



VALIDAÇÃO DE DADOS

Usando o *Bean Validation Framework* para validar entrada de dados

[Visão Geral](#)[Entrada de Dados](#)[Bean Validation](#)[Método criar\(\)](#)[Exibindo Erros](#)[Mantendo Valores](#)[Páginas de Erro](#)[Verificação de Aprendizado](#)

VISÃO GERAL

Nosso CRUD de cidades funciona - desde que todos os dados inseridos estejam corretos!

Como não conseguimos garantir isso o tempo todo, precisamos de meios para validar a entrada de dados. É aí que entra o *Bean Validation Framework*. Ele é um recurso fundamental para garantir que as entradas de dados estejam de acordo com o esperado.

Nesta aula, vamos ver como usar recursos do *Bean Validation Framework* juntamente com o Spring Boot, Freemarker e Bootstrap para garantir que o usuário consiga visualizar os erros e corrigir os dados sempre que necessário.

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.

ENTRADA DE DADOS

Nesse momento, nosso CRUD funciona perfeitamente – pelo menos até alguém inserir um valor inesperado. Por exemplo, sabemos que não existem cidades com o mesmo nome dentro de um estado. Isso é uma regra de negócio observada pelo estado Brasileiro. Contudo, da forma como está agora, nosso CRUD permite não só duas cidades com o mesmo nome dentro de um estado, como também cidades com nomes vazios ou números no lugar de nome. Você já consegue imaginar o problema que isso causaria em um aplicativo real, não é mesmo? Por isso, precisamos de validações na entrada de dados.

The screenshot shows a web application titled "GERENCIAMENTO DE CIDADES" (City Management) with the subtitle "UM CRUD PARA CRIAR, ALTERAR, EXCLUIR E LISTAR CIDADES". The interface includes fields for "Cidade" (Londrina) and "Estado" (PR), and a "CRIAR" button. Below the form is a table listing a single city entry: Londrina, PR, with "ALTERAR" and "EXCLUIR" buttons. The "Estado" field has a dropdown menu open, showing "PR" as the selected value.

A primeira validação que precisamos é a exigência de algumas informações. Por exemplo, o nome da cidade e do estado deve ser sempre informado. Isso pode ser resolvido facilmente usando recurso do próprio HTML. A propriedade `required` pode ser usada para forçar o usuário a informar os valores.

GERENCIAMENTO DE CIDADES

UM CRUD PARA CRIAR, ALTERAR, EXCLUIR E LISTAR CIDADES

Cidade:

Informe o nome da cidade

Please fill out this field.

Informe o estado ao qual a cidade pertence

CRIAR

Nome

Estado

Ações

Para isso, basta adicionar a propriedade required às tags input, na página **crud.ftl**. Veja na Figura abaixo.

```
29      <div class="form-group">
30          <label for="nome">Cidade:</label>
31          <input required value="${cidadeAtual.nome}!" name="nome" type="text" class="form-control" placeholder="Informe o nome da
32              cidade" id="nome">
33      </div>
34      <div class="form-group">
35          <label for="estado">Estado:</label>
36          <input required value="${cidadeAtual.estado}!" name="estado" type="text" class="form-control" placeholder="Informe o
37              estado ao qual a cidade pertence"
            id="estado">
        </div>
```

Outra validação que pode ser feita direto na página é a quantidade de caracteres. A propriedade **maxlength** pode ser usada para definir o valor máximo de caracteres. Por exemplo, não queremos um estado com mais de 2 caracteres.

No nosso exemplo, estamos assumindo que o estado é na, na verdade, a sigla do estado, como 'PR' ou 'SC'.

```
33      <div class="form-group">
34          <label for="estado">Estado:</label>
35          <input maxlength="2" required value="${cidadeAtual.estado}!" name="estado" type="text" class="form-control"
36              placeholder="Informe o estado ao qual a cidade pertence"
            id="estado">
        </div>
```

Contudo, observe que o formulário não emite qualquer aviso. O usuário é apenas impedido de digitar mais do que 2 caracteres. Também, observe que mesmo usando **required**, o usuário pode digitar espaço em branco no nome e ele será aceito. Por isso, precisamos de mais validação do que o HTML oferece.

