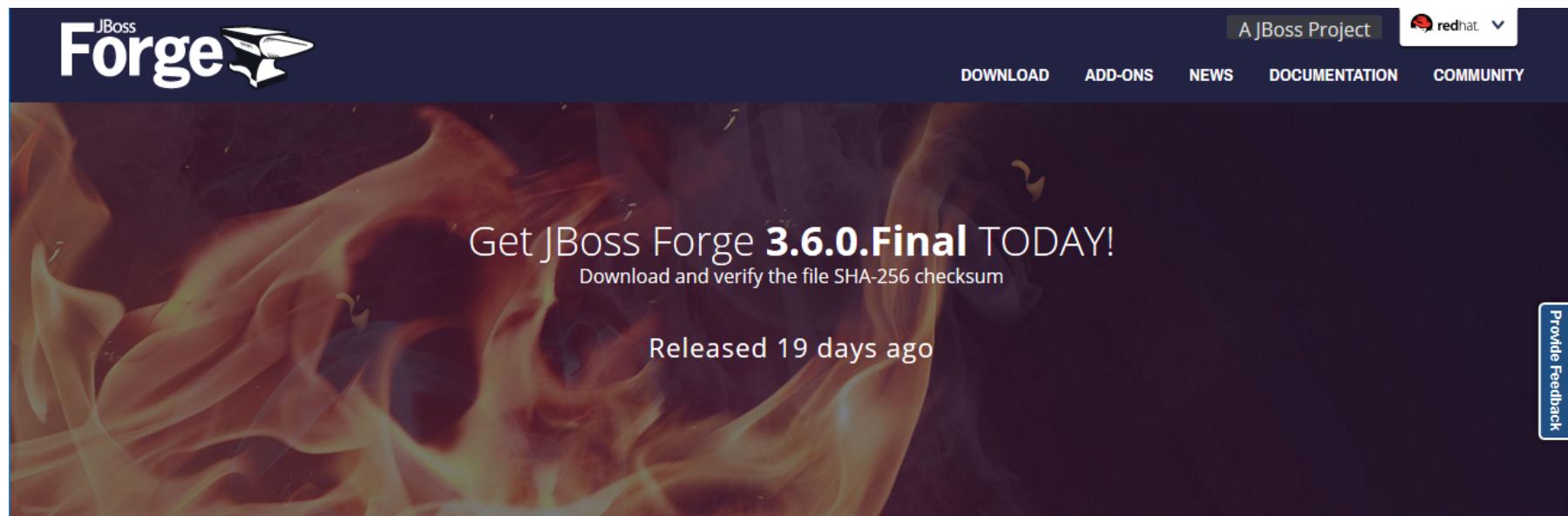


Transcrição

Para nossos estudos, usaremos como base o projeto da [Casa do Código](https://www.casadocodigo.com.br/) (<https://www.casadocodigo.com.br/>). Além de ser um projeto bem conhecido, onde criaremos alguns cadastros, faremos o acompanhamento de livros, autores, preços e iremos mais a fundo com nosso projeto criando outras funcionalidades.

Para não perder tempo com a configuração inicial, faça o download do [JBoss Forge 3.0.1.Final](https://forge.jboss.org/download) (<https://forge.jboss.org/download>). Escolha seu sistema operacional, e faça o download para sua máquina.



Works in the environment you **Use.**

Além disso, vamos rodar nosso projeto com o Application Server **Wildfly**. A versão que utilizaremos será a 10.0.0.Final. A versão mais atual pode ser outra, mas tenha cuidado ao pegar uma versão muito diferente, pois podemos ter problemas de configuração. Você pode baixar o Wildfly no endereço <http://wildfly.org/downloads/> (<http://wildfly.org/downloads/>)

Após realizar o download, descompacte os arquivos. Renomeia a pasta que você descompactou o JBossForge para apenas `forge`, assim será mais fácil o acesso via terminal.

Abra o seu **Terminal** e navegue até seu *workspace*. Dentro do seu workspace, aponte o terminal para o local onde você descompactou o forge. No meu caso:

```
../forge/bin/forge
```

[COPIAR CÓDIGO](#)

Pressione `Enter` e assim já estaremos dentro do JBoss Forge. Depois, podemos digitar comando `forge` que criará nosso projeto inicial. Assim digite no terminal

```
project-new --named casadocodigo
```

[COPIAR CÓDIGO](#)

Os seus passos podem ser um pouco diferentes do realizado no vídeo, dependendo da versão do Forge utilizada, mas o resultado final será muito próximo.

Ao ser perguntado pelo pacote, digite `br.com.casadocodigo`.

Na *Version*, pode deixar a que foi sugerida, apenas pressione *ENTER*.

No *Final Name*, também pressione *ENTER*.

No *Project Location*, apenas pressione *ENTER*.

No *Project Type*, se WAR já estiver selecionado, apenas pressione *ENTER*. Senão, digite a opção para WAR.

No *Build System*, se MAVEN já estiver selecionado, apenas pressione *ENTER*. Senão, digite a opção para MAVEN.

Na opção de *Stack*, vamos apenas digitar *ENTER* sem selecionar nada.

Agora você pode abrir a pasta do seu *Workspace* e navegar até o projeto criado `casadocodigo`. Você vai perceber que temos apenas uma pasta `src` e um arquivo `pom.xml`. Dentro da pasta `src`, temos apenas a sub estrutura de pacotes que informamos na configuração acima.

Agora que já temos a base, vamos configurar nossa stack. Voltando ao *Terminal*, digite

```
faces-setup --facesVersion 2.2
```

[COPIAR CÓDIGO](#)

Logo depois, configuramos para usar o CDI, que será o 1.1.

```
cdi-setup --cdiVersion 1.1
```

[COPIAR CÓDIGO](#)

Agora conseguimos alguma configuração no nosso `pom.xml`. Seu arquivo deve estar parecido com o de abaixo:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instantiation"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
<groupId>br.com.casadocodigo</groupId>
<artifactId>casadocodigo</artifactId>
<version>1.0.0-SNAPSHOT</version>
<build>
    <finalName>casadocodigo</finalName>
    <plugins>
        <plugin>
            <artifactId>maven-war-plugin</artifactId>
            <version>2.6</version>
            <configuration>
                <failOnMissingWebXml>false</failOnMissingWebXml>
            </configuration>
        </plugin>
    </plugins>
</build>
<packaging>war</packaging>
<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jboss.spec</groupId>
            <artifactId>jboss-javaee-6.0</artifactId>
            <version>3.0.3.Final</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>javax.annotation</groupId>
```

```
<artifactId>jsr250-api</artifactId>
<version>1.0</version>
<scope>provided</scope>
</dependency>
</dependencies>
</dependencyManagement>
<dependencies>
<dependency>
<groupId>org.jboss.spec.javax.faces</groupId>
<artifactId>jboss-jsf-api_2.1_spec</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>javax.enterprise</groupId>
<artifactId>cdi-api</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>javax.annotation</groupId>
<artifactId>jsr250-api</artifactId>
<scope>provided</scope>
</dependency>
</dependencies>
</project>
```

COPiar CÓDIGO

Com nosso `pom.xml` criado, podemos importar o projeto dentro do Eclipse. Entre no seu Eclipse e selecione a opção `File > Import` na caixa de busca, digite `maven` e já deve aparecer a opção `Existing Maven Projects`. Clique em `Next` e selecione o `pom.xml`, agora basta clicar em `Finish`.

Abra a view de Servers com o atalho `Ctrl + 3` e digite `Servers`, pressione ENTER. Dentro da View de Servers, clique com botão direito do mouse, e vá em `New > Server`. Procure pela opção `Wildfly 10`, caso não apareça, procure por `JBoss AS, Wildfly & EAP Server Tools`. Precisamos instalar esse Tools, clicando em `Next`. Quando aparecer os termos para serem aceitos, selecione a opção *I accept ...* e depois em `Finish`. Quando o processo estiver finalizado, o Eclipse irá pedir para reiniciar. Reinicie seu Eclipse.

Com o Plugin instalado, podemos colocar nosso servidor. Mais uma vez na aba `Servers`, clique com botão direito do mouse, e vá em `New > Server`. Procure pela opção `Wildfly 10` e clique em `Next` depois em `Next` novamente. Na caixa de `Home Directory`, precisaremos informar onde está a instalação do Wildfly. Clique em `Browser` e coloque a pasta descompactada do `Wildfly`. Na caixa de `Configuration file`, cliquem em `Browse...` e selecione o arquivo `standalone-full.xml`. Pressione em `Next`, então clique duas vezes no projeto `casadocodigo` que deve estar aparecendo para você, e ele estará dentro do servidor. Podemos finalizar a configuração do servidor, clicando em `Finish`.

Para que nosso projeto reconheça as bibliotecas que já estão dentro do `JBoss`, vamos precisar configurar. Clique com botão direto em cima do projeto, vá em `Build Path > Configure Build Path` selecione a aba `Libraries` e clique em `Add Library`. Procure pela opção `Server Runtime` e `Next`, depois selecione o `Wildfly 10...` que já configuramos e depois `Finish`.

Para verificarmos se realmente tudo está funcionando, precisamos criar um arquivo `index.html` dentro da pasta `webapp`. Clique na pasta `webapp` e pressione `Ctrl + N` e na caixa de busca, digite `HTML`. Vá em `Next` e dê o nome `index`. Clique em `Finish`. Abra o arquivo e vamos deixá-lo apenas com uma mensagem simples.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Casa do Código</title>
</head>
<body>
```

<h1>Instalação do Wildfly foi realizada com sucesso!**</h1>**

```
</body>  
</html>
```

COPIAR CÓDIGO

Uma vez que tudo está certinho, podemos inicializar nosso servidor. Clique na aba *Servers*, botão direito em cima do Servidor *Wildfly* e selecione `Start`.

Quando o servidor terminar de imprimir no *Console*, vá para seu navegador preferido e digite:

`http://localhost:8080/casadocodigo`. Se estiver vendo a mensagem `Instalação do Wildfly foi realizada com sucesso!`, seu projeto está configurado corretamente e já poderemos avançar.



04 Usando o JBoss Forge

[PRÓXIMA ATIVIDADE](#)

3%

Para que serve o JBoss Forge?

ATIVIDADES
4 de 20

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

MODO
NOTURNO

ABRIR
CADERNO



A Cria o `beans.xml` como arquivo base para importar toda a estrutura no Eclipse.

B Para configurar o `persistence.xml` e toda a conexão com o banco de dados. Bem como testar a conexão com o banco de dados.

C Ele gera toda a estrutura base do projeto integrada com o Maven. Configuração de dependências (bibliotecas e frameworks), estrutura de pastas e gera configurações iniciais que geralmente envolve arquivos XML.

D Serve para criar as pastas do Eclipse.



O JBoss Forge agiliza a configuração inicial do projeto, permitindo ganhar tempo indo direto para o código.



3%

O JBoss Forge é um gerador de projetos Java. Ele gera toda a estrutura base do projeto integrada com o Maven.

Configuração de dependências (bibliotecas e frameworks), estrutura de pastas e gera configurações iniciais que geralmente envolve arquivos XML.

ATIVIDADES
4 de 20

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

Sem ele, em um projeto tipicamente Java EE, devemos criar vários arquivos XML na mão, por exemplo: `persistence.xml` , `web.xml` , `faces-config.xml` , `pom.xml` , `beans.xml` entre outros.

PRÓXIMA ATIVIDADE





05 Criando a Estrutura Base

[PRÓXIMA ATIVIDADE](#)

6%

Agora que já fizemos o download do JBoss Forge abra o seu Terminal e navegue até o seu workspace com cd .

De dentro do workspace , chame o executável do Forge:

ATIVIDADES
5 de 20

HOME_DO_FORGE/bin/forge

[COPIAR CÓDIGO](#)

FÓRUM DO CURSO

VOLTAR PARA DASHBOARD



Você verá o forge sendo iniciado. Assim que ele iniciar, digite o comando de criação de projetos:

project-new -named casadocodigo

[COPIAR CÓDIGO](#)

Os seus passos podem ser um pouco diferentes do realizado no vídeo, dependendo da versão do Forge utilizada, mas o resultado final será muito próximo.





6%

ATIVIDADES
5 de 20FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

- Ao ser perguntado pelo pacote, digite
`br.com.casadocodigo`.
- Na Version, pode deixar a sugerida, apenas pressione [ENTER].
- No Final Name, também apenas [ENTER].
- No Project Location, apenas pressione [ENTER].
- No *Project Type*, se WAR já estiver selecionado, apenas pressione [ENTER]. Senão, digite a opção para WAR.
- No *Build System*, se MAVEN já estiver selecionado, apenas pressione [ENTER]. Senão, digite a opção para MAVEN.
- Na opção de *Stack*, vamos apenas digitar [ENTER] sem selecionar nada.

