



CRIANDO, ALTERANDO E EXCLUINDO

Um CRUD completo com Java e Spring Boot

Visão Geral

Ajustes

Criar

Renderização

Excluir

Alterar

Verificação de Aprendizado

VISÃO GERAL

Nesta aula nós finalizamos a implementação das quatro operações CRUD. Isso significa que nosso usuário será capaz de criar, alterar, excluir e listar as cidades em uma base de dados. Observe que ainda estamos usando uma base local, baseada em uma lista em memória. Nós vamos evoluir esse projeto até integrarmos essa base com um banco de dados.

Para a aula fluir melhor, nós começamos ajustando o código anterior. Em seguida, damos início à operação de criação. Essa é operação mais simples, por isso começamos por ela. Nós esclarecemos o modelo de renderização usado no desenvolvimento Web com Java. Isso ajuda a explicar alguns comportamentos nas operações seguintes.

Após a operação de exclusão, damos início à operação de alteração. A operação de alteração é a mais complicada de todas, por isso deixamos por último. Você também vai perceber que essa seção ficou grande. Isso porque vários passos são necessários para que a alteração seja realizada. Acompanhe essa seção com atenção redobrada.

Se você tem pressa, o vídeo abaixo mostra de forma resumida tudo o que foi feito nessa aula. As demais Seções explicam em detalhes cada parte do processo.

É importante ressaltar que essa aula apresenta os princípios fundamentais para qualquer aplicação Web com Java.

AJUSTANDO O CÓDIGO

Durante o desenvolvimento desse projeto, vamos fazer várias mudanças no código. Tradicionalmente, precisaríamos parar a execução do aplicativo e iniciar novamente para ver cada alteração. Isso é trabalhoso e pouco produtivo. Por isso, o Spring Boot fornece um recurso chamado *Hot Swapping*. Esse recurso reiniça a aplicação cada vez que o código for salvo. Para ativar o recurso, basta inserir a dependência `spring-boot-devtools`, conforme mostra a Figura abaixo.

```
35
36      <dependency>
37          <groupId>org.springframework.boot</groupId>
38          <artifactId>spring-boot-devtools</artifactId>
39      </dependency>
```

Nesse primeiro momento, vamos continuar usando um objeto `Set` para armazenar a lista de cidades. Contudo, o código atual mantém a lista como uma variável local. Para que a lista fique acessível para os outros métodos que vamos criar, vamos transformar a lista em um atributo da classe, conforme mostra a Figura a seguir.

```
9  @Controller
10 public class CidadeController {
11
12     private Set<Cidade> cidades;
13 }
```

No código anterior, a lista era iniciada usando uma fábrica por meio do método `Set.of`. Esse método cria uma lista imutável. Agora, precisamos de uma lista mutável, pois vamos inserir e remover cidades da lista. Para isso, vamos iniciar uma lista fazendo uso do construtor da classe `CidadeController`. Com isso, podemos remover a variável cidades do método listar. Veja como ficou o código completo na Figura abaixo.