O código desenvolvido nesta Seção está disponível no **Github**, na branch **semana05-10-validacao-html**.

BEAN VALIDATION FRAMEWORK

O [Bean Validation Framework](#) é uma especificação que define como validações podem ser implementadas para validar atributos em uma classe Java. O *Bean Validation Framework* fornece um [conjunto de anotações](#) que, quando empregado nos atributos, define restrições e mensagens de alerta.

Para usar o *Bean Validation Framework*, tudo que precisamos é inserir a dependência `spring-boot-starter-validation` no `pom.xml`.

```
36   <dependency>
37     <groupId>org.springframework.boot</groupId>
38     <artifactId>spring-boot-devtools</artifactId>
39   </dependency>
40
41   <dependency>
42     <groupId>org.springframework.boot</groupId>
43     <artifactId>spring-boot-starter-validation</artifactId>
44   </dependency>
45 </dependencies>
```

O [conjunto de validações é abrangente](#), mas vamos nos concentrar apenas naqueles mais importantes para o problema que temos no momento. Para inserir as validações, abra a classe `Cidade`, que define os atributos `nome` e `estado`. Imediatamente antes dos atributos, vamos usar algumas anotações para definir restrições, conforme mostra a Figura abaixo.

```
6  public final class Cidade {
7
8    @NotBlank(message = "Nome da cidade deve ser informado")
9    @Size(min = 5, max = 60, message = "O nome da cidade deve ter entre 5 e 60 caracteres")
10   private final String nome;
11
12   @NotBlank(message = "Sigla do estado deve ser informado")
13   @Size(min = 2, max = 2, message = "A sigla do estado está limitada a dois caracteres")
14   private final String estado;
```

A anotação `javax.validation.constraints.NotBlank` impede que sejam aceitos valores nulos ou espaços em branco, enquanto a anotação `javax.validation.constraints.Size` restringe um limite inferior e superior de caracteres. Ambas as anotações podem usar o atributo `message`, que define a mensagem que deve ser exibida caso a restrição não seja atendida.

Apesar de ser simples manter a mensagem direto na classe, isso não costuma ser uma boa ideia. Primeiro, qualquer ajuste na mensagem exige recompilação.

Segundo, isso dificulta a internacionalização da aplicação – afinal, a mensagem deve estar na mesma língua do usuário. Por fim, a alteração na mensagem pode causar problemas no código caso o desenvolvedor acabe alterando algo por engano.

Por isso, uma boa prática é colocar as mensagens em um arquivo separado. Para fazer isso, precisamos de um pouco de configuração. Abra a classe `br.edu.utfpr.cp.espjava.crudcidades.CrudCidadesApplication` e insira os métodos `messageSource()` e `getValidator()`, conforme mostra a Figura a seguir.

```
12  @SpringBootApplication
13  public class CrudCidadesApplication {
14
15      public static void main(String[] args) {
16          SpringApplication.run(CrudCidadesApplication.class, args);
17      }
18
19      @Bean
20      public MessageSource messageSource() {
21          ReloadableResourceBundleMessageSource messageSource = new ReloadableResourceBundleMessageSource();
22          messageSource.setBasename("classpath:messages");
23          messageSource.setDefaultEncoding("UTF-8");
24          return messageSource;
25      }
26
27      @Bean
28      public Validator getValidator() {
29          LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();
30          bean.setValidationMessageSource(messageSource());
31          return bean;
32      }
33 }
```

O método `messageSource()` define o nome do arquivo (linha 22) e o tipo de codificação (linha 23) que será usada no arquivo onde são armazenadas as mensagens. Esse método é anotado com `org.springframework.context.annotation.Bean`, sinalizando que o Spring Boot deve gerenciar esse método. O método `getValidator()` registra o arquivo de mensagens para ser usado juntamente com as validações.