Agora que já temos a base, vamos configurar nossa stack.
Voltando ao Terminal, digite

```
faces-setup --facesVersion 2.2
```

[COPIAR CÓDIGO](#)

Logo depois, configuraremos para usar o CDI, que será o 1.1.



```
cdi-setup --cdiVersion 1.1
```

[COPIAR CÓDIGO](#)

6%

ATIVIDADES
5 de 20

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

MODO
NOTURNO

ABRIR
CADERNO



06

Import do projeto no Eclipse



7%

Abra o seu Eclipse e selecione a opção `File > Import`, e na caixa de busca, digite **maven** que já deve aparecer a opção `Existing Maven Projects`. Clique em `Next` e selecione o `pom.xml`, agora basta clicar em `Finish`.

ATIVIDADES
6 DE 20FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

85.5k xp



Abra a view de Servers com o atalho `Ctrl + 3` e digite `Servers`, pressione ENTER.

Dentro da View de Servers, clique com botão direito do mouse, e vá em `New > Server`.

Procure pela opção `Wildfly 10`, caso não apareça, procure por `JBoss AS, Wildfly & EAP Server Tools` e clique em `Next`.

Quando surgirem os termos, selecione a opção `I accept ...` e depois em `Finish`. Seu Eclipse irá solicitar ser reiniciado.

Com o plugin instalado, vá novamente em `Servers`, botão direito do mouse `New > Server` procure por `Wildfly 10` e clique em `Next` depois em 'Next' novamente. Na caixa de `Home Directory`, clique em `Browser` e coloque a pasta descompactada do `Wildfly`. Na caixa de `Configuration`



file , clique em Browse... e selecione o arquivo standalone-full.xml . Clique em Next , então coloque o projeto casadocodigo para o quadro da direita e ele estará dentro do servidor. Podemos finalizar a configuração do servidor, clicando em Finish .



7%

ATIVIDADES
6 DE 20

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

Por fim, clique com botão direto em cima do projeto, vá em Build Path > Configure Build Path selecione a aba Libraries e clique em Add Library . Procure pela opção Server Runtime e Next , depois selecione o Wildfly 10... que já configuramos e por fim Finish .



Opinião do instrutor



85.5k xp

Seu projeto deve aparecer pronto e configurado no Eclipse, já com as libs do servidor configurada. Dentro de Servers deve ser possível ver o Wildfly configurado e nosso projeto dentro dele.





07 Criando o arquivo inicial

[PRÓXIMA ATIVIDADE](#)

8%

Clique na pasta webapp e pressione `Ctrl + N` e na caixa de busca, digite HTML. Vá em 'Next' e dê o nome `index.html`.

Clique em `Finish`. Coloque uma mensagem padrão dentro de uma tag `<h1>...</h1>` e inicie o servidor para testar.

ATIVIDADES
7 de 20

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD



Opinião do instrutor

Você deve visualizar a mensagem que digitou na tag `<h1>` corretamente, o que significa que tudo parece estar configurado corretamente. O código deve ser parecido com:

```
<!DOCTYPE html>
```



```
<html>
<head>
<meta charset="UTF-8">
<title>Casa do Código</title>
</head>
<body>
```



8%

<h1>Instalação do Wildfly foi realizada c

ATIVIDADES
7 de 20

```
</body>
</html>
```

COPIAR CÓDIGO

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD





08 Criando nosso primeiro formulário

Transcrição

Vamos criar mais um HTML para o cadastro. Começamos criando uma pasta para separar os arquivos, New > Folder e nomeamos de `livro`. Dentro dessa pasta, vamos criar um HTML New > HTML e vamos nomeá-lo de `form.html`. Clicamos em Next, e procuramos uma opção chamada **New Facelet Composition Page**. Facelets é o framework que cuida dos templates do JSF, e clique em `Finish`. Vamos deixar apenas a estrutura abaixo:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
  
<html xmlns="http://www.w3.org/1999/xhtml"  
      xmlns:h="http://xmlns.jcp.org/jsf/html"  
      xmlns:f="http://xmlns.jcp.org/jsf/core">  
  
</html>
```

COPiar Código

Nesse curso, não estudaremos as coisas básicas do JSF, iremos detalhar apenas o que é um mais avançado ou um pouco diferente do padrão do JSF.

Vamos começar criando o form, e dentro dele colocamos um `Label` e um `InputText` para a entrada do texto, conforme abaixo:

```
<!-- Mantém o código de abertura do html acima -->
<h:form>
    <div>
        <h:outputLabel value="" />
        <h:inputText />
    </div>
</h:form>
<!-- Fecha o html abaixo -->
```

COPIAR CÓDIGO

Alguns campos que são importantes quando tratamos de cadastro de livro, são eles: Título, Descrição, Número de Páginas e o Preço.

Vamos duplicar os dados, selecionando o que queremos copiar, e pressionando (Alt + Ctrl + Up / Down). Se no seu sistema operacional não funcionar, basta copiar e colar os campos.

Nomeamos cada um dos `<h:outputLabel />` com os quatro campos que citamos acima. E ao

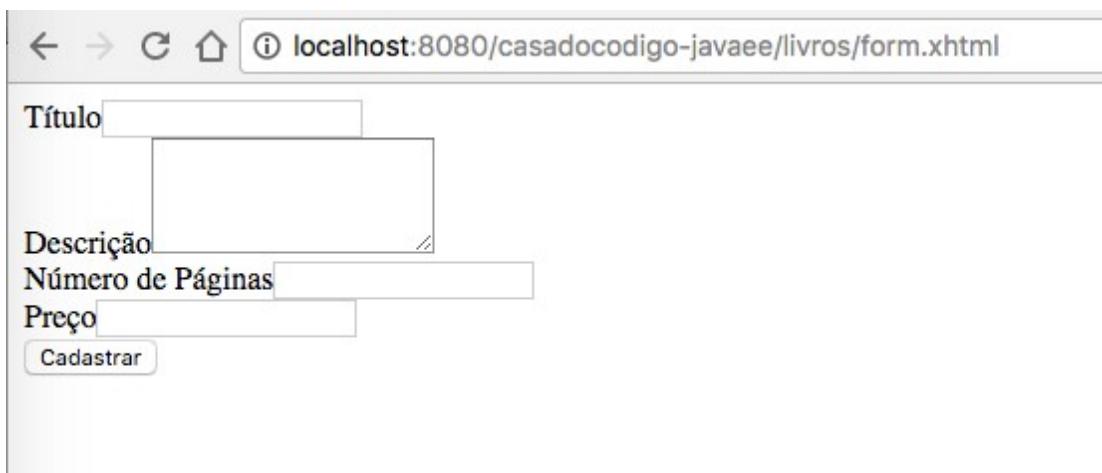
final de cada formulário, sempre temos um botão de confirmação. Assim, vamos colocar um botão com `<h:commandButton />` e colocar o value dele para *Cadastrar* conforme abaixo:

```
<div>
    <h:outputLabel value="Título" />
    <h:inputText />
</div>
<div>
    <h:outputLabel value="Descrição"/>
    <h:inputTextarea />
</div>
<div>
    <h:outputLabel value="Número de Páginas"/>
    <h:inputText />
</div>
<div>
    <h:outputLabel value="Preço"/>
    <h:inputText />
</div>
<h:commandButton value="Cadastrar" />
```

COPIAR CÓDIGO

Além disso, precisamos renomear nossa página para o padrão do JSF, assim, feche o arquivo `form.html` e renomeei-o para `form.xhtml`. Como o XHTML já é reconhecido pelo servidor

como arquivo JSF, já podemos subir o servidor e realizar o primeiro teste. Quando acessamos <http://localhost:8080/casadocodigo/livro/form.xhtml> temos o seguinte resultado.



Toda tela jsf, em geral possui uma classe Java por trás que chamamos de BackBean. Por isso, vamos criar uma classe, clicando em cima da pasta *src/main/java* pressione **Ctrl + N**, escolha a opção **Class** e dê o nome de `AdminLivrosBean`. Vamos colocá-la no pacote `br.com.casadocodigo.loja.beans`.

Dentro dessa classe, criaremos o método salvar da seguinte forma:

```
public class AdminLivrosBean {  
  
    private Livro livro;
```

```
public void salvar() {  
    System.out.println("Livro salvo com Sucesso!");  
}  
}
```

COPiar CÓDIGO

Os dados do livro no JSF são associados diretamente aos atributos do bean, assim vamos criar um atributo em nossa classe, como sendo `private Livro livro;`. Nesse ponto, recebemos um erro, pois a classe Livro ainda não existe. Vamos criá-la conforme abaixo, e já gerar os *getters* e *setters* da classe com o atalho `Ctrl + 3 > ggas`.

```
public class Livro {  
  
    private String titulo;  
    private String descricao;  
    private BigDecimal preco;  
    private Integer numeroPaginas;  
  
    // getters e setters aqui!  
  
    @Override  
    public String toString() {  
        return "Livro [titulo=" + titulo + ", descricao=" + descricao + ", preco=" + preco + ", numeroPaginas=" + numeroPaginas + "]";  
    }  
}
```

```
+ numeroPaginas + " ] " ;  
}  
}
```

COPiar CÓDIGO

Já criamos também o `toString()` do Livro, com o atalho `Ctrl + 3` e na caixa de busca, digite `toString`. Selecionamos a opção `Generate toString()` e clicamos em selecionar todos os campos, logo depois em `Finish`.

Vá para a classe `AdminLivrosBean`, e também crie os getters e setters do Livro, com o atalho `Ctrl + 3 > ggas` como já estamos acostumados.

Uma vez que temos a classe `Livro` e também `AdminLivrosBean`, vamos relacioná-los em nossa tela pelos atributos `value` do xhtml utilizando a *ExpressionLanguage* do JSF.

```
<div>  
    <h:outputLabel value="Título" />  
    <h:inputText value="#{adminLivrosBean.livro.titulo}" /> <!-- NOVID  
</div>  
<!-- REPITA O PROCESSO PARA OS DEMAIS INPUTS --&gt;</pre>
```

COPiar CÓDIGO

Para vermos os resultados na tela, basta ir na view *Servers* abrir na setinha do servidor para que possamos ver nosso projeto. Clicamos em cima do projeto com o botão direito vamos na opção `Full Publish` que reenviará os nossos arquivos novos para o servidor. Assim, podemos testá-los na tela.

Ao acessar a aplicação no navegador, recebemos o erro: `Target Unreachable, identifier 'adminLivrosBean' resolved to null`. Pois o JSF ainda não exerga nosso Bean. Para que o JSF veja nosso Bean, temos que utilizar o **CDI**. Que veio no *JavaEE 6* e ficou mais poderoso no *JavaEE 7*.

Assim, para que o CDI libere nosso Bean para a tela, utilizamos a annotation `@Named` em cima do Bean. Agora o JSF conseguirá ver nosso Bean, com o nome *default* `adminLivrosBean`, ou seja, apenas a primeira letra ficou minúscula. Vamos realizar o `Full Publish` novamente e poderemos testar.

Ao preencher o formulário e tentar cadastrar, recebemos o seguinte erro: `... value="#" {adminLivrosBean.livro.titulo}": Target Unreachable, identifier 'adminLivrosBean' resolved to null`. O que aconteceu agora é que, ao tentar pegar o título, o livro veio nulo. Por isso, vamos fazer `private Livro livro = new Livro()` lá no nosso Bean, e dessa forma evitamos receber esse `null`. Realizamos `Full Publish` novamente, e ao preencher os dados e clicar em *Cadastrar*, dessa vez os dados sumiram e não obtivemos erros.

Isso ocorreu pois nosso Bean agora está sendo gerenciado pelo CDI, e o CDI por default possui

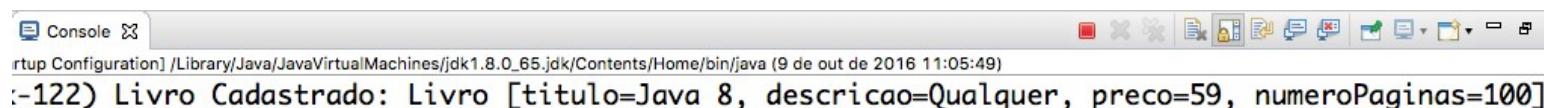
um tempo de vida muito curto, no caso, sempre que o JSF precisa do Bean, ele cria uma nova instância e entrega ao JSF. Porém, para que os dados fiquem vivos durante toda a requisição do usuário, precisamos que o CDI deixe ele vivo durante todo o Request, para isso, usamos a annotation `@RequestScoped`. Fique atento para utilizar o `@RequestScoped` correto, do pacote do CDI, que é `javax.enterprise.context.RequestScoped`. Ao realizar o Full Publish novamente, podemos testar o cadastro.

Dessa vez, os valores permaneceram, porém não obtivemos nenhum resultado no console, como era esperado. Pois precisamos "ligar" o botão Cadastrar com o método salvar do nosso Bean. Fazemos isso com o atributo `action` do `commandButton`, ficando assim:

```
<h:commandButton value="Cadastrar" action="#{adminLivrosBean.salvar}"
```

[COPIAR CÓDIGO](#)

Finalmente, realizamos *Full Publish* e ao preencher o form e clicar em Cadastrar, vemos no Console do Eclipse o resultado esperado.