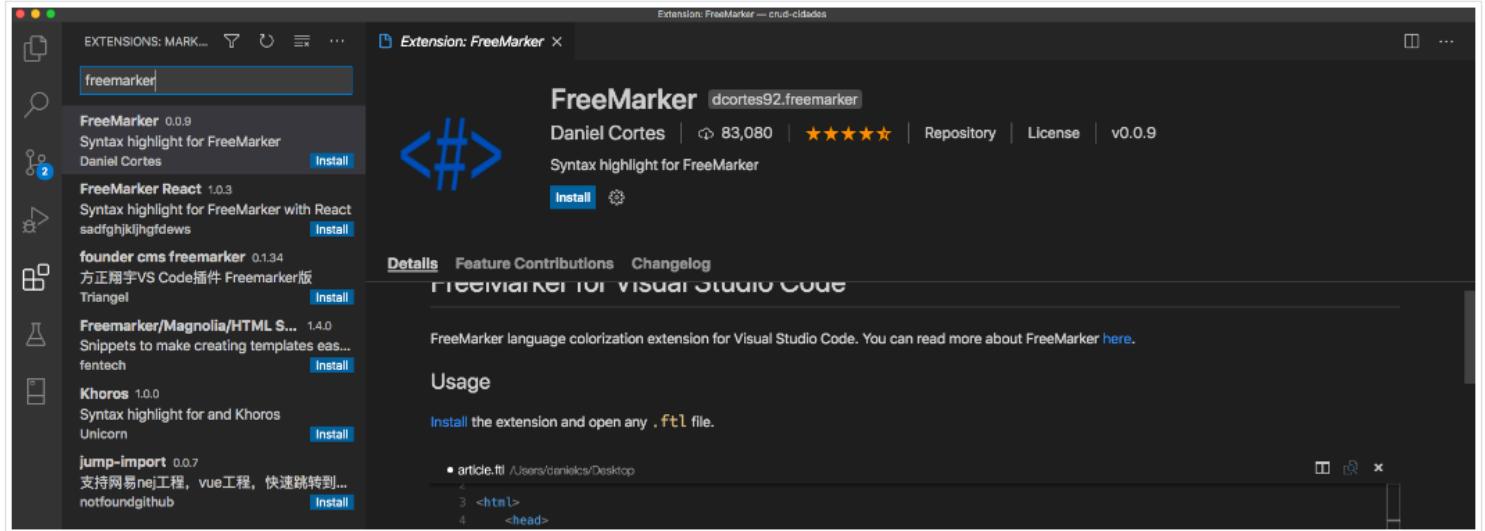
```
10 @Controller
11 public class CidadeController {
12
13     private Set<Cidade> cidades;
14
15     public CidadeController() {
16         cidades = new HashSet<>();
17     }
18
19     @GetMapping("/")
20     public String listar(Model memoria) {
21
22         memoria.addAttribute("listaCidades", cidades);
23
24         return "/crud";
25     }
26 }
```

O `HashSet` é uma das implementações do `Set`, assim como o `ArrayList` é uma das implementações do `List`. Tanto o `List` quanto o `Set` fazem parte do *Collections Framework* do Java.

Se você está usando o VS Code, deve ter percebido que perdemos a formatação e o destaque do código quando alteramos a extensão da página Web de `crud.html` para `crud.ftlh`. Isso porque o VS Code não reconhece a extensão `.ftlh` por padrão. Por isso, vamos usar um plugin para o Freemarker, produzido por Daniel Cortes.

Se você está usando uma IDE ou outro editor, procure saber como é o suporte à páginas Freemarker na sua IDE/editor.

Para encontrar esse plugin, basta ir na aba de extensões do VS Code e procurar por *Freemarker*. Clique no botão instalar, e o plugin já estará disponível.



Contudo, se você abrir a página `crud.ftlh`, vai perceber que nada aconteceu. Isso porque o plugin, por padrão, espera por páginas com extensão `.ftl`, que é o padrão do Freemarker. Mas o Spring Boot usa um outro padrão, o `.ftlh`. Não se preocupe, isso é uma chance para usarmos o mecanismo de flexibilização do Spring Boot: o arquivo `resources/application.properties`.

O Spring Boot é altamente flexível, e o arquivo `resources/application.properties` tem um papel fundamental nessa flexibilidade. Por meio desse arquivo é possível fazer ajustes como o que precisamos nesse momento. Para que o Spring Boot reconheça uma página com a extensão `.ftl`, basta adicionar a linha de código abaixo direto no arquivo `resources/application.properties`.

```
spring.freemarker.suffix=.ftl
```

Salve o arquivo e, em seguida, altere a extensão do arquivo `crud.ftlh` para `crud.ftl`. Pronto, agora temos o melhor dos dois mundos: *Syntax Highlithing* e o Spring Boot funcionando!

Ao executar o projeto você vai perceber que não tem mais uma lista de cidades na tabela. Isso porque inicializamos a lista vazia. Nas próximas seções vamos implementar os demais métodos do CRUD e popular a tabela..

O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana04-10-crud-ajustes`.

CRIANDO

Para criar uma nova cidade, vamos começar alterando nossa página Web (*View*) para que ela envie dados da cidade (*Model*) para o método `criar()`, na classe `CidadeController` (*Controller*).

