

CETEJ35 - Java Web - JAVA_XXX (2024_01)

[Meus cursos](#) / [CETEJ35 - Web \(2024_01\)](#) / [Semana 06: 07/10 a 13/10](#) / [Integração](#)

Integração

✓ **Feito:** Ver

A fazer: Gastar pelo menos 20 minutos na atividade

A fazer: Passar pela atividade até o fim

Fecha: segunda-feira, 2 dez. 2024, 00:00

Normalmente, um aplicativo real vai precisar mais do que salvar dados em memória - ele vai precisar de persistência em banco! É aí que entra o conteúdo dessa aula.

PERSISTÊNCIA

Usar JPA para fazer a persistência em um banco relacional com o Spring Boot não é difícil. Primeiro, você precisa adicionar as dependências do sistema gerenciador de banco de dados (SGBD) e do JPA. Segundo, você precisa de uma entidade persistente que defina os campos que serão usados para salvar os dados no banco de dados. Em seguida, você precisa criar uma interface que define as operações da sua entidade, como criar e listar. Depois, você usa os recursos do Spring Boot para integrar a persistência com o controlador da aplicação. Por fim, precisa remover o código antigo de persistência em lista e fazer pequenos ajustes. Vamos explorar cada passo.



Essa aula assume o conhecimento de persistência de dados com Java e JPA como pré-requisito.

Expanda os itens abaixo para entender os detalhes de implementação.

▼ 1. Adicionando dependências

Como você já sabe, o Java permite acesso a uma ampla gama de SGBDs por meio de drivers específicos. Esses [drivers fornecem implementações](#) do JDBC para o SGBD específico. Apesar de usarmos o JPA para fazer a persistência, esses drivers são essenciais para que o próprio JPA funcione. Por isso, precisamos começar inserindo a dependência que adiciona o driver do SGBD no nosso projeto.

Por simplicidade, vamos usar o [H2](#). O H2 é um banco de dados que pode funcionar tanto em memória como com persistência em disco. Ele é simples, rápido e não necessita de configuração. Além disso, o Spring Boot fornece autoconfiguração para o banco. Tudo que precisamos fazer é adicionar a dependência no **pom.xml**.

Você poderia inserir um banco MySQL com a mesma **simplicidade** com que estamos fazendo aqui. A diferença é que seria necessário informar os dados para acesso ao banco, além de precisar configurar o banco de dados, é claro.

Abra o **pom.xml** e insira as linhas 46-50 da Figura abaixo logo após a última dependência. Isso indica ao Maven que o H2 será usado. O Maven se encarrega de baixar a dependência localmente e deixa-la disponível para o Spring Boot. O Spring Boot se encarrega do resto. Adicione também a dependência do [Spring Data JPA](#) (linhas 52 a 55), que é o projeto do Spring que carrega, além do JPA, várias outras funcionalidades para facilitar a persistência de dados.

```
41     <dependency>
42         <groupId>org.springframework.boot</groupId>
43         <artifactId>spring-boot-starter-validation</artifactId>
44     </dependency>
45
46     <dependency>
47         <groupId>com.h2database</groupId>
48         <artifactId>h2</artifactId>
49         <scope>runtime</scope>
50     </dependency>
51
52     <dependency>
53         <groupId>org.springframework.boot</groupId>
54         <artifactId>spring-boot-starter-data-jpa</artifactId>
55     </dependency>
56 </dependencies>
```

▼ 2. Criando entidade persistente

A entidade persistente é uma classe Java simples (POJO), que segue as [especificações do JPA](#): implementar a interface `java.io.Serializable`, possuir atributos não transientes e não finais, possuir métodos de acesso e um construtor sem argumentos. Cada classe representa uma tabela no banco de dados, enquanto os atributos representam os dados que serão armazenados nas tabelas.