```
Console X
rtup Configuration] /Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/bin/java (9 de out de 2016 11:05:49)
:-122) Livro Cadastrado: Livro [titulo=Java 8, descricao=Qualquer, preco=59, numeroPaginas=100]
```

Fizemos uso do CDI e começamos com algo simples, mas vamos fazer de fato nosso cadastro funcionar.



09 Criando o formulário de cadastro de Livros

[PRÓXIMA ATIVIDADE](#)

10%

Crie uma nova pasta dentro de `src/main/webapp/` chamada `livros`.

ATIVIDADES
9 de 20

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

MODO
NOTURNO

ABRIR
CADERNO

Dentro dessa nova pasta, crie um novo arquivo `xhtml` chamado `form.xhtml`. A estrutura base do arquivo conforme abaixo, dentro da estrutura adicione os campos `titulo`, `descrição`, numero de páginas e preço. Adicione também um botão para realizar o Cadastro.

Ao terminar, tente subir o servidor para ver o resultado:

[\(http://localhost:8080/casadocodigo/livros/form.xhtml\)](http://localhost:8080/casadocodigo/livros/form.xhtml)

Opinião do instrutor



Seu formulário deve estar parecido com:



10%

ATIVIDADES
9 de 20

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD


MODO
NOTURNO


ABRIR
CADERNO



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-t

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">

<h:body>
    <h:form>
        <div>
            <h:outputLabel value="Título" />
            <h:inputText />
        </div>
        <div>
            <h:outputLabel value="Descrição"/>
            <h:inputTextarea rows="4" cols="2">
        </div>
        <div>
            <h:outputLabel value="Número de P
            <h:inputText />
        </div>
        <div>
            <h:outputLabel value="Preço"/>
```



10%

```
<h:inputText/>
```

```
</div>
```

```
<h:commandButton />
```

```
</h:form>
```

```
</h:body>
```

```
</html>
```

[COPIAR CÓDIGO](#)

ATIVIDADES
9 de 20

E resultado vai ser parecido com isso :

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

titulo	<input type="text"/>	<input type="button" value=""/>
descricao	<input type="text"/>	<input type="text"/>
Números de Páginas	<input type="text"/>	
Preço	<input type="text"/>	<input type="button" value=""/>
<input type="button" value="Cadastrar"/>		

MODO
NOTURNO

ABRIR
CADERNO





10 Criando o bean de cadastro de livros

[PRÓXIMA ATIVIDADE](#)

12%

No nosso formulário, precisamos criar um responsável por ele que irá receber e controlar algumas informações, ele será uma classe. Por ser o responsável de controlar uma view, damos o nome dele o sufixo Bean.

ATIVIDADES
10 de 20

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

MODO
NOTURNOABRIR
CADEIRNO



```
package br.com.casadocodigo.beans;  
  
public class AdminLivrosBean {  
  
    public void salva(){  
        System.out.println("O livro foi salvo");  
    }  
}
```

12%

ATIVIDADES
10 de 20[COPIAR CÓDIGO](#)FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

O livro que passamos no sysout é a representação de um Livro no nosso código, ele será uma classe. Teremos que criar ela. A classe se chamará `Livro` e seu pacote será `br.com.casadocodigo.loja.models`, precisamos fazer os seguintes passos nessa classe :

- Criar os seus atributos que serão: `titulo` , `descrição` , `preço` , Número de páginas .
- Criar os getters e setters para cada atributo, também será necessário sobrescrever o método `toString` , lembre-se de usar o atalho `ctrl+3 ggas` e `ctrl+3 toString` para agilizar esse processo.

Feito isso podemos voltar na classe `AdminLivrosBean` e fazer os seguinte:



12%

ATIVIDADES
10 de 20

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

MODO
NOTURNO

ABRIR
CADERNO

- Instanciar a classe Livro no nosso Bean
- Criar um getter e um setter para a nossa instância da classe livro
- No método `salva` vamos criar um syout do livro cadastrado.

Para concluir precisamos associar a nossa view com o `AdminLivrosBean`, para fazer isso adicionamos uma *Expression Language* do jsf no value de cada campo no nosso form.xhtml, ele vai ficar dessa forma :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-t

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelet"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
```

L



12%

ATIVIDADES
10 de 20FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARDMODO
NOTURNOABRIR
CADERNO

```
<h:form>
    <div>
        <h:outputLabel value="titulo" />
        <h:inputText value="#{adminLivrosBean.titulo}" />
    </div>
    <div>
        <h:outputLabel value="descricao" />
        <h:inputTextarea rows="4" cols="8" value="#{adminLivrosBean.descricao}" />
    </div>
    <div>
        <h:outputLabel value="Números de Páginas" />
        <h:inputText value="#{adminLivrosBean.numeroDePaginas}" />
    </div>
    <div>
        <h:outputLabel value="Preço" />
        <h:inputText value="#{adminLivrosBean.preco}" />
    </div>
    <h:commandButton value="Cadastrar" />
</h:form>
</html>
```

COPIAR CÓDIGO

Se você subir a aplicação e tentar submeter o formulário, o seguinte erro irá acontecer :

Error processing request

Context Path: casadocodigo
Servlet Path:/livros/form.xhtml
Path Info:
Query String:
Stack Trace

```
javax.servlet.ServletException: /livros/form.xhtml @13,57 value="#{adminLivrosBean.livro.titulo}": Target Unreachable, identifier 'adminLivrosBean' resolved to null
javax.faces.webapp.FacesServlet.service(FacesServlet.java:671)
```

io.undertow.servlet.handlers.ServletHandler.handleRequest(ServletHandler.java:85)
in undertow servlet handlers security ServletSecurityRoleHandler handleRequest(ServletSecurityRoleHandler.java:67)



Esse tipo de exception acontece pois nossa ainda não está associada com a nosso Bean, por isso ele não sabe identificar a Expression Language #{adminLivrosBean}. Para idenficiar o nosso Bean para a nossa view, fazemos isso usando a anotação @Named do pacote javax.inject.Named , implemente essa annotation no seu Bean e teste novamente.



12%

ATIVIDADES
10 de 20

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

Opinião do instrutor

O seu Bean vai ficar parecido com isso:



```
package br.com.casadocodigo.loja.bean;

import javax.inject.Named;

import br.com.casadocodigo.loja.modelo.Livro;

@Named
```





```
public class AdminLivrosBean {  
  
    private Livro livro = new Livro();  
  
    public void salva() {  
        System.out.println("Livro cadastrado:  
    }  
  
    public Livro getLivro() {  
        return livro;  
    }  
  
    public void setLivro(Livro livro) {  
        this.livro = livro;  
    }  
}
```

COPIAR CÓDIGO



O @Named faz parte da API que faz parte do JavaEE 6, e deve trazer uma grande melhora no JavaEE 7.



A nossa view vai ficar dessa forma :





12%

ATIVIDADES
10 de 20FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD
MODO
NOTURNO
ABRIR
CADERNO

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
                     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-t

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelet"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">

<h:form>
    <div>
        <h:outputLabel value="titulo" />
        <h:inputText value="#{adminLivrosBean
    </div>
    <div>
        <h:outputLabel value="descricao" />
        <h:inputTextarea rows="4" cols="8" va
    </div>
        <h:outputLabel value="Números de Pági
        <h:inputText value="#{adminLivrosBean
    <div>
        <h:outputLabel value="Preço" />
        <h:inputText value="#{adminLivrosBean
    </div>
        <h:commandButton value="Cadastrar" />
    </h:form>
</html>
```

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 1 - Atividade 11 Os dados estão sumindo e concluindo o form | Alura

Meu caderno do curso (beta)

Este é seu caderno de anotações aqui na Alura. Todo o seu conteúdo fica visível apenas para você. Próximos passos para essa funcionalidade no nosso [FAQ](#)

Sabemos que o form fica em branco quando o tentamos submitta-lo. Isso acontece por causa do CDI que está gerenciando o Bean, e ele tem um tempo de vida muito curto, por isso ele limpa tudo quando o JSF não precisa mais. Temos que aumentar esse tempo de vida pelo menos até um pouco depois que o usuário recebe a sua resposta do servidor.

Para isso, aplique a anotação `@RequestScoped` do pacote `javax.enterprise.context.RequestScoped` no seu Bean.

E na nossa view, precisamos fazer que o formulário envie os dados para o bean, no `commandButton Cadastrar` adicionamos a seguinte expressão: `<h:commandButton value="Cadastrar" action="#{adminLivrosBean.salvar}" />`

Opinião do instrutor

O seu Bean é para ficar parecido com isso:

```
package br.com.casadocodigo.loja.bean;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

import br.com.casadocodigo.loja.modelo.Livro;

@Named
@RequestScoped
public class AdminLivrosBean {

    private Livro livro = new Livro();

    public void salvar() {
```

```
        System.out.println("Livro cadastrado: " + livro);
    }

    public Livro getLivro() {
        return livro;
    }

    public void setLivro(Livro livro) {
        this.livro = livro;
    }
}
```

E nossa view também assim:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">

<h:form>
    <div>
        <h:outputLabel value="titulo" />
        <h:inputText value="#{adminLivrosBean.livro.titulo}" />
    </div>
    <div>
        <h:outputLabel value="descricao" />
        <h:inputTextarea rows="4" cols="8"
                        value="#{adminLivrosBean.livro.descricao}" />
    </div>
    <h:outputLabel value="Números de Páginas" />
    <h:inputText value="#{adminLivrosBean.livro.numeroPaginas}" />
    <div>
        <h:outputLabel value="Preço" />
        <h:inputText value="#{adminLivrosBean.livro.preco}" />
    </div>
    <h:commandButton value="Cadastrar" action="#{adminLivrosBean.salvar}" />
</h:form>
</html>
```

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 1 - Atividade 12 Configurando nosso DAO | Alura

Video Player is loading.

Current Time 0:00

Duration 10:06

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Configurando nosso DAO

Como os dados já estão chegando no Bean, precisaremos usar um `DAO` (Data Access Object) para enviar o Livro ao banco de dados. Criamos uma classe chamada `LivroDao` no pacote `br.com.casadocodigo.loja.daos`. Dentro de nossa

classe, faremos uso do `EntityManager` que você já deve conhecer. Ele é o objeto responsável por gerenciar nossas entidades, mantendo a ligação delas com o banco de dados. Usamos a annotation `@PersistenceContext` em cima do `EntityManager` para que o JPA possa injetá-lo no nosso Dao. Nossa classe ficará assim:

```
package br.com.casadocodigo.loja.daos;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

public class LivroDao {

    @PersistenceContext
    private EntityManager manager;
}
```

Se você estiver com qualquer dúvida sobre essa parte mais básica do JPA, recomendo que faça o [curso da plataforma da Alura](#) e que é pré-requisito. No de curso de Java EE, estudaremos recursos mais avançados do JPA.

Agora vamos criar o método `salvar()` para que nosso livro vá efetivamente para o banco de dados. Usaremos o `EntityManager` com o método `persist()` para que ele envie nosso livro para o banco. Ficando nosso método assim:

```
public void salvar(Livro livro) {
    manager.persist(livro);
}
```

Precisaremos informar ao JPA que nossa entidade `Livro` está vinculada a uma tabela do banco de dados, e para isso usamos a anotação `@Entity` do pacote `javax.persistence.Entity`. Além disso, precisamos informar quem é o `Id` que será ligado à tabela e sua forma de incremento, para que nosso banco crie automaticamente a chave primária e sua estratégia de incremento.

Usaremos o MySQL para este curso por ser um banco simples, fácil de usar e bem aceito no mercado.