Abra o arquivo `crud.ftl` e adicione as seguintes as propriedades conforme indicado na Figura abaixo.

```
20 | <form action="/criar" method="POST">
```

A propriedade `action` indica para onde os dados devem ser enviados, enquanto a propriedade `method` indica o método HTTP que deve ser usado para o envio de dados. POST costuma ser o método mais adequado para envio de dados de formulários.

O próximo passo é adicionar a propriedade `name` nas tags `input`, que são usadas para inserir o nome da cidade e do estado. A propriedade `name` mapeia os valores digitados na tag `input` com os atributos na classe `Cidade` (*Model*). Para que esse mapeamento seja feito automaticamente, é importante que os valores usados na propriedade `name` seja igual ao nome dos atributos na classe `Cidade`. Veja na Figura a seguir como ficou o código do formulário após as alterações.

```
crud.ftl
src > main > resources > templates > crud.ftl
20 | <form action="/criar" method="POST">
21 |   <div class="form-group">
22 |     <label for="nome">Cidade:</label>
23 |     <input name="nome" type="text"
24 |       class="form-control" placeholder="Informe o nome
25 |       da cidade" id="nome">
26 |   </div>
27 |   <div class="form-group">
28 |     <label for="estado">Estado:</label>
29 |     <input name="estado" type="text"
30 |       class="form-control" placeholder="Informe o
31 |       estado ao qual a cidade pertence"
32 |       id="estado">
33 |   </div>
34 |   <button type="submit" class="btn btn-primary">CRIAR</
35 |   button>
36 | </form>
37 | <table class="table table-striped table-hover mt-5">
38 |   <thead class="thead-dark">
```

```
Cidade.java
src > main > java > br > edu > utpr > cp > espjava > visao > Cidade.java > ...
1 package br.edu.utpr.cp.espjava.crudcidades.visao;
2
3 public final class Cidade {
4     private final String nome;
5     private final String estado;
6
7     public Cidade(final String nome, final String estado) {
8         this.nome = nome;
9         this.estado = estado;
10    }
11
12    public String getEstado() {
13        return estado;
14    }
15
16    public String getNome() {
17        return nome;
18    }
19 }
```

Agora, vamos adicionar o método responsável por receber os dados do formulário e adicionar a nova cidade na lista. Para isso, abra a classe `CidadeController` e crie um novo método, que vamos chamar de `criar()`.

Esse método deve receber um objeto do tipo `Cidade` como parâmetro. Assim como o método `listar`, `criar` retorna uma `String`, representando a página que deve ser carregada. Nesse caso, o que queremos é que o usuário seja enviado para a página `crud.ftlh`. Mas também queremos que a lista de cidades seja carregada e mostrada na tabela.

Por isso, em vez de retornar o nome da página, vamos retornar um `redirect:/`, que representa um redirecionamento para o método `listar()`. O método `listar()`, por sua vez, carrega a lista de cidades e chama a página `crud.ftlh`.

Por fim, precisamos adicionar a anotação que vai mapear o endereço no formulário com o método `criar()`. Diferentemente do método `listar()`, o método `criar()` vai receber uma anotação `org.springframework.web.bind.annotation.PostMapping("/criar")`. A URL na anotação deve ser a mesma usada na propriedade `action`, do formulário na página `crud.ftlh`. Também observe que estamos usando um `@PostMapping` em vez de `@GetMapping`. Isso porque o formulário envia os dados por meio do método `POST`, do protocolo HTTP.

```

1 CidadeController.java x
src > main > java > br > edu > utfpr > cp > espjava > crudcidades > visao
27
28     @PostMapping("/criar")
29     public String criar(Cidade cidade) {
30
31         cidades.add(cidade);
32
33         return "redirect:/";
34     }
35 
```

```

1 crud.ftl x
src > main > resources > templates > crud.ftl
20     <form action="/criar" method="POST">
21         <div class="form-group">
22             <label for="nome_cidade">Nome:</label>
23             <input name="nome" type="text" class="form-control" placeholder="Informe o nome da cidade" id="nome">
24         </div>
25         <div class="form-group">
26             <label for="estado">Estado:</label>
27             <input name="estado" type="text" class="form-control" placeholder="Informe o estado da cidade" id="estado">

```

Veja como ficou o código completo do *Controller* na Figura abaixo.

```

11 @Controller
12 public class CidadeController {
13
14     private Set<Cidade> cidades;
15
16     public CidadeController() {
17         cidades = new HashSet<>();
18     }
19
20     @GetMapping("/")
21     public String listar(Model memoria) {
22
23         memoria.addAttribute("listaCidades", cidades);
24
25         return "/crud";
26     }
27
28     @PostMapping("/criar")
29     public String criar(Cidade cidade) {
30
31         cidades.add(cidade);
32
33         return "redirect:/";
34     }
35 } 
```

Neste ponto, você já pode executar o projeto e tentar inserir uma nova cidade.