Agora, precisamos transferir as mensagens para um arquivo próprio e mapear as validações com as mensagens. Por padrão, o arquivo é `resources/message.properties`. O arquivo tem um formato de par de valores. Do lado esquerdo é definida uma chave e, do lado direito, um valor. A chave é usada pelas anotações de validação na classe para associar a mensagem que deve ser usada. O resultado por ser visto na próxima Figura.

The screenshot shows two code editor panes. The left pane displays the `messages.properties` file with the following content:

```
app.cidade.blank=Nome da cidade deve ser informado
app.cidade.size=Nome da cidade deve ter entre 5 e 60 caracteres
app.estado.blank=Sigla do estado deve ser informado
app.estado.size=A sigla do estado está limitada a dois caracteres
```

The right pane displays the `Cidade.java` class with the following code:

```
public final class Cidade {
    public final String nome;
    @NotBlank(message = "{app.cidade.blank}")
    @Size(min = 5, max = 60, message = "{app.cidade.size}")
    private final String nome;
    public final String estado;
    @NotBlank(message = "{app.estado.blank}")
    @Size(min = 2, max = 2, message = "{app.estado.size}")
    private final String estado;
}
```

Annotations from the `messages.properties` file are mapped to the validation constraints in the `Cidade.java` class. Specifically, the key `app.cidade.blank` maps to `@NotBlank`, and the key `app.estado.size` maps to `@Size(min = 2, max = 2)`.

Observe que você tem liberdade para definir o nome da chave que será usada.

O código desenvolvido nesta Seção está disponível no **Github**, na branch `semana05–20–validacao–avax`.

VALIDANDO MÉTODO `criar()`

Apesar de termos inserido as anotações de validação, isso não é o suficiente para a validação acontecer de forma automática. Precisamos indicar aos métodos que eles precisam usar a validação quando receberem a classe `Cidade` como parâmetro de uma solicitação. Para fazer isso, adicione a anotação `javax.validation.Valid` antes do parâmetro `cidade`, na classe `CidadeController`.

```
32  @PostMapping("/criar")
33  public String criar(@Valid Cidade cidade) {
```

Isso fará com que o Spring Boot acione a validação no método `criar()` quando ele receber um objeto do tipo `Cidade`. A validação será realizada de acordo com as anotações na classe `Cidade`. Portanto, a nulidade e tamanho dos atributos `nome` e `estado` serão validadas.

Quando a `Cidade` for validada, os possíveis erros gerados são enviados para um objeto do tipo `org.springframework.validation.BindingResult`. Por isso, precisamos definir esse objeto como um parâmetro no método `criar()`, conforme mostra a Figura.

Observe que o método `alterar()` usa o método `criar`. Ao mudar o número de parâmetros do método, seu editor/IDE deve reclamar um erro. Por enquanto, simplesmente mude a chamada do método `criar()`, no corpo de `alterar()`, para `criar(cidade, null)`.

```
32  @PostMapping("/criar")
33  public String criar(@Valid Cidade cidade, BindingResult validacao) {
```

Agora, tudo que precisamos fazer é usar os métodos disponíveis em `org.springframework.validation.BindingResult` para verificar pela existência de erros. Em caso de erros, vamos imprimir os erros e as mensagens correspondentes na console do sistema. Observe que a criação de uma nova cidade agora está condicionada à não existência de problemas de validação.

```

32     @PostMapping("/criar")
33     public String criar(@Valid Cidade cidade, BindingResult validacao) {
34
35         if (validacao.hasErrors()) {
36             validacao
37                 .getFieldErrors()
38                 .forEach(error ->
39                     System.out.println(
40                         String.format("O atributo %s emitiu a seguinte mensagem %s",
41                             error.getField(),
42                             error.getDefaultMessage())
43                     )
44                 );
45         }
46     } else {
47         cidades.add(cidade);
48     }
49
50     return "redirect:/";
51 }
```

O método `hasErrors()`, na linha 35, retorna true caso exista algum erro de validação. Se não houver erros, o código na linha 47 é executado e uma nova cidade é inserida.