```
@Entity

public class Livro {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    private String titulo;
    @Lob
    private String descricao;
    private BigDecimal preco;
    private Integer numeroPaginas;
```

```
public String getTitulo() {  
    return titulo;  
}  
public void setTitulo(String titulo) {  
    this.titulo = titulo;  
}
```

Observe que adicionamos o `@GeneratedValue`, em que usamos o `strategy`. A chave-primária será criada automaticamente pelo MySQL.

Cada um dos atributos da classe Livro, se tornará uma coluna em nossa tabela automaticamente. Para finalizar esta classe, vamos anotar o atributo `descricao` com `@Lob` para informar que ele pode receber um grande valor de texto.

E então, precisamos de mais alguma configuração?

Sempre que usamos JPA, precisaremos do arquivo padrão de configuração, o `persistence.xml`. Esse arquivo deve ser criado na pasta `src/java/resources/META-INF/`, que é a pasta *default* que a especificação procura o `persistence.xml`. Assim, a configuração básica do `persistence` ficará assim:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>  
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    version="2.1" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence  
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">  
    <persistence-unit name="casadocodigo-dev" transaction-type="JTA">  
        <description>Dev persistence unit</description>  
        <provider>org.hibernate.ejb.HibernatePersistence</provider>  
        <!-- java transaction api || JNDI -->  
        <jta-data-source>java:jboss/datasources/casadocodigoDS</jta-data-source>  
        <properties>  
            <property name="hibernate.hbm2ddl.auto" value="update"/>  
            <property name="hibernate.show_sql" value="true" />  
            <property name="hibernate.format_sql" value="true" />  
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect"/>  
        </properties>  
    </persistence-unit>  
</persistence>
```

Dentro do arquivo, não temos muita novidade. Destacando as principais tags, temos `provider`, que informa ao JPA nosso provedor (ou seja, implementação da especificação) que vamos usar. No caso o Hibernate que já é disponibilizado pelo *Wildfly*.

As tags padrão de propriedades, que seguem o formato *chave / valor*, ficando

- `hibernate.hbm2ddl.auto` onde pedimos para o Hibernate criar e manter nosso banco atualizado de acordo com as

entidades, através do valor `update`:

- `hibernate.show_sql` pedimos para o Hibernate exibir o SQL que ele gera, através do valor `true`;
- `hibernate.format_sql` pedimos para o SQL exibido na configuração anterior, ser formatado bonitinho através do valor `true`;
- `hibernate.dialect` dizemos ao Hiberante o dialeto do banco, para ele criar *Queries* próprias para aquele banco de dados;

Outra tag importante é a `<jta-data-source />`. Um `datasource` é a fonte de dados do sistema. Essa fonte (local onde obtemos os dados) é ligada a JTA, que é mais um carinha novo que temos no curso, porém muito usado no mundo JavaEE. O JTA (Java Transaction API) é a API de transações Java, que cuida de toda a transação da nossa aplicação, onde começa e termina, e quando realizar o rollback das transações.

Toda vez que trabalhamos com banco de dados, o banco nos exige que começemos uma transação antes de qualquer operação que altera o estado do banco. Se você já trabalhou com banco de dados sem cuidar de transações, certamente o seu sistema realizava commit automático. Esse não é o padrão, geralmente temos que trabalhar com a transação, e se ninguém fizer, você terá que fazer na mão. Um código de exemplo, seria:

```
public void salvar(Livro livro){  
    manager.getTransaction().begin();  
    manager.persist(livro);  
    manager.getTransaction().commit();  
}
```

Desta forma, abrimos e alteramos o banco, depois, comitamos. Lembrando que transações são úteis apenas em casos que alteram o estado do banco de dados. Mas temos a opção de deixar a cargo do JTA o gerenciamento das transações, ou seja, não precisamos mais inicializar e nem finalizar as transações.

Ainda no arquivo `persistence.xml` na tag `jta-data-source`, vemos que o valor informado começa com `java:..`. Isso é uma referência ao framework de nomes do java, o **JNDI** (Java Naming and Directory Interface). Ele nos permite relacionar um nome a um recurso, desta forma podemos concluir que o valor "java:jboss/datasources /casadocodigoDS" tem um recurso relacionado a ele e este recurso é o nosso datasource.

Mas e onde estão os dados do datasource? Como usuário, senha, driver, URL e etc...? Nós informamos essas configurações no datasource que é mantido no próprio servidor. O Wildfly nos provê uma area específica de data source onde podemos colocar tudo isso.

Abra o arquivo `standalone-full.xml` e localize a tag `<datasources>`, dentro desta tag nós podemos configurar nosso datasource, já temos até um exemplo, porém esse exemplo não serve para nosso sistema. Vamos criar o nosso próprio datasource.



13 Criando o objeto LivroDao

[PRÓXIMA ATIVIDADE](#)

15%

Crie seu `LivroDao` e adicione o método `salvar` que deve pegar o `EntityManager` e chamar o método `persist` para o livro que foi passado como parâmetro.

ATIVIDADES
13 de 20FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

Opinião do instrutor

Seu código do `LivroDao` deve parecer-se com o abaixo e o `beans.xml` precisa ter habilitado a descoberta de beans.



```
public class LivroDao {  
  
    @PersistenceContext  
    private EntityManager manager;  
  
    public void salvar(Livro livro) {  
        manager.persist(livro);  
    }  
}
```

[COPIAR CÓDIGO](#)

86.0k xp





14 Criando a classe de Livro

PRÓXIMA ATIVIDADE



16%

Na classe livro, crie o mapeamento necessário para o JPA funcionar.

ATIVIDADES
14 de 20FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

Opinião do instrutor

A classe livro é uma Entity do JPA, por isso as anotações de @Entity e @Id são obrigatórias. A anotação

@GeneratedValue(strategy=GenerationType.IDENTITY) serve para deixar o campo id como auto increment



```
@Entity
public class Livro {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    private String titulo;
    @Lob
    private String descricao;
    private BigDecimal preco;
    private Integer numeroPaginas;

    // getter's e setter's
}
```

[COPIAR CÓDIGO](#)

86.1k xp



Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 1 - Atividade 15 Adicionando a configuração da JPA | Alura

Meu caderno do curso (beta)

Este é seu caderno de anotações aqui na Alura. Todo o seu conteúdo fica visível apenas para você. Próximos passos para essa funcionalidade no nosso [FAQ](#)

Uma vez que nossa classe `LivroDao` e `Livro` está completa, podemos criar a configuração do JPA necessária para começar a comunicação com o banco de dados. Vimo que temos que uma pasta chamada META-INF dentro da pasta resources, e dentro desta temos que criar arquivo `persistence.xml`, ele é o responsável pela configuração do jpa, uma api do java para framework de mapeamento-objeto-relacional. Para implementar o JPA usaremos a ferramenta mais popular do mercado, o Hibernate. Então o nosso `persistence.xml` vai ficar assim :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    version="2.1"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="casadocodigo-dev" transaction-type="JTA">
        <description>Dev persistence unit</description>
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <!-- java transaction api || JNDI -->
        <jta-data-source>java:jboss/datasources/casadocodigoDS</jta-data-source>
        <properties>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
        </properties>
    </persistence-unit>
</persistence>
```

Crie o arquivo no seu projeto.

Opinião do instrutor

Nesse arquivo temos 2 novos responsáveis que nos ajuda a facilitar a configuração: O primeiro deles é o JTA,

basicamente ele cuida de cada transação que é feita no sistema, por isso não temos que nos preocupar tanto em abrir uma transação, realizar a operação e commitar, tudo isso é feito pelo JTA facilitando muito a nossa vida.

O outro responsável, é o JNDI, podemos ver a sua configuração na linha que chamamos o JTA:

```
<jta-data-source>java:jboss/datasources/casadocodigoDS</jta-data-source>
```

O parâmetro que passamos nessa tag é o JNDI, basicamente é uma forma de informar ao servidor onde fica o caminho dos serviços que iremos utilizar, nesse caso informamos onde vai ficar a configuração dele.

Mesmo criado o persistence.xml, ainda temos algumas configurações a serem feitas, ainda não passamos os dados do banco de dados para o nosso projeto, especificamente o servidor, o arquivo que armazena esse tipo de arquivo é o standalone-full.xml que iremos configurar em breve.

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 1 - Atividade 16 Configurando o Datasource | Alura

Video Player is loading.

Current Time 0:00

Duration 9:34

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Configurando o Datasource

Uma vez que precisamos de um datasource, vamos criá-lo com base no exemplo e então colocaremos algumas configurações adicionais. Ficando nosso código conforme abaixo:

```
<datasource jndi-name="java:jboss/datasources/casadocodigoDS" pool-name="casadocodigoDS">
    <connection-url>jdbc:mysql://localhost:3306/casadocodigo_javaee</connection-url>
    <connection-property name="DatabaseName">
        casadocodigo_javaee
    </connection-property>
    <driver>mysql</driver>
    <pool>
        <min-pool-size>10</min-pool-size>
        <max-pool-size>20</max-pool-size>
    </pool>
    <security>
        <user-name>root</user-name>
    </security>
</datasource>
```

- Começando pela tag `<datasource>`, temos o atributo `jndi-name`, que é a ligação entre o datasource que estamos criando e a nossa aplicação;
- Mais abaixo, temo a tag `<connection-url>` que recebe o endereço do nosso banco de dados;
- Em seguida temos a tag `<connection-property>`, que obrigatoriamente precisamos adicionar quando usamos o Wildfly 10. Perceba que estamos informando algo que já está na tag `connection-url`, porém não será levado em consideração lá, apenas aqui na tag `<connection-property>`;
- Na tag `<pool>` nós informamos o número mínimo e máximo de conexões que o datasource poderá criar com o banco de dados;
- Por fim, temos a tag `<security>`, onde informamos o nome do usuário do banco de dados na tag `<user-name>`, no nosso caso `root`; Caso você possua uma senha para acessar seu mysql, basta adicionar a tag `<password>` abaixo de `user-name`;

Parece que pulamos uma tag, que foi a `<driver>`. Essa tag referencia outra área específica dentro da configuração, que fica logo abaixo de `<datasource>`, é a tag `<drivers>`. Dentro dela temos a tag `<driver>` de exemplo, porém, assim como em `<datasource>`, precisamos criar a nossa configuração de driver. Ficando nosso código, assim:

```
<driver name="mysql" module="com.mysql">
    <datasource-class>com.mysql.jdbc.jdbc2.optional.MysqlDataSource</datasource-class>
</driver>
```

Essa configuração da tag `<driver>`, serve principalmente para informar a classe do datasource, o qual fazemos através da tag `<datasource-class>` do código acima. Mas ainda dentro da tag, temos um atributo um atributo importante, é o `module`. Todo **driver JDBC**, precisa de um **JAR**. Esse **JAR** não fica armazenado no projeto e sim no servidor, para ser mais específico, no Wildfly os drivers ficam armazenados na pasta: `wildfly/modules/`. Devemos colocar nosso **JAR** dentro desta pasta, com a estrutura a seguir:

```
com/
    mysql/
```

```
main/
    mysql-connector-java-5.1.35.jar
```

Dentro da pasta `main` nós temos que adicionar um arquivo chamado `module.xml` além do **JAR** do driver de conexão. No arquivo `module.xml` temos o caminho para o jar do driver de conexão e uma dependência para a API de persistência do *JDBC* chamada `javax.api`. O `module.xml` ficará assim:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.3" name="com.mysql">
    <resources>
        <resource-root path="mysql-connector-java-5.1.35.jar"/>
    </resources>
    <dependencies>
        <module name="javax.api"/>
    </dependencies>
</module>
```

Agora só falta criar o banco de dados. Abra o seu Terminal e conecte no MySQL com o comando `mysql -u root` (ou utilize uma ferramenta de sua preferência) e crie a base com o comando `create database casadocodigo_javaee;`. Acesse o banco criado com o comando `use casadocodigo_javaee;` e verifique as tabelas existentes usando o `show tables;`. Até aqui, não devemos ter nenhuma tabela na base.