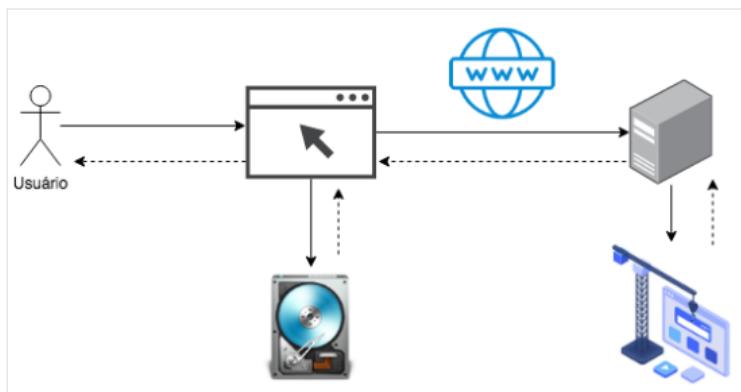
Não esqueça que estamos usando uma lista em memória para guardar os valores. Por isso, quando o aplicativo é reiniciado, os dados são perdidos. Muito em breve vamos adicionar persistência no código e armazenar os dados em banco.

O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana04-20-crud-criar`.

RENDERIZAÇÃO

Quando um usuário digita uma URL na barra de endereços do navegador, o navegador busca uma página HTML no servidor referenciado pelo endereço digitado. Então, o navegador faz o download da página HTML e salva ela localmente, no HD do usuário. Por fim, o navegador abre a página, lê o conteúdo e renderiza na tela o conteúdo HTML. Se houver CSS, o CSS é interpretado e o conteúdo é formatado. Se houver JavaScript, o código é executado quando o navegador o encontrar durante a renderização.

Aplicações Web com Java seguem exatamente esse mesmo modelo. A diferença é que as páginas dinâmicas são *montadas* no servidor. *Montadas* significa que o framework preenche as variáveis com valores e coloca esses valores direto na página. Quando o navegador faz o download da página, o que ele recebe é uma página estática, como qualquer outra.



Por isso, toda informação referente à uma solicitação é perdida quando a solicitação alcança o destino. Quando o navegador envia uma solicitação para o servidor informando dados de um formulário, por exemplo, os dados são perdidos assim que o servidor os recebe.

A mesma coisa acontece no caminho oposto. Quando o servidor retorna a página com dados anexados à uma solicitação, como um token de autenticação, esse valor é descartado logo após a página ter sido entregue.

Existem formas de manter essas informações. Pode-se usar *sessões* ou *cookies*. Vamos falar sobre isso em uma aula futura.

Por enquanto, é importante que você entenda que os valores que estão armazenados no `org.springframework.ui.Model` são perdidos assim que a solicitação atinge o destino. Por exemplo, quando o `org.springframework.ui.Model` carrega a lista de cidades para serem preenchidas na tabela. Ou quando a classe `Cidade` é usada para enviar os dados do formulário até o controlador.

EXCLUINDO

Para excluir uma cidade, tudo que precisamos é remover a `Cidade` correspondente da lista cidades. Para isso, vamos começar alterando o botão `EXCLUIR` na página `crud.ftl` para que ele envie uma solicitação para o `CidadeController`.

Assim, vamos adicionar o atributo `href` à `tag a`, usada para criar o botão. Vamos enviar a solicitação para a URL `/excluir`. Adicionalmente, precisamos informar qual cidade queremos excluir. Nesse caso, sabemos que, teoricamente, o nome é único dentro de um estado. Por isso, vamos enviar o nome e o estado como parâmetros na requisição. A Figura abaixo mostra como ficou o código do botão.