Normalmente, as entidades recebem o nome dos objetos no mundo real que elas representam. Por exemplo, **Produto** representando um produto que é vendido, ou **Cliente**, representando os clientes que são atendidos. Atualmente, nosso código já tem uma classe chamada `visao.Cidade`. Por isso, vamos criar uma nova classe chamada `visao.CidadeEntidade`, conforme mostra a Figura a seguir.



```
6 public class CidadeEntidade implements Serializable{
7
8     private Long id;
9     private String nome;
10    private String estado;
11
12    public String getEstado() {
13        return estado;
14    }
15
16    public Long getId() {
17        return id;
18    }
19
20    public String getNome() {
21        return nome;
22    }
23
24    public void setEstado(String estado) {
25        this.estado = estado;
26    }
27
28    public void setId(Long id) {
29        this.id = id;
30    }
31
32    public void setNome(String nome) {
33        this.nome = nome;
34    }
35 }
```



O Java criar um construtor sem argumentos se nenhum for definido. Além do **nome** e do **estado**, que são atributos essenciais para definir a **Cidade**, precisamos também de um **id**. Observe que o **id** é um requisito do banco de dados, uma vez que precisamos ter uma chave primária. Um bom padrão é definir o **id** como tipo **Long**, que armazena muito mais valores do que um **int**. Além disso, usamos um objeto em vez do tipo primitivo porque o Spring Boot precisa de um objeto **id** quando for fazer as operações de persistência.

O próximo passo é inserir as anotações. Para essa classe, só precisamos de três anotações. A anotação **jakarta.persistence.Entity** define que essa classe será uma classe persistente. O parâmetro **name** define o nome da tabela no banco de dados que será usada para persistir os dados dessa classe. A anotação **jakarta.persistence.Id** define o atributo **id** como a chave primária da tabela. Por fim, a anotação **jakarta.persistence.GeneratedValue** define que essa chave primária vai usar a estratégia de auto-incremento do SGBD para gerar a chave. Os demais atributos não precisam de mapeamento porque eles representam valores primitivos e são mapeados automaticamente pelo JPA.

```
10 @Entity (name = "cidade")
11 public class CidadeEntidade implements Serializable{
12
13     @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Long id;
15     private String nome;
16     private String estado;
```

▼ 3. Criando Interface de Persistência

Precisamos de uma classe que implemente as operações de persistência e execute essas operações usando o driver do SGBD. O Spring Data facilita esse processo fornecendo uma interface que já define as operações mais comuns, como as operações CRUD. Durante a compilação, o próprio Spring Data se encarrega de gerar uma implementação da interface que usa as operações sobre as entidades que você definiu.

Para isso, precisamos criar uma interface que estenda a interface `org.springframework.data.jpa.repository.JpaRepository`. Ao estender essa interface, precisamos definir qual é a entidade que a interface vai gerenciar, e qual é o tipo de dados da chave primária. Crie um arquivo chamado `visao.CidadeRepository.java` e coloque o conteúdo da Figura abaixo.

Adicionar "Repository" ao final da classe que define as operações de persistência é um padrão comum no ecossistema Java/Spring Boot.

```
1 package br.edu.utfpr.cp.esjava.crudcidades.visao;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5 public interface CidadeRepository extends JpaRepository<CidadeEntidade, Long> { }
```

▼ 4. Integrando `CidadeRepository` ao controlador

Agora podemos usar o `CidadeRepository` para realizar a persistência de dados no banco. Para fazer isso, abra o `visao.CidadeController` e adicione um atributo final que faz referencia para interface `CidadeRepository`. Em seguida, altere o método construtor para que ele receba o `CidadeRepository` como parâmetro. Dentro do construtor, faça com que o atributo receba o parâmetro. O resultado final pode ser visto na Figura a seguir.

```
15 @Controller
16 public class CidadeController {
17
18     private Set<Cidade> cidades;
19
20     private final CidadeRepository repository;
21
22     public CidadeController(CidadeRepository repository) {
23
24         cidades = new HashSet<>();
25         this.repository = repository;
26     }
```

▼ 5. Removendo persistência em lista e adicionando persistência em banco

Vamos começar pelo método `listar()`. Em vez de acessar a lista de cidades, vamos usar o atributo `repository` para acessar as cidades armazenadas no banco. O método `org.springframework.data.jpa.repository.JpaRepository.findAll()` retorna uma lista do tipo `CidadeEntidade`. Porém, observe que nosso MVC está usando `Cidade` como referência para apresentar dados na tela, e não `CidadeEntidade`. Por isso, precisamos converter os dados.