Se houverem erros, o método `getFieldErrors()` retorna uma lista de erros (linha 37). Nesse código, usamos um `forEach` para iterar sobre a lista e imprimir os erros na linha de comando. Usamos dois métodos de `FieldError`, representado pela variável `error`. Esse objeto fornece dois métodos que permitem identificar o atributo onde o erro foi detectado (`getField()`, linha 41) e a mensagem de erro associada (`getDefaultMessage()`, linha 42).

Se você executar a aplicação e tentar inserir uma cidade sem informar os valores nome e estado, vai perceber a seguinte mensagem na console do sistema:

```

0 atributo nome emitiu a seguinte mensagem O nome da cidade deve ter entre 5 e 60 caracteres
0 atributo estado emitiu a seguinte mensagem A sigla do estado está limitada a dois caracteres
0 atributo estado emitiu a seguinte mensagem Sigla do estado deve ser informado
0 atributo nome emitiu a seguinte mensagem O nome da cidade deve ser informado
```

Contudo, não é bem isso que um usuário espera, não é mesmo? A mensagem precisa aparecer na página Web, e não na console do sistema. Por isso, vamos precisar do Freemarker e do Bootstrap para ajustar nossa página para mostrar as mensagens de erro.

O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana05-30-validacao-javax-completa`.

EXIBINDO ERROS NA TELA

Nosso usuário espera ser avisado caso erros impeçam alguma operação de ser executada. Por isso, vamos usar o Bootstrap e Freemarker para mostrar esses erros na tela. Além disso, precisamos de um pequeno ajuste no `CidadeController` para enviar os erros para página Web.

Vamos começar pelo controlador. Abra a classe `CidadeController` e adicione o parâmetro `memoria`, do tipo `org.springframework.ui.Model`, no método `criar()`. Assim como já fizemos antes, esse objeto é responsável por carregar dados em uma solicitação – seja ela indo para a GUI ou vindo da GUI.

Mais uma vez, observe que o método `alterar()` usa o método `criar()`. Ao mudar o número de parâmetros do método, seu editor/IDE deve reclamar um erro. Por isso, assim como fizemos antes, simplesmente mude a chamada do método `criar()`, no corpo de `alterar()`, para `criar(cidade, null, null)`.

```
32 |     @PostMapping("/criar")
33 |     public String criar(@Valid Cidade cidade, BindingResult validacao, Model memoria) {
```

Em seguida, em vez de imprimir os erros na console do sistema, vamos usar a mesma repetição para adicionar os erros como atributos da variável `memoria`.

```
35 |         if (validacao.hasErrors()) {
36 |             validacao
37 |                 .getFieldErrors()
38 |                 .forEach(error ->
39 |                     memoria.addAttribute(
40 |                         error.getField(),
41 |                         error.getDefaultMessage())
42 |                 );
```

Assim como antes, o método `getFieldErrors()`, na linha 37, retorna uma lista de erros caso eles existam (linha 35). A linha 38 faz uma iteração sobre esses erros, usando a variável `error` como referência para cada erro encontrado durante a iteração. Em vez de imprimir o erro na console do sistema, usamos a variável `memoria` para guardar os erros como atributos. Observe que o método `getField()` retorna o atributo onde o erro ocorreu. Por exemplo, se o nome da cidade não foi informado, então o `getField()` retorna `nome`. O `getDefaultMessage()` retorna a mensagem de erro definida para o erro ocorrido. Por exemplo, dizendo que o nome não foi informado.

Lembre-se que as mensagens de erro foram definidas no arquivo `messages.properties`.

Por fim, precisamos adicionar a lista de cidades à `memoria` antes de redirecionar o usuário para a página `crud.ftl` novamente. Veja como ficou o método completo após essas mudanças.

```

32     @PostMapping("/criar")
33     public String criar(@Valid Cidade cidade, BindingResult validacao, Model memoria) {
34
35         if (validacao.hasErrors()) {
36             validacao
37                 .getFieldErrors()
38                 .forEach(error ->
39                     memoria.addAttribute(
40                         error.getField(),
41                         error.getDefaultMessage()
42                     );
43
44             memoria.addAttribute("listaCidades", cidades);
45
46             return ("/crud");
47         } else {
48             cidades.add(cidade);
49         }
50
51         return "redirect:/";
52     }