Tudo certo, configuração finalizada. Vamos subir nossa aplicação e depois de subir a aplicação execute o comando `show tables;` novamente. Nesse momento, a tabela `Livro` deve ter sido criada. Compare as colunas usando `desc Livro;` para ver se está de acordo com sua classe. Sua estrutura, deve estar parecida com a seguinte:

```
mysql> desc Livro;
```

Field	Type	Null	Key	Def
id	int(11)	NO	PRI	NUL
descricao	longtext	YES		NUL
numeroPaginas	int(11)	YES		NUL
preco	decimal(19,2)	YES		NUL
titulo	varchar(255)	YES		NUL

```
5 rows in set (0,01 sec)
```

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 1 - Atividade 17 DataSource, module e mais configurações | Alura

Uma vez que nosso `persistence.xml` está na pasta resources/META-INF criado, temos que adicionar as configurações do banco para o servidor. Abra o `standalone-full.xml` do seu Wildfly para adicionar o `datasource` que iremos usar e também o `driver` relacionado ao nosso banco. A estrutura que você precisa é essa:

```
<datasource jndi-name="java:jboss/datasources/casadocodigoDS" pool-name="casadocodigoDS">
    <connection-url>jdbc:mysql://localhost:3306/casadocodigo_javaee</connection-url>
    <connection-property name="DatabaseName">
        casadocodigo_javaee
    </connection-property>
    <driver>mysql</driver>
    <pool>
        <min-pool-size>10</min-pool-size>
        <max-pool-size>20</max-pool-size>
    </pool>
    <security>
        <user-name>root</user-name>
    </security>
</datasource>
<drivers>
    <driver name="mysql" module="com.mysql">
        <datasource-class>com.mysql.jdbc.jdbc2.optional.MysqlDataSource</datasource-class>
    </driver>
</drivers>
```

Lembre de colar logo abaixo (ou logo a cima) do datasource que vem configurado.

E para que o sistema possa pegar as configurações corretas, vamos criar também o arquivo `module.xml` dentro do nosso Wildfly na pasta `modules` crie a estrutura `com > mysql > main` e dentro de `main` o `module.xml`, o seu conteúdo será esse:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.3" name="com.mysql">
    <resources>
        <resource-root path="mysql-connector-java-5.1.35.jar"/>
    </resources>
    <dependencies>
        <module name="javax.api"/>
    </dependencies>
```

```
</module>
```

Além desse xml, colar o mysql-connector-java, que é um jar usado para realizar a conexão com o banco, ele pode ser baixado aqui: <http://central.maven.org/maven2/mysql/mysql-connector-java/5.1.35/mysql-connector-java-5.1.35.jar>

Para finalizar, só precisamos criar o banco de dados, e faça os seguintes comandos:

- acesse o Mysql: no terminal faça : mysql -u root
- create database casadocodigo_javaee; para criar o banco.

Opinião do instrutor

A estrutura da pasta e arquivos modules criado para wildfly comunicar com banco é essa deve ser essa:

Dentro da pasta do wildfly/modules:

```
com/
  mysql/
    main/
      mysql-connector-java-5.1.35.jar
      module.xml
```

Feito tudo isso, o que nos falta é subir a aplicação para garantir que não há problemas, em caso de problemas lembre-se que sempre você poderá usar o fórum para buscar auxílio.

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 1 - Atividade 18 Inserindo os Dados no Banco de Dados | Alura

Video Player is loading.

Current Time 0:00

Duration 11:19

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Inserindo os dados no Banco

Vamos agora voltar para o nosso `AdminLivrosBean` e nele temos o método `salvar` que persiste os nossos dados no banco de dados, para isso precisamos do `LivroDao`. Nós temos aqui a opção de inicializar o `LivroDao`:

```
private LivroDao dao = new LivroDao();
```

Mas desta forma nós estamos dizendo que vamos assumir a responsabilidade de instanciar todos os atributos. Vale lembrar que dentro do `LivroDao` temos o atributo `manager` do tipo `EntityManager` que é injetado pelo servidor. Devemos pedir para que o nosso `LivroDao` seja também injetado e assim o `EntityManager` será criado pelo servidor também. Quem é responsável por isso é o CDI, que cuida do contexto de injeção de dependências. Com o uso do CDI o nosso código ficará dessa forma:

```
@Inject  
private LivroDao dao;
```

E o nosso método salvar da `AdminLivrosBean`:

```
public void salvar(Livro livro) {  
    dao.salvar(livro);  
}
```

Nesse ponto, se você tentou subir a aplicação e cadastrou um *Livro*, recebeu o erro a seguir:

`TransactionRequiredException`. Isso ocorre, porque estamos tentando alterar o estado do banco de dados. E se tentarmos salvar (ou alterar/remover) o nosso livro, vamos receber essa exception. Para tudo que altera o estado do banco, o servidor espera uma transação e nós não temos. Precisamos pedir essa transação para o JTA, o pedido é feito pela annotation `@Transactional` (*import* do pacote `javax.transaction.Transactional`) logo acima do método `salvar`.

```
@Transactional  
public void salvar(Livro livro) {  
    // o código continua intacto aqui  
}
```

Se subirmos nosso servidor agora, nós conseguiremos efetuar o cadastro, faça um select no banco para verificar se o nosso livro foi inserido corretamente. Observe também o *console* do Eclipse, se você encontra um comando `insert` na tabela *Livro*.

Um ponto interessante é que não anotamos a classe `LivroDao` e mesmo assim ele foi injetado. Isso acontece porque o **CDI** consegue descobrir quais são os objetos que podem ser injetados e onde devem ser injetados. Mas como ele descobre quais beans injetar e onde injetar? Vamos abrir o nosso arquivo `beans.xml` localizado em `src/main/webapp /WEB-INF/`, e vemos que não está definido a forma de busca, isso aconteceu devido a criação feita pelo Forge. Isso não nos gerou problema, mas para evitar problemas futuros vamos corrigir logo. Deixando nosso arquivo conforme abaixo:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>  
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       bean-discovery-mode="all" version="1.1"  
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
```

```
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"/>
```

Nesse arquivo, o que mudamos de importante foi o atributo `bean-discovery-mode` que não havia sido adicionado, e informamos o valor `all`, para que o CDI procure por toda aplicação, e não apenas as classes anotadas com `@Named`. E o outro ponto foi o namespace novo do JavaEE 7, que é `http://xmlns.jcp.org/xml/ns/javaee`, onde também informamos que queremos usar a versão **1.1** do CDI.



19 Entendendo o CDI

[PRÓXIMA ATIVIDADE](#)

21%

Qual a vantagem de utilizarmos o CDI no JavaEE? O que ganhamos com seu uso no projeto?

ATIVIDADES
19 de 20

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

MODO
NOTURNO

ABRIR
CADERNO

L

Com o CDI ganhamos injeção de dependências entre as classes da aplicação e também o controle de escopo, que nos permite manter uma classe viva pelo tempo que precisarmos.

Essa alternativa é a correta pois o CDI controla ao escopo e a injeção das dependências das aplicações JavaEE, substituindo os EJBs que antes no JavaEE eram a única forma de fazer isso.

B

Com o CDI ganhamos o controle de escopo de cada objeto isolado. Mas não temos como ligar um objeto ao outro.

Com o CDI ligamos objetos uns ao outros pela injeção de



dependências provida por ele e também controlamos os escopos dos objetos por dependência.



21%

ATIVIDADES
19 de 20

C

O CDI é apenas `annotation` que colocamos em cima das classes

O CDI possui annotations como forma de configuração, porém ele não se resume apenas a isso.

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

**D**

O CDI serve para ligar camadas da aplicação, tornando a comunicação entre as classes melhor, mas ele não conhece nada sobre o ciclo de vida dessas classes.

O CDI também é responsável pelo Ciclo de Vida dos objetos gerenciados por ele. Ele liga as camadas através da injeção de dependências, colocando objetos prontos para serem utilizados nas classes que solicitarem. Assim, ele liga objetos e não camadas.



O CDI se tornou uma ferramenta poderosa no JavaEE, sendo





responsável pela injeção de dependências do projeto. Além disso, também ganhamos o controle de escopo (tempo de vida) que essas dependências ficarão vivas no sistema. O CDI veio com o propósito de possibilitar a quebra dos EJBs como único meio de realizar injeção de dependências, contexto, tempo de vida, dentre outras coisas.



21%

ATIVIDADES
19 de 20

PRÓXIMA ATIVIDADE

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

MODO
NOTURNOABRIR
CADERNO



20 Cadastrando o primeiro Livro

[PRÓXIMA ATIVIDADE](#)

24%

Estamos quase finalizando, vamos apenas fechar os conceitos vistos na nossa aula. A injeção de dependência que precisa ser feita no `AdminLivrosBean` para o `LivroDao` e a transação que precisa ser adicionada.

ATIVIDADES
20 de 20

Vamos fazer isso com `@Inject` e a transação com `@Transactional` no método `salvar` que é onde queremos colocar a nossa transação.

FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

Para que o CDI injete o `LivroDao` ele precisa ser capaz de achar todos os livros, e para isso precisamos adicionar uma configuração no `beans.xml` que temos, para `bean-discovery-mode="all"` e também a versão do CDI `version="1.1"`.

Após toda a configuração pronta, suba o sistema e teste a aplicação.



Opinião do instrutor

A classe `LivroBean` deve ficar assim:



86.5k xp

```
// outros atributos acima  
  
@Inject  
private LivroDao dao;
```





```
@Transactional  
public void salvar() {  
    dao.salvar(livro);  
}  
  
// Demais métodos abaixo
```

[COPIAR CÓDIGO](#)

24%

ATIVIDADES
20 de 20

E o arquivo beans.xml ficará parecido com:

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>  
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       bean-discovery-mode="all" version="1.1"  
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee  
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"/
```

[COPIAR CÓDIGO](#)

86.5k xp



Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 2 - Atividade 1 Criando a relação de Livro com Autor | Alura

Video Player is loading.

Current Time 0:00

Duration 7:54

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Nosso próximo objetivo será permitir o cadastro de um ou mais autores para os livros.

Vamos abrir nosso `form.xhtml`, e inserir mais um campo, com `h:selectManyListbox` que nos permitirá selecionar mais de um autor, como no exemplo abaixo:

```
<div>
    <h:outputLabel value="Autores" />
    <h:selectManyListbox>
        <f:selectItems value="#{adminLivrosBean.autores}"
            var="autor"
            itemValue="#{autor}"
            itemLabel="#{autor.nome} "/>
    </h:selectManyListbox>
</div>
```

Observe os atributos informados na `f:selectItems`, dentro dele temos o atributo `value` que servirá para informar a lista de autores que vem do nosso servidor. Além disso, temos ainda mais dois atributos importantes, o `itemLabel` e também o `itemValue`. O `label` serve para dizer ao JSF qual será o valor exibido para o usuário selecionar, mas esse valor não serve para relacionarmos no nosso banco de dados, por isso temos o `itemValue`, onde diremos qual valor será colocado na nossa entidade. Para `value` e `label` usamos a expression language do JSF com o `#{}auto...{}`, mas de onde ele veio?

Nossa lista de autores, está no formato `List`, mas queremos apenas um único `Autor` dessa lista, assim usamos um outro atributo chamado `var`, que passa para cada elemento da lista, o valor dele para dentro do atributo `var` que será nosso autor.

Já que o `h:selectManyListbox` está recebendo como valor o `adminLivrosBean.autores`, temos que criar em nosso arquivo `AdminLivrosBean` um método chamado `getAutores` que devolve uma lista de Autor. Vamos criá-los.

```
public List<Autor> getAutores() {
}
```

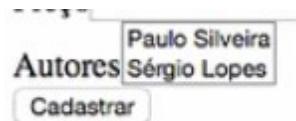
Mas ainda não criamos a entidade `Autor`, vamos cria-la no pacote `br.com.casadocodigo.loja.models`. Nosso autor deverá parecer-se com:

```
public class Autor {
    private Integer id;
    private String nome;
    public Autor(Integer id, String nome) {
        this.id = id;
        this.nome = nome;
    }
    // gera os getters e setters de id e nome
}
```

Voltando para a classe `AdminLivrosBean`, agora conseguimos pegar os autores e retornar para o xhtml.

