JpaRepository` que acabei de criar. Para fazer a injeção, uso o mesmo esquema do Spring `@Autowired`. Você declara um atributo, que no nosso caso vai ser do tipo `topicoRepository`, que é nossa interface, e aí vou chamar de `topicoRepository` o atributo. Vou só importar o `@Autowired` e o `TopicoRepository`. Está pronta a injeção de dependências como em qualquer outra classe.

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

import java.util.Arrays;

@RestController
public class TopicosController {

    @Autowired
        private TopicoRepository topicoRepository;
```

```
//...
}
```

COPIAR CÓDIGO

[05:11] No método `lista()`, vou apagar a linha que estava criando um tópico em memória.

```
Topico topico = new Topico("Dúvida", "Dúvida com Spring",
new Curso("Spring", "Programação"));
```

COPIAR CÓDIGO

E apagar a linha que criava uma `Arrays.aslist(topico, topico, topico)`.Na hora de chamar o `TopicoDto.converter()`, preciso passar uma lista chamada `topicos`, vai dar erro de compilação porque essa variável não existe, por isso, vou vou criar na linha de cima `List<Topico> topicos = topicoRepository.findAll()`.

Isto é, `List<Topico>` e essa lista vem de `topicos = topicoRepository.` porque vou usar o `Repository` que foi injetado. Seguindo, vou chamar o método `findAll()`, que é o método que faz uma consulta carregando todos os registros do banco de dados e é justamente o que quero neste momento. Agora, vou só importar a classe `Topico` e tudo continua compilando normalmente.

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

import java.util.List;

@RestController
public class TopicosController {

    @Autowired
        private TopicoRepository topicoRepository;

        @RequestMapping("/topicos")
```

```
        public List<TopicoDto> lista() {
            List<Topico> topicos =
topicoRepository.findAll();
                return TopicoDto.converter(topicos);
        }
    }
```

COPIAR CÓDIGO

[06:17] Percebe como é simples? Eu só injeto o `topicoRepository` e chamo o método `findAll()`, que nem fomos nós que escrevemos - se olharmos lá no nosso `TopicoRepository.java`, está vazio, porque ele já veio herdado do `JpaRepository`. Essa é uma comodidade para quem trabalha com JPA, não precisar criar mais as classes seguindo o padrão DAO. Usa o `Repository` e deixa o seu código muito menor, muito mais simples e fácil de fazer manutenção.

[06:38] Vamos testar, ver se ele vai carregar os registros do nosso banco de dados (importante lembrar que já criamos um arquivo `data.sql`, que popula o banco com alguns registros) . Vou acessar a URL http://localhost:8080/topicos (http://localhost:8080/topicos). Vai dar um probleminha:

"Caused by: org hibernate InstantiationException: No default constructor for entity: : br.com.alura.forum.modelo.Topico"

A JPA exige que todas as entidades tenham o construtor *default*, que não recebe parâmetros. Ele está reclamando que nossa classe `Topico` não tem o construtor *default*. Vamos dar uma olhada na classe `Topico`.

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

COPIAR CÓDIGO

[07:19] Não sei se vocês lembram, mas quando estávamos criando aquele tópico em memória, eu tinha criado o construtor só para facilitar, para, na hora de dar `new`, já passar as informações. Não preciso mais. Já que estou carregando do banco, não preciso mais o construtor com parâmetros. Se eu apagar, o Java vai gerar o construtor *default* sem nenhum parâmetro.

Nós também fizemos isso na entidade cursos: criamos um construtor que já recebia o nome e a categoria. Não precisamos mais disso, porque não estamos criando o curso estático no `Controller`. Então, vamos apagar, porque ele vai ter o construtor *default*.

```
public Curso(String nome, String categoria) {
    this.nome = nome;
        this.categoria = categoria;
}
```

COPIAR CÓDIGO

[07:56] Voltando para o navegador, na URL http://localhost:8080/topicos (http://localhost:8080/topicos), e atualizando a página (com o comando "F5"), veremos que agora tudo funciona certinho. Ele está carregando os registros do banco de dados (o primeiro é "id", "Dúvida" e a "mensagem"; o segundo e o terceiro, se você olhar aquele arquivo `data.sql`, é exatamente aqueles três registros que eu populei - que o Spring lê e popula o banco de dados). Com isso, conseguimos acessar o banco de dados de verdade no `Topicos.Controller.java`, usando o padrão `TopicoRepository`, que faz o acesso ao banco de dados via JPA usando o banco de dados H2 que está em memória.

[08:34] Essa era a aula de hoje. Espero vocês no próximo vídeo para completarmos nossa API, onde vamos fazer uma consulta usando filtros, porque aqui estamos fazendo `findAll()`, ou seja, estou carregando todos os registros.