```

Agora, precisamos ajustar a página `crud.ftl` para que ela exiba as mensagens de erro caso existam. Vamos começar ajustando os atributos dos formulários de criação e alteração. Na tag `form`, precisamos adicionar o atributo `novalidate` para evitar que o formulário use as mensagens padrão do HTML. Também precisamos adicionar a classe `needs-validation` do Bootstrap. Essa classe especifica que esse formulário tem validações.

```

21         <#if cidadeAtual??>
22             <form action="/alterar" method="POST" class="needs-validation" novalidate>
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" class="needs-validation" novalidate>
27             </#if>

```

Em seguida, vamos criar dois elementos para exibir as mensagens de erro. Esses elementos estarão logo abaixo dos elementos de entrada de dados, conforme mostra a Figura a seguir.

```

29         <div class="form-group">
30             <label for="nome">Cidade:</label>
31             <input
32                 value="${(cidadeAtual.nome) !}"
33                 name="nome"
34                 type="text"
35                 class="form-control ${!(nome??)?then('is-invalid', '')}"
36                 placeholder="Informe o nome da cidade"
37                 id="nome">
38
39             <div class="invalid-feedback">
40                 ${!nome}
41             </div>
42         </div>
43
44         <div class="form-group">
45             <label for="estado">Estado:</label>
46             <input
47                 value="${(cidadeAtual.estado) !}"
48                 name="estado"
49                 type="text"
50                 class="form-control ${!(estado??)?then('is-invalid', '')}"
51                 placeholder="Informe o estado ao qual a cidade pertence"
52                 id="estado">
53
54             <div class="invalid-feedback">
55                 ${!estado}
56             </div>
57         </div>

```

Observe que os novos elementos usam uma formatação de erro definida pelo Bootstrap (`invalid-feedback`). Dentro desses elementos nós apresentamos a mensagem de erro acessando o atributo definido no controlador, desde que ele exista. Para isso, usamos a sintaxe do Freemarker (`{}!`), o nome do atributo que colocação na variável `memoria` (`nome` e `estado`), e o marcador do Freemarker (`!`), que apenas apresenta um valor caso ele exista.

Outra alteração que já pode ser observada na Figura anterior está nas linhas 35 e 50. Além da classe `form-control`, já definida em aulas anteriores, adicionamos uma expressão que avalia a existência do atributo `nome` (linha 35) e `estado` (linha 50) na primeira parte. A segunda parte da expressão (`then`), adiciona a classe `is-invalid` caso nome ou estado existam na variável `memoria`. Isso é usado pelo Bootstrap para marcar a caixa de texto em vermelho, indicando que existe um erro.

Execute o código e tente enviar o formulário sem informar o valores. Você verá as mensagens de erro. Depois, tente colocar um valor, mas não outro. Também tente colocar espaços em branco e veja o que acontece.

GERENCIAMENTO DE CIDADES

UM CRUD PARA CRIAR, ALTERAR, EXCLUIR E LISTAR CIDADES

Cidade:

Informe o nome da cidade

①

O nome da cidade deve ser informado

Estado:

Informe o estado ao qual a cidade pertence

①

A sigla do estado está limitada a dois caracteres

CRIAR

| Nome | Estado | Ações |
|------|--------|-------|
| | | |

O código desenvolvido nesta Seção está disponível no **Github**, na branch **semana05-40-validacao-web**.

MANTENDO VALORES DIGITADOS

Ao executar o código na seção anterior você deve ter notado que os valores digitados antes da validação não são mantidos quando a página é recarregada. Isso acontece porque quando o formulário é enviado para o controlador, os dados informados são validados e depois são descartados. Quando a página é recarregada com os novos dados vindos do controlador, a solicitação possui as mensagens de erro, mas não os valores digitados anteriormente.

É importante mantes os valores anteriores porque o usuário consegue ver o que está errado. Para fazer isso, tudo que precisamos é armazenar os valores na variável memoria e mostra-los quando a página for recarregada.