Não se preocupe com isso agora. Ainda nessa aula vamos falar sobre o padrão DTO que permite resolver essa questão.

```
26 @GetMapping("/")
27 public String listar(Model memoria) {
28
29     memoria.addAttribute("listaCidades", repository
30                                     .findAll()
31                                     .stream()
32                                     .map(cidade -> new Cidade(
33                                         cidade.getNome(),
34                                         cidade.getEstado()))
35                                     .collect(Collectors.toList()));
36
37     return "/crud";
38 }
```

Ao executar o código, a lista na página Web deve aparecer vazia, uma vez que não temos dados no nosso banco ainda. Vamos alterar agora o método `criar()` para inserir os dados.



Em vez de inserir a cidade informada na lista de cidades, vamos usar o atributo `repository` para persistir os dados da cidade no banco. Não podemos esquecer que estamos usando a `Cidade` no MVC, mas `CidadeEntidade` para a persistência. Por isso, precisamos converter os dados. Depois da conversão, basta chamar o método `org.springframework.data.jpa.repository.JpaRepository.save()` para persistir os dados em `CidadeEntidade`. Veja como ficou o código.

```
62     } else {
63         var novaCidade = new CidadeEntidade();
64         novaCidade.setNome(cidade.getNome());
65         novaCidade.setEstado(cidade.getEstado());
66
67         repository.save(novaCidade);
68     }
69
70     return "redirect:/";
71 }
```

Observe que a persistência só acontece se passar pela validação.

O método `excluir()` demanda apenas alguns ajustes. Vamos abrir o `CidadeRepository` e adicionar um método que permite localizar por `nome` e `estado`. Para fazer isso, basta usar as [especificações padrão do Spring Data](#). Dessa forma, teremos o método `findByNomeAndEstado` em `visao.CidadeRepository`. Esse método retorna uma `CidadeEntidade`, e recebe dois parâmetros representando o `nome` e o `estado` informados na página Web.

Se você está se perguntando: "Onde está o corpo do método?". Lembre-se que Spring Data implementa isso pra você. Tudo que você precisa fazer é seguir as especificações.

```
7 public interface CidadeRepository extends JpaRepository<CidadeEntidade, Long> {  
8  
9     public Optional<CidadeEntidade> findByNomeAndEstado(String nome, String estado);  
10 }
```

Por que **Optional**? Porque não sabemos se a cidade existe! O **Optional** nos permite fazer validações antes da exclusão.

Agora podemos retornar ao **CidadeController**, buscar tentar recuperar a **CidadeEntidade** de acordo com o nome e estado informados na página Web e, se tudo der certo, remover a entidade do banco. Veja como ficou o código do método **excluir()**.

```
73 @GetMapping("/excluir")  
74 public String excluir(  
75     @RequestParam String nome,  
76     @RequestParam String estado) {  
77  
78  
79     var cidadeEstadoEncontrada = repository.findByNomeAndEstado(nome, estado);  
80  
81     cidadeEstadoEncontrada.ifPresent(repository::delete);  
82  
83     return "redirect:/";  
84 }
```

Para a operação de alteração, precisamos ajustar tanto o método **preparaAlterar()** quando o método **alterar()**. Vamos começar pelo **preparaAlterar()**.

```
85 @GetMapping("/preparaAlterar")
86 public String preparaAlterar(
87     @RequestParam String nome,
88     @RequestParam String estado,
89     Model memoria) {
90
91     var cidadeAtual = cidades
92         .stream()
93         .filter(cidade ->
94             cidade.getNome().equals(nome) &&
95             cidade.getEstado().equals(estado))
96         .findAny();
97
98     if (cidadeAtual.isPresent()) {
99         memoria.addAttribute("cidadeAtual", cidadeAtual.get());
100         memoria.addAttribute("listaCidades", cidades);
101     }
102
103     return "/crud";
104 }
```

Atualmente, o método `preparaAlterar()` usa um filtro para buscar uma cidade pelo `nome` e `estado` (linhas 91 a 96). Podemos usar o método de busca que definimos no `CidadeRepository` para substituir o filtro atual. Em seguida, o método atual grava a cidade atual e a lista de cidades na memória da solicitação caso a cidade buscada exista (linhas 98 a 101). Podemos substituir a estrutura condicional pelo método `Optional.ifPresent()`, assim como fizemos no método `excluir()`. Os valores que são gravados na solicitação para serem usados na página Web serão buscados do banco de dados vez da lista. O resultado pode ser visto no código abaixo.



```
85 @GetMapping("/preparaAlterar")
86 public String preparaAlterar(
87     @RequestParam String nome,
88     @RequestParam String estado,
89     Model memoria) {
90
91     var cidadeAtual = repository.findByNomeAndEstado(nome, estado);
92
93     cidadeAtual.ifPresent(cidadeEncontrada -> {
94         memoria.addAttribute("cidadeAtual", cidadeEncontrada);
95         memoria.addAttribute("listaCidades", repository.findAll());
96     });
97
98     return "/crud";
99 }
```