Na verdade, na forma como está o código, não tem qualquer validação durante a inserção. Portanto, ele vai aceitar valores vazios, cidades repetidas, etc. Teremos uma aula específica sobre validação, na qual iremos resolver isso.

```

40 |         <tbody>
41 |             <#list listaCidades as cidade >
42 |                 <tr>
43 |                     <td>${cidade.nome}</td>
44 |                     <td>${cidade.estado}</td>
45 |                     <td>
46 |                         <div class="d-flex d-justify-content-center">
47 |                             <a class="btn btn-warning mr-3">ALTERAR</a>
48 |                             <a href="/excluir?nome=${cidade.nome}&estado=${cidade.estado}" class="btn btn-danger">EXCLUIR</a>
49 |                         </div>
50 |                     </td>
51 |                 </tr>
52 |             </#list>
53 |         </tbody>
```

A linha 48 da Figura acima mostra que estamos enviando dois parâmetros, `nome` e `estado`, para a URL `/excluir`. Os valores desses parâmetros são dinâmicos, uma vez que os valores da tabela são gerados automaticamente (Veja linha 41). Portanto, tudo que precisamos fazer é extrair o nome e o estado da cidade atual no `loop`, e apresentar esses valores usando a sintaxe de interpolação do Freemarker (`${ }`).

Agora, na classe `CidadeController`, precisamos criar um método que receba esses parâmetros e que também esteja mapeado com a URL `/excluir`. Para isso, vamos criar um método chamado `excluir()`, que retorna uma `String` e recebe duas `String` como parâmetro: `nome` e `estado`.

Vamos adicionar aos parâmetros a anotação `org.springframework.web.bind.annotation.RequestParam`. Essa anotação mapeia os parâmetros enviados pelo formulário com os parâmetros do método `excluir()`. Observe o nome dos parâmetros no formulário e no método `excluir` são iguais. Isso é necessário para que o Spring Boot faça o mapeamento automático entre eles.

```

38 |     public String excluir(
39 |         @RequestParam String nome,
40 |         @RequestParam String estado) {
```

Dentro do método, vamos usar o método `removeIf()` do *Collections Framework* para remover a `Cidade` da lista. Para isso, vamos usar o atributo `cidades`, do tipo `Set`. O método `removeIf()` usa um predicado (Lambda) como parâmetro. Para iterar pela lista, vamos usar uma variável chamada `cidadeAtual`. Tudo que precisamos fazer é verificar se o `nome` e `estado` da `cidadeAtual` é igual ao `nome` e `estado` passado como parâmetro para o método `excluir()`.

Não esqueça que comparação de `String` em Java é feita pelo método `equals()`.

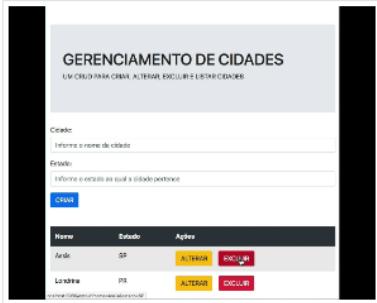
```

42 |         cidades.removeIf(cidadeAtual ->
43 |             cidadeAtual.getNome().equals(nome) &&
44 |             cidadeAtual.getEstado().equals(estado));
```

Agora, vamos redirecionar o resultado do método para o método `listar()`, que vai carregar a página e mostrar os dados atualizados. Por fim, adicionamos a anotação que mapeia a URL enviada pelo botão `EXCLUIR`, na página `crud.ftl`, com o método `excluir()`, na `CidadeController`. Veja como ficou o código completo do método `excluir()`.

```
37     @GetMapping("/excluir")
38     public String excluir(
39         @RequestParam String nome,
40         @RequestParam String estado) {
41
42         cidades.removeIf(cidadeAtual ->
43             cidadeAtual.getNome().equals(nome) &&
44             cidadeAtual.getEstado().equals(estado));
45
46         return "redirect:/";
47     }
```

Você já pode executar o projeto, inserir, listar e excluir cidades. Na próxima Seção vamos criar o último método: alterar.



O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana04-30-crud-excluir`.