```
public List<Autor> getAutores() {
    return Arrays.asList(new Autor(1, "Paulo Silveira"), new Autor(2, "Sérgio Lopes"));
}
```

Uma vez que temos nossa lista de autores pronta, já podemos subir nosso servidor para testar. Os autores ainda não estão vindo do banco de dados, porém vamos resolver isso logo em seguida.



Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 2 - Atividade 2 Adicionando autores no Livro | Alura

Meu caderno do curso (beta)

Este é seu caderno de anotações aqui na Alura. Todo o seu conteúdo fica visível apenas para você. Próximos passos para essa funcionalidade no nosso [FAQ](#)

Dica: Caso precise do projeto que foi feito na ultima aula, para seguir com seus estudos daqui, basta baixar o código aqui : <https://github.com/alura-cursos/java-ee-webapp/archive/e865030287b34b48df96d12ef8ad3e0e8b56c8df.zip>

Adicione a `div` e o `<h:selectManyListbox>` referente a seleção dos autores, exibindo o `nome` do autor e relacionando o valor do `select` pelo `id` do autor. Também já vamos colocar o converter `javax.faces.Integer` para que os ID sejam enviados como `Integer` e não `String`.

Lembre-se também de guardar o valor dentro do nosso `AdminLivrosBean` como sendo uma lista de `Integer's`

Opinião do instrutor

Sua `div` como o `h:selectManyListbox` deverá ficar parecido com o código abaixo:

```
<div>
    <h:outputLabel value="Autores" />
    <h:selectManyListbox>
        <f:selectItems value="#{adminLivrosBean.autores}"
            var="autor" itemValue="#{autor.id}" itemLabel="#{autor.nome}" />
    </h:selectManyListbox>
</div>
```



03 Criando o Autor para a busca

PRÓXIMA ATIVIDADE



27%

Veja que para nosso select itens não funcionar, pois no nosso `AdminLivrosBean` não existe os métodos `getAutores` e mais alguns detalhes, então que temos que fazer é :

ATIVIDADES
3 de 12FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

Opinião do instrutor



Então, a sua classe `Autor` é para ficar mais ou menos assim

```
package br.com.casadocodigo.loja.modelo;

public class Autor {
    private Integer id;
    private String nome;
```



86.7k xp





27%

ATIVIDADES
3 de 12FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

86.7k xp

```
public Autor(Integer id, String nome) {
    this.id = id;
    this.nome = nome;
}
public Integer getId() {
    return id;
}
public void setId(Integer id) {
    this.id = id;
}
public String getNome() {
    return nome;
}
public void setNome(String nome) {
    this.nome = nome;
}
```

[COPIAR CÓDIGO](#)

No AdminLivrosBean os métodos getAutores :

```
public List<Autor> getAutores() {
    return Arrays.asList(new Autor(1, "Paulo Silvera"),
}
```

[COPIAR CÓDIGO](#)

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 2 - Atividade 4 Salvando os Autores | Alura

Video Player is loading.

Current Time 0:00

Duration 19:09

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Já conseguimos enviar do bean para a tela os autores, mas ainda não conseguimos fazer o inverso. Precisamos agora ligar os autores da tela e relacioná-los com o nosso bean. Para fazer isso precisamos ter uma forma de pegar os autores. Lembra do nosso formulário? Temos o atributo `itemValue` da `h:selectManyListbox` que nos devolve os `id's` dos Autores, por isso criaremos então uma lista para armazenar esses `id's` que serão enviados do formulário.

```
public class AdminLivrosBean{  
    private List<Integer> autoresId = new ArrayList<>(); // fazemos new para evitar NullPointerException  
  
    // demais atributos e métodos abaixo  
}
```

Vamos alterar o nosso formulário para vincular ao nosso atributo `autoresId`.

```
<h:selectManyListbox value="#{adminLivrosBean.autoresId}">  
    <!-- selectItens aqui no meio, não altere -->  
</h:selectManyListbox>
```

E uma vez que temos essa ligação, podemos alterar o método salvar do `AdminLivrosBean` para *setar* os autores no livro.

```
public void salvar(){  
    for(Integer autorId : autoresId){  
        livro.getAutores().add(new Autor(autorId));  
    }  
  
    dao.salvar();  
    System.out.println("Livro cadastrado com sucesso " + livro);  
}
```

Perceba que nesse momento, o código não está compilando, pois não temos os autores dentro da classe `Livro`. Vamos resolver isso agora, criando um relacionamento entre as entidades `Livro` e `Autor`, assim, vamos abrir a classe `Livro` e adicionar o atributo `Autor`.

```
public class Livro{  
  
    // demais atributos acima  
  
    private List<Autor> autores;  
}
```

Para o JPA somente este atributo não é suficiente para relacionar as entidades `Livro` e `Autor`, precisamos anotar o atributo, mas qual anotação usar? Vamos pensar, quantos autores podemos ter em um livro? Quantos livros um autor pode escrever? A resposta é, um autor pode escrever vários livros e um livro pode ter vários autores, vamos usar a anotação `@ManyToMany`.

```
public class Livro{  
  
    // demais atributos acima  
  
    @ManyToMany  
    private List<Autor> autores = new ArrayList<>();
```

}

O `@ManyToMany` está reclamando que o Autor não é uma entidade, precisamos anotar a classe `Autor` como `Entity`. Um outro ponto importante aqui é que o JPA precisa de um construtor vazio para instanciar a classe, vamos criá-lo também e já remover o atributo `nome` do segundo construtor, pois no `AdminLivrosBean` estamos passando apenas o `ID`. Não esqueça de sobrescrever o `toString()`.

```
@Entity
public class Autor{

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    private String nome;

    public Autor() {}

    public Autor(Integer id) {
        this.id = id;
    }

    // getters e setters abaixo e toString
}
```

Recapitulando o que vai acontecer aqui com toda essa configuração *JPA* realizada. O `ManyToMany`, cria uma tabela auxiliar para armazenar os id's de cada tabela, assim podemos ter vários autores para um livro, e um autor pode escrever vários livros.

Mas até o momento, nossa classe `AdminLivrosBean` só estava criando 2 autores na mão. Vamos alterar isso, para que possamos pegar os autores direto do banco. Para isso vamos criar primeiro a classe `AutorDAO` no pacote `br.com.casadocodigo.loja.daos` com o método que nos retorna a lista de autores.

```
public class AutorDAO {

    @PersistenceContext
    private EntityManager manager;

    public List<Autor> listar(){
        manager.createQuery("select a from Autor a", Autor.class)
            .getResultList();
    }
}
```

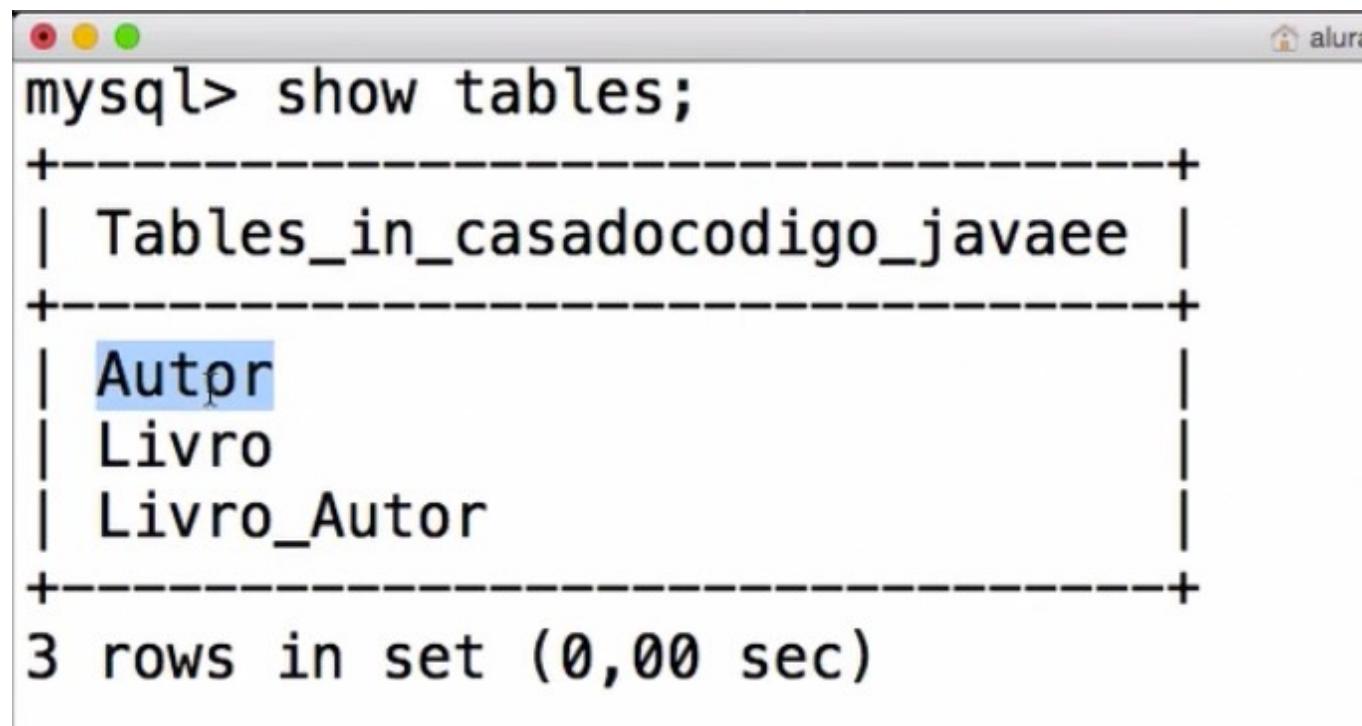
Temos nossa lista de autores, vamos voltar para o bean `AdminLivrosBean` e modificar o método `getAutores()`. Para isso

vamos precisar injetar o `AutorDao`, usando mais uma vez o **CDI** conforme abaixo:

```
public class AdminLivrosBean {  
  
    // demais atributos acima  
    @Inject  
    private AutorDao autorDao;  
  
    public List<Autor> getAutores() {  
        return autorDao.listar();  
    }  
  
    // demais métodos abaixo  
}
```

Agora vamos testar. Não esqueça que não temos nenhum autor cadastrado, desta forma não aparecerá nenhum autor no form. Mas precisamos subir a aplicação para que o *Hibernate* crie a nossa tabela de `Autor`.

Depois de subir o servidor vamos verificar o nosso banco, veja que aparece não só a tabela de `Autor`, mas também a `Livro_Autor` que foi gerado devido a nossa anotação `@ManyToMany`, onde serão armazenado os id's que vinculam a tabela `Livro` e `Autor`. Ao realizar um `show tables;` pelo MySQL temos o resultado:



```
mysql> show tables;  
+-----+  
| Tables_in_casadocodigo_javaee |  
+-----+  
| Autor |  
| Livro |  
| Livro_Autor |  
+-----+  
3 rows in set (0,00 sec)
```

Vamos inserir alguns autores para continuar com nosso teste. Conecte-se ao seu MySQL com sua ferramenta preferida, e execute o seguinte comando:

```
insert into Autor (nome) values ('Paulo Silveira'), ('Sérgio Lopes'), ('Guilherme Silveira'), ('Alberto Souza');
```

Com a tabela autor preenchida, ao selecionarmos um Autor e tentar cadastrar um Livro recebemos um erro: `ClassCastException`. Isso aconteceu por que o JSF está passando os `id's` de autores como `String` e no nosso bean `AdminLivrosBean` esperamos uma lista de `Integer`. Para resolver esse problema, usaremos um `Converter` do JSF que já está pronto para nos ajudar a resolver esse problema. Um converter, serve para informar ao JSF que o valor passado para o bean deve ser convertido para `Integer`, o nome do converter que usaremos é `javax.faces.Integer`.

```
<!-- Adicionamos o converter aqui - CÓDIGO NOVO -->
<h:selectManyListbox value="#{adminLivrosBean.autoresId}" converter="javax.faces.Integer">
    <f:selectItems value="#{adminLivrosBean.autores}"
        var="autor"
        itemValue="#{autor.id}"
        itemLabel="#{autor.nome}" />
</h:selectManyListbox>
```

Vamos subir nossa aplicação após essas alterações e tentar cadastrar um novo livro. Após clicar em salvar verifique pelo console do Eclipse, o log do Hibernate para conferir se aparecem os `inserts` de Livro e Livro_Autor.

Vá até seu cliente de MySQL e faça um select na tabela Livro_Autor para ver o relacionamento entre as duas tabelas:

```
select * from Autor_Livro;.
```

Uma boa prática, é já realizar uma pesquisa no google sobre os converters existentes para o JSF. Temos diversos conversores e vamos falar mais sobre eles logo logo.

Você deve ter observado que após salvar, os dados preenchidos continuam aparecendo no formulário. Vamos limpá-los, e para isso no método salvar vamos limpar os atributos de Livro e de Autor.