Primeiro, precisamos salvar na variável `memoria` os valores informados pelo usuário. Para isso, abra a classe `CidadeController` e adicione o código mostrado nas linhas 44 e 45 da Figura abaixo.

```
35     if (validacao.hasErrors()) {
36         validacao
37             .getFieldErrors()
38             .forEach(error ->
39                 memoria.addAttribute(
40                     error.getField(),
41                     error.getDefaultMessage())
42             );
43
44         memoria.addAttribute("nomeInformado", cidade.getNome());
45         memoria.addAttribute("estadoInformado", cidade.getEstado());
46         memoria.addAttribute("listaCidades", cidades);
47
48     return ("/crud");
```

Esse código adiciona os atributos `nomeInformado` e `estadoInformado` na variável `memoria`. Esses atributos armazenam os valores informados pelo usuário, que foram recebidos pelo parâmetro `cidade`, e podem ser recuperados usando o método `getNome()` e `getEstado()`.

Em seguida, precisamos adicionar uma expressão condicional para exibir esses valores no input caso eles existam. Para isso, abra a página `crud.ftl` e acione às tags `input` a expressão `${nomeInformado!}` e `${estadoInformado}` , conforme mostra a Figura a seguir.

```
29 <div class="form-group">
30     <label for="nome">Cidade:</label>
31     <input
32         value="${cidadeAtual.nome}!${nomeInformado!}"
33         name="nome"
34         type="text"
35         class="form-control ${nome??}?then('is-invalid', '')}"
36         placeholder="Informe o nome da cidade"
37         id="nome">
38
39     <div class="invalid-feedback">
40         ${nome!}
41     </div>
42 </div>
43
44 <div class="form-group">
45     <label for="estado">Estado:</label>
46     <input
47         value="${cidadeAtual.estado}!${estadoInformado!}"
48         name="estado"
49         type="text"
50         class="form-control ${estado??}?then('is-invalid', '')}"
51         placeholder="Informe o estado ao qual a cidade pertence"
52         id="estado">
53
54     <div class="invalid-feedback">
55         ${estado!}
56     </div>
57 </div>
```

Pronto! Teste seu código e veja como as coisas funcionam agora.

É importante lembrar que ao fazer os ajustes no método `criar()`, acabamos quebrando o método `alterar()`, que usava o `criar()` para salvar uma cidade alterada. Para corrigir isso, basta adicionar os parâmetros `org.springframework.validation.BindingResult` e `org.springframework.ui.Model` na assinatura do método `alterar` e repassar essas variáveis para o método `criar()`, conforme mostra a Figura abaixo.

Observe que ao fazer isso o método `alterar()` não aproveita automaticamente a validação do método `criar()`.

O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana05-50-validacao-web-completa`.

PÁGINAS DE ERRO

Outro tipo de validação comum é redirecionar o usuário para uma página de erro quando algo acontece, como alguém digitar uma página que não existe. Fazer isso com o Spring Boot é [bem simples](#). Basta criar a pasta `error` em `resources/static/`. Depois, basta criar em `/resources/static/error/` uma página para cada código de erro.

Você pode encontrar uma lista dos códigos de erro HTML [aqui](#).

Vamos fazer um exemplo para o erro 404 – página não encontrada. Crie uma página HTML chamada `404.html` em `/resources/static/error`. Em seguida, adicione a mensagem que quer mostra ao seu usuário. Pronto! Execute o projeto e tente informar uma página que não existe.

The screenshot shows a web browser window with the URL `localhost:8080/qualquercoisa` in the address bar. The main content area displays the following text:

Oooops, parece que essa página não existe!

GERENCIAMENTO DE CIDADES

UM CRUD PARA CRIAR, ALTERAR, EXCLUIR E LISTAR CIDADES

Cidade:

Estado:

CRIAR

| Nome | Estado | Ações |
|------|--------|-------|
|------|--------|-------|

The browser interface includes standard navigation buttons (back, forward, search) and a toolbar with various icons.

O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana05-60-validacao-pagina-erro`.