ALTERANDO

Deixamos a alteração por último porque ela é a mais complexa das quatro operações. Primeiro, é necessário carregar os dados na tela. Segundo, a tela precisa ser adaptada para alterar em vez de criar. Em seguida, os dados precisam ser enviados para que o método no controlador encontre os dados na base, recupere esses dados, faça a atualização e recarregue a página com os novos dados. Vamos fazer um passo de cada vez nesta seção.

1. Carregando dados na tela

Vamos começar inserindo comportamento no botão **ALTERAR**, na página **crud.ftl**. Para isso, siga o mesmo procedimento usado no botão **EXCLUIR**, conforme pode ser observado na linha 47, na Figura a seguir.

```

41 |          <#list listaCidades as cidade >
42 |          <tr>
43 |              <td>${cidade.nome}</td>
44 |              <td>${cidade.estado}</td>
45 |              <td>
46 |                  <div class="d-flex d-justify-content-center">
47 |                      <a href="/preparaAlterar?nome=${cidade.nome}&estado=${cidade.estado}" class="btn btn-warning mr-3">ALTERAR</a>
48 |                      <a href="/excluir?nome=${cidade.nome}&estado=${cidade.estado}" class="btn btn-danger">EXCLUIR</a>
49 |
50 |                  </div>
51 |              </td>
52 |          </tr>
      </#list>
```

A propriedade **href** identifica para onde a solicitação será enviada. Nesse caso, estamos considerando uma URL **/preparaAlterar**. Essa mesma URL precisa ser usada quando fizermos o mapeamento do método no controlador. Assim como no botão **EXCLUIR**, precisamos enviar o nome e estado para que a cidade seja encontrada. Ambos valores são enviados como parâmetros. Esses valores são extraídos da lista de cidades usando a sintaxe de interpolação do Freemarker (**\${}**).

Em seguida, precisamos criar o método que será responsável por tratar essa solicitação na classe **CidadeController**. Abra a classe e crie um método chamado **preparaAlterar()**. Assim como no método **excluir()**, o método **preparaAlterar** recebe dois parâmetros, representando o nome da cidade e do estado.

O método **preparaAlterar()** deve recuperar a cidade cujo nome e estado foram informados pelo parâmetro, colocar esses valores recuperados na memória da solicitação e recarregar a página. Para colocar os valores na memória da solicitação, vamos precisar de mais um parâmetro do tipo **org.springframework.ui.Model**. Veja como ficou a assinatura do método **preparaAlterar()**.

```

49 |     public String preparaAlterar(
50 |         @RequestParam String nome,
51 |         @RequestParam String estado,
52 |         Model memoria) {
```

Agora, o método precisa recuperar os dados e coloca-los na memoria. Vamos usar o atributo **cidades** para ter acesso à lista de cidades. Vamos filtrar apenas a cidade cujo nome e estado é igual aos valores passados como parâmetro. Como vamos ter apenas uma combinação de cidade/estado, então podemos usar o método **findAny()** para recuperar um objeto **java.util.Optional**, que encapsula a cidade buscada. O código da busca fica conforme a Figura abaixo.

Nesse curso fazemos uso massivo dos recursos introduzidos desde o Java 8. Se você sente dificuldade em usar esses recursos, dê uma olhada em XXX

```
55     var cidadeAtual = cidades
56         .stream()
57         .filter(cidade ->
58             cidade.getNome().equals(nome) &&
59             cidade.getEstado().equals(estado))
60         .findAny();
```

Se existir uma cidade com os valores buscados, ela será armazenada em um objeto do tipo `Optional`, acessível pela variável `cidadeAtual`. O último passo aqui é verificar se o `cidadeAtual` não está vazia. Nesse caso, precisamos adicionar a `cidadeAtual` na memória da solicitação.