```
@Transactional
public void salvar(){
    // código já existente aqui, continue ao final do método

    this.livro = new Livro();
    this.autoresId = new ArrayList<>();
}
```

Realize o *Full Publish* novamente, e tente salvar um novo Livro. Perceba que agora sim, após clicar em Cadastrar, nosso formulário volta a ficar vazio.



05

Criando os ids dos autores no Bean e associando com a view

[PRÓXIMA ATIVIDADE](#)

30%

Vamos fazer agora, que nosso sistema busque os autores do sistema.

ATIVIDADES
5 de 12

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

MODO
NOTURNO

ABRIR
CADERNO

L

- Sabemos que nosso formulário, temos o atributo `itemValue` da `h:selectManyListBox`, que nos devolve os id's dos Autores. Por isso precisamos criar uma lista de inteiros para armazenar no nosso `AdminLivrosBean` os ids que são enviados. A lista vai ser chamar `autoresId`, também gere os seus getters e setters.
- Feito isso, precisamos vincular o atributo `autoresID` no formulário, só precisamos adicionar um `value="#{adminLivrosBean.autoresId}"` no nosso `h:selectManyListBox`.

Opinião do instrutor

Então nosso AdminLivrosBean vai um atributo, um getter e um setter novo:



30%

ATIVIDADES
5 de 12FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARDMODO
NOTURNOABRIR
CADEIRNO

```
private List<Integer> autoresId = new Arr  
  
public List<Integer> getAutoresId() {  
    return autoresId;  
}  
  
public void setAutoresId(List<Integer> au  
    this.autoresId = autoresId;  
}
```

COPIAR CÓDIGO

Enquanto o nosso form.xhtml vai ter apenas um atributo no nosso h:selectManyListBox :

```
<h:selectManyListbox value="#{adminLi  
    <f:selectItems value="#{adminLivr  
        var="autor" itemValue="#{autor.id  
    </h:selectManyListbox>
```

COPIAR CÓDIGO



06 Fazendo um Set de autores em um livro

[PRÓXIMA ATIVIDADE](#)

31%

E uma vez que temos essa ligação, podemos alterar o método salvar do AdminLivrosBean para setar os autores no livro:

ATIVIDADES
6 de 12FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

- No método salvar do AdminLivrosBean precisamos criar uma laço de autoresID, para cada autorID, criamos um novo autor e já vinculamos com um livro.
- O eclipse vai reclamar que não existe um `getAutores` e que não existe a classe `Autor`, você precisa cria-los

O JPA precisa saber que tipo de relação existe entre as entidades `Livro` e `Autor` e esta ultima ainda não está mapeada como uma Entidade:

- No nosso `getAutores` da Classe livros precisamos mapear o tipo de relacionamento.
- A classe `Autor` precisa ser mapeada como uma entidade, com isso precisamos informar para o JPA quem é o ID e que ele será gerado automaticamente pelo banco de dados.



86.9k xp

Opinião do instrutor



Na classe AdminLivrosBean a implementação vai ser essa :



31%

ATIVIDADES
6 de 12

```
private List<Integer> autoresId = new ArrayList<>();  
  
@Transactional  
public void salvar() {  
    for (Integer autorId : autoresId) {  
        livro.getAutores().add(new Autor(autorId));  
    }  
  
    livroDAO.salva(livro);  
}
```

[COPIAR CÓDIGO](#)FÓRUM DO
CURSOE a nossa classe `Autor` vai ficar assim :VOLTAR
PARA
DASHBOARD

86.9k xp

```
package br.com.casadocodigo.loja.modelo;  
  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
  
@Entity  
public class Autor {  
  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private Integer id;  
    private String nome;  
  
    public Autor() {}
```





31%

ATIVIDADES
6 de 12FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

```
public Autor(Integer id) {  
    this.id = id;  
}  
public Integer getId() {  
    return id;  
}  
public void setId(Integer id) {  
    this.id = id;  
}  
public String getNome() {  
    return nome;  
}  
public void setNome(String nome) {  
    this.nome = nome;  
}  
  
@Override  
public String toString() {  
    return "Autor [id=" + id + ", nome=" + nome + "]";  
}
```

[COPIAR CÓDIGO](#)

E para completar a nossa classe Livro iremos adicionar o relacionamento com

Autor :



86.9k xp

```
@Entity  
public class Livro {
```



```
@ManyToMany
```



31%

ATIVIDADES
6 de 12

```
public List<Autor> autores = new ArrayList<>();  
@Override  
public String toString() {  
    return "Livro [id=" + id + ", titulo=" + titulo +  
           ", numeroPaginas=" + numeroPaginas + ",  
}
```

```
public List<Autor> getAutores() {  
    return autores;  
}
```

[COPIAR CÓDIGO](#)FÓRUM DO
CURSO

Fazendo isso perceba que nossa classe `AdminLivrosBean` não vai compilar,
fique tranquilo, já iremos arrumar em seguida

VOLTAR
PARA
DASHBOARD

86.9k xp





07 Consultar Autores do Banco

[PRÓXIMA ATIVIDADE](#)

32%

ATIVIDADES
7 de 12FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

87.0k xp

Veja o `getAutores` do `AdminLivrosBean` :

```
public List<Autor> getAutores() {  
    return Arrays.asList(new Autor(1, "Paulo Silvera"),  
}
```

[COPIAR CÓDIGO](#)

Além dele não estar compilando, o Bean só estava criando 2 autores na mão.

Vamos alterar isso, para que possamos pegar os autores direto do banco. Para isso vamos fazer o seguinte :

- Criar a classe `AutorDao` no pacote `br.com.casadocodigo.loja.daos` com o método que nos retorna a lista de autores.
- O nosso DAO vai precisar de um `EntityManager`, e precisamos fazer com o que o servidor cuide de instanciar esse objeto.
- Lembre-se que nosso método do dao precisa retornar uma lista de autores.
- Injete o DAO no `getAutores` do `AdminLivrosBean`

Feito isso suba o projeto, veja que no nosso formulário vai ter o campo de autores em branco. Você precisa fazer um `insert` no seu MySQL

Opinião do instrutor



O seu `AutorDao` deve ficar parecido com esse :



32%

```
package br.com.casadocodigo.loja.dao;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import br.com.casadocodigo.loja.modelo.Autor;

public class AutorDao {

    @PersistenceContext
    private EntityManager manager;

    public List<Autor> getLista() {
        return manager.createQuery("select a from Autor a")
            .getResultList();
    }
}
```

[COPIAR CÓDIGO](#)

87.0k xp



E o que foi implementado no `AdminLivrosBean` :

```
//pacotes
```



```
@Named  
@RequestScoped  
public class AdminLivrosBean {
```



32%

```
//outros atributos
```

```
@Inject  
private AutorDao autorDao;
```

ATIVIDADES
7 de 12

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

```
public List<Autor> getAutores() {  
    return autorDAO.getLista();  
}  
//outros métodos  
}
```

COPIAR CÓDIGO



87.0k xp





08 Múltiplos Autores

PRÓXIMA ATIVIDADE



32%

ATIVIDADES
8 de 12FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

MODO NOTURNO

ABRIR CADerno



87.0k xp



Se subirmos o formulário agora, tomaremos a seguinte exception :

```
javax.servlet.ServletException: java.lang.ClassCastException:  
java.lang.String cannot be cast to java.lang.Integer Parece que o  
nosso <h:selectManyListbox> precisa de um conversor para Integer . Qual a  
razão de precisarmos do conversor?
```

A

Sempre utilizamos converters quando estamos usando JSF.

Nem sempre é necessário um converter, quando usamos tipos básicos (String, Integer, Boolean, etc) ele não é preciso.



Em uma lista genérica de valores o JSF não converte valores automaticamente, mesmo que os valores sejam de tipos básicos. Como o valor em tela é considerado String , precisamos informar o conversor para o tipo correto que precisamos.

Essa é a resposta exata pois a razão de utilizarmos conversores é exatamente o uso de uma lista.

C

Para converter o valor da lista!



Certo, é para converter, mas por que razão? Essa resposta está incompleta, tente escolher uma opção que esteja mais completa.



32%

ATIVIDADES
8 de 12FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

Todos os valores que são enviados da tela para o *ManagedBean* são enviados como texto. O JSF não consegue converter automaticamente, o que seria possível fazer em uma lista. O `generics` não é identificado pelo JSF, o que nos obriga a fazer uso de conversores que *explicitamente* convertem o valor de texto para inteiro. Assim, sempre que utilizamos uma lista é importante converter os valores para o tipo que precisamos com os `converters` do JSF.

Lembre-se de implementar o converter na sua view, ele vai ficar parecido com isso:

```
<h:selectManyListbox value="#{adminLivrosBean.autoresId}">
    <f:selectItems value="#{adminLivrosBean.autores}">
        var="autor"
        itemValue="#{autor.id}"
        itemLabel="#{autor.nome}" />
</h:selectManyListbox>
```

[COPIAR CÓDIGO](#)[PRÓXIMA ATIVIDADE](#)

87.0k xp





09 Manter o formulário em branco

[PRÓXIMA ATIVIDADE](#)

33%

Temos nosso formulário funcionando, porém, veja que se você salvar um livro o sistema voltar para o mesmo form mas todo preenchido. O que devemos fazer para evitar esse tipo de coisa?

ATIVIDADES
9 de 12

FÓRUM DO CURSO

VOLTAR PARA DASHBOARD



87.1k xp

**A**

Basta criar um novo `Livro` e uma nova lista de `autorID` no método `getAutoresId` do `AdminLivrosBean`

o método `getAutoresId` só chamado quando acessamos a tela do formulário, teria que ser um método que o JSF antes de redirecionar para o `form.xhtml`

Basta criar um novo `Livro` e uma nova lista de `autorID` no método `salvar` do `AdminLivrosBean`

Dessa forma, o JSF associa novos objetos em branco, mantendo esse formulário dessa forma. Seria algo mais ou menos assim :

```
@Transactional
public void salvar() {
    for (Integer autorId : autoresId) {
        livro.getAutores().add(new Autor(autorId));
    }
    livroDAO.salva(livro);
}
```



33%

```
this.livro = new Livro();
this.autoresId = new ArrayList<>();
}
```

[COPIAR CÓDIGO](#)ATIVIDADES
9 de 12FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

C Deleto todos os dados do branco.

Dessa forma, você nunca terá dados na plataforma.

Basta criar um novo `Livro` e uma nova lista de `autorID` no método salva do `AdminLivrosBean`

[PRÓXIMA ATIVIDADE](#)

87.1k xp



Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 2 - Atividade 10 Listando os Livros e Autores | Alura

Video Player is loading.

Current Time 0:00

Duration 19:39

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Nosso próximo passo é conseguir ver os livros e autores cadastrados, vamos criar um novo arquivo dentro de `src/main/webapp/livros`. Basta copiar o `form.xhtml` e colar, modificando o nome do arquivo para `lista.xhtml`. Dentro do arquivo vamos remover todo o `form` dele e aproveitar apenas a estrutura, por isso que copiamos e colamos ele, pois o Eclipse não possui um template default com essa estrutura base. Assim, temos o arquivo `lista.xhtml` apenas com as

tags abaixo:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://xmlns.jcp.org/jsf/core">

    <!-- Código vai aqui -->

</html>
```

Dentro do nosso novo arquivo, vamos adicionar um `h:datatable` que é a tabela de dados onde iremos exibir nossos livros. Nossa tabela de livros tem várias colunas, vamos adicionar a princípio somente a coluna título com a tag `h:column`. Perceba no código abaixo, que o resultado do título precisa de um `livro`, esse livro é o `var` que colocamos no `h:DataTable` e o `livro`, vem da lista de livros do atributo `value`. Assim, temos o seguinte código:

```
<h:DataTable var="livro" value="#{adminListaLivrosBean.livros}">
    <h:column>
        #{livro.titulo}
    </h:column>
</h:DataTable>
```

Observe que assim como fizemos no `h:selectManyListbox`, onde tivemos que informar de onde vem os dados, temos que fazer o mesmo para o nosso `h:datatable`. Vamos precisar criar um novo manage bean e dentro dele criar um método que nos devolva os livros do banco de dados. Essa prática de usar um manage bean por tela é bem comum no desenvolvimento de projetos JSF, cada `xhtml` tem um *ManageBean* por trás. Perceba que no atributo `value`, já usamos o nome `adminListaLivrosBean`, vamos então criar a classe `AdminListaLivrosBean` no pacote `br.com.casadocodigo.loja.beans` que será o nosso Bean.

Como ele é um bean que pertence ao JSF, deveríamos anotá-lo com o `@Named` e para que os dados permaneçam na tela durante o `request`, deveríamos anota-lo com `@RequestScoped`, mas o pessoal do CDI decidiu unificar estas duas anotações em uma só, criando assim a annotation `@Model`. Desta forma nossa bean ficará assim:

```
@Model
public class AdminListaLivrosBean {

    private List<Livro> livros = new ArrayList<>();

    public List<Livro> getLivros() {
        return livros;
    }

}
```

Se você pressionar o **F3** pelo Eclipse, em cima da annotation `@Model`, você perceberá que ela é um *Estereótipo* (ou metadado) para `@RequestScoped` e `@Named`.