O método `alterar()` precisa de dois ajustes. Primeiro, precisamos substituir a exclusão da cidade existente da lista pela busca no banco (linhas 109 a 111). Isso porque agora, em vez de excluir e criar novamente uma cidade, vamos alterar os dados da cidade existente. Em seguida, vamos substituir a chamada do método `criar()`, pelo método apropriado do `repository` para alterar a cidade existente.

```
101     @PostMapping("/alterar")
102     public String alterar(
103         @RequestParam String nomeAtual,
104         @RequestParam String estadoAtual,
105         Cidade cidade,
106         BindingResult validacao,
107         Model memoria) {
108
109         cidades.removeIf(cidadeAtual ->
110             cidadeAtual.getNome().equals(nomeAtual) &&
111             cidadeAtual.getEstado().equals(estadoAtual));
112
113         criar(cidade, validacao, memoria);
114
115         return "redirect:/";
116     }
117 }
```

Veja como ficou o código após a alteração. Usamos o mesmo método usado no método `preparaAlterar()` e `excluir()` para buscar a cidade existente. Se a cidade atual é encontrada no banco de dados, então alteramos seus dados e atualizamos no banco (linhas 109 a 116).

Observe que também excluímos os parâmetros que não seriam mais usados com a substituição do método `criar()` pelo procedimento de alteração.



```
101     @PostMapping("/alterar")
102     public String alterar(
103         @RequestParam String nomeAtual,
104         @RequestParam String estadoAtual,
105         Cidade cidade) {
106
107         var cidadeAtual = repository.findByNomeAndEstado(nomeAtual, estadoAtual);
108
109         if (cidadeAtual.isPresent()) {
110
111             var cidadeEncontrada = cidadeAtual.get();
112             cidadeEncontrada.setNome(cidade.getNome());
113             cidadeEncontrada.setEstado(cidade.getEstado());
114
115             repository.saveAndFlush(cidadeEncontrada);
116         }
117
118         return "redirect:/";
119     }
120 }
```

Observe que poderíamos usar o `id` como identificador da cidade atual. Assim, toda a busca seria feita pelo `id`. Mas, para isso, precisaríamos fazer ajustes incluindo a página Web. Para manter a compatibilidade com o código anterior, preferimos manter o nome e o estado como identificadores.

Agora que substituímos a lista pelo banco de dados, podemos remover as referencias à lista. Isso inclui: atributo, chamada do atributo no construtor, uso da lista no método `criar()`, `imports` desnecessários.

O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana05-10-integracao-persistencia`.


Retroceder

Avançar

◀ Verificação de aprendizado - Validação de Dados

Seguir para...

API Reativa ▶

✉ Contate o suporte do site 



Você acessou como RAFAEL ROCHA DA SILVA PROENCA (Sair)
CETEJ35 - Web (2024_01)

- Tema
- Adaptable
- Boost
- Clássico
- Campus
- Apucarana
- Campo Mourão
- Cornélio Procopio
- Curitiba
- Dois Vizinhos
- Francisco Beltrão
- Guarapuava
- Londrina
- Medianeira
- Pato Branco
- Ponta Grossa
- Reitoria
- Santa Helena
- Toledo
- UTFPR
- Ajuda
- Chat UTFPR
- Calendário Acadêmico
- Biblioteca
- e-Mail
- Nuvem (OwnCloud)
- Produção Acadêmica
- Secretaria Acadêmica
- Sistemas Corporativos
- Sistema Eletrônico de Informação - SEI
- Suporte ao usuário
- Criação de curso
- Comunidade
- Português - Brasil (pt_br)
- Deutsch (de)
- English (en)
- Português - Brasil (pt_br)



Resumo de retenção de dados

Baixar o aplicativo móvel.

 Dê um feedback sobre este software 

Universidade Tecnológica Federal do Paraná - UTFPR

Suporte ao usuário