Teoricamente, sempre existirá uma cidade com os valores buscados porque a URL é montada de acordo com os dados na tabela. Contudo, alguém pode tentar acessar o aplicativo digitando a URL diretamente na barra de endereços do navegador. Para evitar problemas, faz sentido usar a validação.

```
55     var cidadeAtual = cidades
56         .stream()
57         .filter(cidade ->
58             cidade.getNome().equals(nome) &&
59             cidade.getEstado().equals(estado))
60         .findAny();
61
62     if (cidadeAtual.isPresent()) {
63         memoria.addAttribute("cidadeAtual", cidadeAtual.get());
64         memoria.addAttribute("listaCidades", cidades);
65     }
```

Observe que a linha 64 também adiciona a lista de cidades na memória. Precisamos disso porque esse método não fará um redirecionamento. Esse método vai carregar a página diretamente. Por isso, se quisermos ver a lista de cidades, precisamos coloca-la na memoria da solicitação. O último passo no controlador é retornar o nome da página que será carregada e adicionar a anotação que faz o mapeamento com a URL de alteração. Veja como ficou o código completo.

```
49     @GetMapping("/preparaAlterar")
50     public String preparaAlterar(
51         @RequestParam String nome,
52         @RequestParam String estado,
53         Model memoria) {
54
55         var cidadeAtual = cidades
56             .stream()
57             .filter(cidade ->
58                 cidade.getNome().equals(nome) &&
59                 cidade.getEstado().equals(estado))
60             .findAny();
61
62         if (cidadeAtual.isPresent()) {
63             memoria.addAttribute("cidadeAtual", cidadeAtual.get());
64             memoria.addAttribute("listaCidades", cidades);
65         }
66
67         return "/crud";
68     }
```

Por que não redirecionar? Por que cada solicitação tem um tempo de vida, lembra? Ela só existe em um caminho: o de ida, ou de volta. Nesse caso, se fizermos um redirecionamento, o valor de `cidadeAtual` será perdido.

O último passo para carregar os dados na tela é atualizar a página `crud.ftl` para que ela exiba os dados de `cidadeAtual` se ela existir na memória. Para isso, vamos usar o atributo `value` da `input` para definir os dados do campo. Também usamos uma sintaxe especial do Freemarker que só exibe valores caso eles existam. Veja como ficou o código dos nomes da cidade e do estado na página `crud.ftl`.

```
21      <div class="form-group">
22          <label for="nome">Cidade:</label>
23          <input value="${cidadeAtual.nome}!" name="nome" type="text" class="form-control"
24              placeholder="Informe o nome da cidade" id="nome">
25      </div>
26      <div class="form-group">
27          <label for="estado">Estado:</label>
28          <input value="${cidadeAtual.estado}!" name="estado" type="text" class="form-control"
29              placeholder="Informe o estado ao qual a cidade pertence" id="estado">
30      </div>
```

Nesse ponto, você já pode executar o projeto, criar uma cidade e solicitar a alteração clicando no botão `ALTERAR`. Você verá a página inicial sendo recarregada com os valores da cidade que você escolheu. Agora, precisamos alterar e enviar os novos valores para alteração.

2. Adaptando tela para alterar em vez de criar

Se você tentou fazer um teste após a alterar os dados e clicou no botão `CRIAR`, vai perceber que uma nova cidade foi criada com os dados que você inseriu. Não é bem isso que esperamos. Na verdade, o que queremos é alterar os dados e não inserir uma nova cidade. Por isso, precisamos que, no momento em que os dados são carregados, o formulário também seja atualizado para alterar em vez de criar.

Para isso, vamos começar usando a diretiva `<if>/<else>` do Freemarker. Essas diretivas permitem tomar decisões com base em algum valor, da mesma forma que um `if/else` em Java.

Duas partes do formulário precisam ser alteradas. A primeira é a parte que define para onde os dados serão enviados. A segunda é o rótulo do botão. Para alterar a URL vamos criar uma estrutura condicional que verifica se a variável `cidadeAtual` existe na memória da solicitação. Se existir, então estamos *alterando* a cidade. Senão, significa que estamos *criando* uma nova cidade.

Contudo, para que alteração seja bem sucedida, também precisamos enviar os valores atuais. Os valores atuais são necessários para o controlador localize os dados na lista de cidades e faça a alteração. Nesse ponto, talvez você esteja pensando: "Já não fizemos isso?". Sim, fizemos, mas isso foi feito para recuperar os valores e mostra-los na página. Agora, vamos efetivamente realizar a alteração. Também, é importante lembrar que os dados que estão na memória da solicitação são resetados toda vez que uma nova solicitação é feita.

Para enviar os dados atuais, vamos usar um atributo escondido no formulário. Esse atributo usa os valores de `cidadeAtual` para identificar qual cidade deve ser alterada. Veja como ficou o código de envio do formulário com essas alterações.

```
20
21      <if cidadeAtual??>
22          <form action="/alterar" method="POST">
23              <input type="hidden" name="nomeAtual" value="${cidadeAtual.nome}!"/>
24              <input type="hidden" name="estadoAtual" value="${cidadeAtual.estado}!"/>
25      <else>
26          <form action="/criar" method="POST">
27      </if>
28
```

A segunda parte do formulário que precisa ser alterada é o botão que envia o formulário. Basicamente, o que muda é o rótulo. A estratégia adotada é a mesma usada para alterar o envio do formulário.

```
39 |     <#if cidadeAtual??>
40 |         <button type="submit" class="btn btn-warning">CONCLUIR ALTERAÇÃO</button>
41 |     <#else>
42 |         <button type="submit" class="btn btn-primary">CRIAR</button>
43 |     </#if>
```

3. Enviando dados para controlador

Esse procedimento envolve criar um novo método na classe `CidadeController`, mapear esse método com a URL usada na página `crud.ftl` para concluir a alteração, e alterar os dados na lista de cidades.

Vamos criar um método chamado `alterar()` na classe `CidadeController`. Esse método deve receber três parâmetros. Os dois primeiros representam os nomes atuais da cidade e do estado que devem ser alterados. O terceiro valor representa os novos valores, que são recebidos da mesma forma como acontece no método `criar()`. Também precisamos mapear o método com a URL de alteração enviada pelo `crud.ftl`.

```
1 CidadeController.java
2 crud.ftl
```

```
70 @PostMapping("/alterar")
71 public String alterar(
72     @RequestParam String nomeAtual,
73     @RequestParam String estadoAtual,
74     Cidade cidade) {
75
76     cidades.removeIf(cidadeAtual ->
77         cidadeAtual.getNome().equals(nomeAtual) &&
78         cidadeAtual.getEstado().equals(estadoAtual));
79
80     criar(cidade);
81
82     return "redirect:/";
83 }
```

```
21 <#if cidadeAtual??>
22     <form action="/alterar" method="POST">
23         <input type="hidden" name="nomeAtual" value="${cidadeAtual.nome}" />
24         <input type="hidden" name="estadoAtual" value="${cidadeAtual.estado}" />
25     </form>
26 </#if>
```

O corpo do método é simples. Primeiro, precisamos remover a cidade atual da mesma forma que fizemos no método `excluir()`. Por que remover? Porque cada cidade é imutável, lembra? Não conseguimos alterar os dados de uma cidade existente. Por isso, vamos remover e inserir uma nova cidade com os dados novos.

Observe que podemos reutilizar o método `criar()`. Isso porque o método `criar` simplesmente insere novos valores e redireciona para a página inicial. Isso é exatamente o que queremos. Veja como ficou o código completo do método `alterar()`.

```
70 @PostMapping("/alterar")
71 public String alterar(
72     @RequestParam String nomeAtual,
73     @RequestParam String estadoAtual,
74     Cidade cidade) {
75
76     cidades.removeIf(cidadeAtual ->
77         cidadeAtual.getNome().equals(nomeAtual) &&
78         cidadeAtual.getEstado().equals(estadoAtual));
79
80     criar(cidade);
81
82     return "redirect:/";
83 }
```

Veja que o retorno do método `alterar()` nunca é usado. Isso porque o método `criar()` já retorna a solicitação. Ainda assim, o retorno é necessário para evitar erros de compilação.

GERENCIAMENTO DE CIDADES

UM CRUD PARA CRIAR, ALTERAR, EXCLUIR E LISTAR CIDADES

Cidade:

Londrina

Estado:

PR

CONCLUIR ALTERAÇÃO

Nome	Estado	Ações
Londrina	PR	ALTERAR EXCLUIR
Assis	SP	ALTERAR EXCLUIR

localhost:8080/preparaAlterar?name=Londrina&estado=PR

O código desenvolvido nesta Seção está disponível no **Github**, na branch **semana04-40-crud-alterar**.