Precisamos preencher o nosso atributo `livros`. Que classe consegue manipular os dados do livro no banco? A classe `LivroDao`. Então vamos injeta-la no nosso bean e chamar o método `listar` dentro do método `getLivros()`.

```
@Model
public class AdminListaLivrosBean {

    @Inject
    private LivroDao dao;

    private List<Livro> livros = new ArrayList<>();

    public List<Livro> getLivros() {
        this.livros = dao.listar();

        return livros;
    }
}
```

Opa, o metodo `listar` ainda não existe na nossa classe `LivroDao`, precisamos cria-lo. Abra seu arquivo `LivroDao` e adicione o método a seguir:

```
public List<Livro> listar() {
    String jpql = "select l from Livro l";

    return manager.createQuery(jpql, Livro.class).getResultList();
}
```

Certo, temos agora o nosso método e o *JPQL* informado nos informa que serão retornados todos os livros, mas falta algo, precisamos também dos autores do livro, temos que modificar o nosso JPQL.

Sempre que for necessário relacionar uma tabela com a outra temos que usar o *Join*. E vamos precisar que este *Join*, faça a junção entre `Livro` e `Autores`, ou seja, para cada livro ele deve carregar os autores, vamos usar então o `join fetch`. Lembrando que estamos tratando de **Objetos**, assim usamos `Livro` referenciando a classe e `autores` como sendo o atributo `List<Autor>` dentro da classe `Livro` que está sendo referenciada pelo JPA.

Existe outro problema que precisamos resolver. Como podemos ter mais de um autor para cada livro temos que fazer um `distinct` para distinguir o livro independente da quantidade de autores, ou seja, só trará um único resultado de livro independente da quantidade de autores. Desta forma nosso código ficará assim:

```
public List<Livro> listar() {
    String jpql = "select distinct(l) from Livro l"
```

```
+ " join fetch l.autores";  
  
    return manager.createQuery(jpql, Livro.class).getResultList();  
}
```

Voltando para a classe `AdminListaLivrosBean`, agora não temos mais erros. Vamos rodar a nossa aplicação e testar a lista.

Tudo agora deve estar funcionando, mas estamos exibindo apenas a coluna título, vamos colocar os demais valores. Nossa `dataTable` completa ficará assim:

```
<html xmlns="http://www.w3.org/1999/xhtml"  
      xmlns:h="http://xmlns.jcp.org/jsf/html"  
      xmlns:f="http://xmlns.jcp.org/jsf/core"  
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"> <!-- Novo namespace -->  
  
<h:dataTable var="livro" value="#{adminListaLivrosBean.livros}">  
    <h:column>  
        <f:facet name="header">Título</f:facet>  
        #{livro.titulo}  
    </h:column>  
    <h:column>  
        <f:facet name="header">Descrição</f:facet>  
        #{livro.descricao}  
    </h:column>  
    <h:column>  
        <f:facet name="header">Páginas</f:facet>  
        #{livro.numeroPaginas}  
    </h:column>  
    <h:column>  
        <f:facet name="header">Preço</f:facet>  
        #{livro.preco}  
    </h:column>  
    <h:column>  
        <f:facet name="header">Autores</f:facet>  
        <ui:repeat value="#{livro.autores}" var="autor">  
            #{autor.nome},  
        </ui:repeat>  
    </h:column>  
</h:dataTable>  
  
</html>
```

Alguns detalhes importantes de notar que fizemos na `h:DataTable` acima:

- Aproveitamos e adicionamos a tag `f:facet` com o atributo `name="header"`, que cria um cabeçalho na nossa coluna.

- Observe a coluna de autores, usamos o componente `ui:repeat` que é do *facelets*. Por isso adicionamos um novo *namespace* na tag `html` com o prefixo `ui`.
- Feito isso podemos usar o componente `ui:repeat`, que irá iterar e imprimir todos os autores vinculados a cada livro. O `ui:repeat` é basicamente um `for`, onde o `value` é a lista e o `var` a variável que ficará disponível com o autor dentro do `for`.

Realize mais um *Full Publish* para verificar sua lista de livros com os autores relacionados a ele.

Agora que já temos a listagem de livros, seria bem interessante que após o cadastro de nosso formulário, o usuário fosse direcionado para a listagem, onde ele pode ver o livro que ele acabou de cadastrar.

Vamos voltar para a classe `AdminLivrosBean`, precisaremos modificar o método salvar que atualmente salva o nosso livro no banco e em seguida limpa o formulário, para que redirecione o usuário para nossa lista de livros.

```
@Transactional  
public String salvar() { // Mudamos o tipo de retorno  
    for (Integer autorId : autoresId) {  
        livro.getAutores().add(new Autor(autorId));  
    }  
  
    dao.salvar(livro);  
    System.out.println("Livro Cadastrado: " + livro);  
  
    return "/livros/lista"; // E retornamos a página que o usuário irá sem o .xhtml  
}
```

Observe que mudamos o retorno que antes era `void` para `String`, assim conseguimos dizer ao JSF que envie o usuário de volta para a lista de livros. E no retorno, dizemos o caminho a partir da raiz `/src/main/webapp/`, onde ele irá procurar o arquivo, assim `return "/livros/lista";` está procurando por `/src/main/webapp/livros/lista.xhtml`. Também não precisamos adicionar a extensão do arquivo.

Vamos testar nossa aplicação e percebemos que conseguimos salvar e após salvar somos direcionados para a tela de listar livros. Agora experimente apertar F5 após o cadastro de algum produto.

Veja que o navegador nos questiona se queremos resubmeter o formulário. Isso quer dizer que se confirmarmos, ele vai enviar os dados do produto novamente e teremos o produto recadastrado. Duplicando os produtos, o que não é bom!

O que aconteceu foi que o navegador ainda está guardando os dados do post do formulário. Apesar de ser um problema real, não podemos culpar o navegador, pois este é o funcionamento normal no caso de *methods posts* de formulário. Modificaremos então este comportamento em nossa aplicação.

Resolveremos isto através de recursos do protocolo HTTP chamado de *redirect*. O *redirect* passa um *http status* para o navegador carregar uma outra página e esquecer dos dados da requisição anterior. O *http status* que o navegador recebe é um 302.

Para isso devemos mudar o retorno do método salvar, que além do caminho que já retornava, retornará um parametro informando ao JSF para enviar um *redirect* ao navegado.

```
return "/livros/lista?faces-redirect=true";
```

Vamos testar a aplicação e cadastrar um novo livro. Observe que a URL mudou de `form.xhtml` para `lista.xhtml`, confirmando que houve o redirect.

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 2 - Atividade 11 Criando a tela de lista de livros | Alura

-
- [Sugerir alteração](#)

Nesse exercício vamos criar a tela de listagem dos livros com seus respectivos autores

Crie um novo arquivo chamado `lista.xhtml` dentro da pasta `livros`. Você pode usar o arquivo `form.xhtml` como base para a estrutura inicial.

Dentro do nosso novo arquivo vamos colocar uma `datatable` do JSF pegando todos livros do banco de dados e exibindo os seus dados. Inclusive, já podemos exibir os autores dos livros. Lembre-se que para exibir o valor corretamente sem fazer o `toString` na classe autor devemos usar tag `ui:repeat`

Para que tudo funcione corretamente podemos colocar o `value` da `datatable` para pegar os livros. Mas vamos criar um novo `Bean` conforme vimos na aula. Assim, crie uma nova classe chamada `AdminListaLivrosBean` e dentro dela injete o `LivroDao` para pegar todos os livros. Lembre-se da nova anotação para fazer que o JSF reconheça nosso novo Bean.

O método de listar todos os livros ainda não existe, então vamos criá-lo também para que nossa lista funcione corretamente. Lembre-se de usar `JPQL` para essa listagem e que usamos as queries planejadas da **JPA** com `join fetch` para trazer os autores relacionados com o livro.

Opinião do instrutor

-
- [Sugerir alteração](#)

Você deve ter começado pelo arquivo de `lista.xhtml`, que ao final deve ter ficado parecido com:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:h="http://xmlns.jcp.org/jsf/html"
 xmlns:f="http://xmlns.jcp.org/jsf/core"
 xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
```

```
<h:dataTable var="livro" value="#{adminListaLivrosBean.livros}">
    <h:column>
        <f:facet name="header">Título</f:facet>
        #{livro.titulo}
    </h:column>
    <h:column>
        <f:facet name="header">Descrição</f:facet>
        #{livro.descricao}
    </h:column>
    <h:column>
        <f:facet name="header">Páginas</f:facet>
        #{livro.numeroPaginas}
    </h:column>
    <h:column>
        <f:facet name="header">Preço</f:facet>
        #{livro.preco}
    </h:column>
    <h:column>
        <f:facet name="header">Autores</f:facet>
        <ui:repeat value="#{livro.autores}" var="autor">
            #{autor.nome},
        </ui:repeat>
    </h:column>
</h:dataTable>

</html>
```

Sua nova classe AdminListaLivrosBean deve ter ficado parecida com o resultado abaixo:

```
@Model
public class AdminListaLivrosBean {

    @Inject
    private LivroDao dao;

    private List<Livro> livros = new ArrayList<>();

    public List<Livro> getLivros() {
        this.livros = dao.listar();

        return livros;
    }
}
```

E por fim o `LivroDao`, você deve ter adicionado o método `listar` parecido com o que segue abaixo:

```
public class LivroDao {  
  
    // Mantenha os demais códigos aqui!  
  
    public List<Livro> listar() {  
        String jpql = "select distinct(l) from Livro l "  
            + " join fetch l.autores";  
  
        return manager.createQuery(jpql, Livro.class).getResultList();  
    }  
}
```



12 Usando Transações

[PRÓXIMA ATIVIDADE](#)

37%

Falamos um pouco sobre o uso da annotation `@Transactional` na aula anterior. Vamos relembrar o conceito. Para que ela serve e qual é quem é responsável por ela no JavaEE?

ATIVIDADES
12 de 12

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

MODO
NOTURNO

ABRIR
CADERNO

A

Ativa uma transação, fazendo com que os dados sejam enviados para o banco de dados mesmo se qualquer exception acontecer na aplicação.



A transação apenas salvará os dados no banco de dados e nenhum problema acontecer, caso acontece, a própria JTA fará o *rollback*.



Garante que os dados enviados para o banco de dados serão salvos, abrindo e fechando a transação no momento correto. O JTA é a especificação responsável pelas transações no JavaEE.





37%

ATIVIDADES
12 de 12FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARDMODO
NOTURNOABRIR
CADERNO

Correto, o JTA cuida das transações e abre e fecha na chamada do método, garantindo que tudo que estiver sendo enviado para o banco de dados dentro do método, será guardado ou retornado em caso de erro.

C

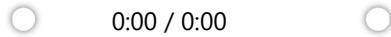
O JTA é o responsável por garantir as transações no JavaEE mas ele não sabe fazer rollback, apenas cuida de abrir e fechar com sucesso, o programador precisa tratar os erros.



Não é bem isso, pois o JTA também cuida do rollback, assim o programador apenas se preocupa em anotar o método, e todo o resto é feito pelo JTA.

A annotation `@Transactional` ativa uma transação que é gerenciada pelo **JTA** (Java Transaction API). Sem uma transação o banco de dados não executará a operação e os dados não serão salvos, o JTA fica responsável por abrir a transação, enviar os dados e fechar, e caso tenha qualquer problema, voltar o banco para o estado anterior. Daí a sua importância para o JavaEE e para o nosso projeto.

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 3 - Atividade 1 Usando o escopo de Flash | Alura



Video Player is loading.

Current Time 8:52

Duration 8:52

1.25x

- 2x
- 1.75x
- 1.5x
- 1.25x, selected
- 1x
- 0.75x
- 0.5x
- 0.25x

Transcrição

Usando o escopo de flash

Ao salvarmos um livro, estamos sendo encaminhados para uma lista:

Título	Descrição	Páginas	Preço	Autores
Java 8 Prático	Java 8 Prático	120	59.00	Paulo Silveira, Guilherme Silveira,
Java 8 Prático 3	Java 8 Prático 3	120	59.00	Paulo Silveira, Sérgio Lopes, Guilherme Silveira,
Java 8 Prático	Java 8 Prático	120	59.00	Sérgio Lopes, Guilherme Silveira, Alberto Souza,
TDD no Mundo Real	TDD no Mundo Real	320	59.00	Sérgio Lopes,
SOLID	Solid do Aniche	240	78.00	Sérgio Lopes, Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,

Após salvar o livro estamos apresentando uma mensagem de sucesso no console do Eclipse, porém ela não é enviada para a tela. Seria bem mais interessante que uma mensagem fosse exibida na tela e que o usuário pudesse visualizá-la. Para isso, depois de salvar o formulário, vamos implementar o envio desta mensagem para a tela do cadastro.

O JSF já tem um objeto responsável por mensagens, que é o `FacesMessages`. Temos que pegar este objeto e adicionar nele a nossa mensagem. Para adicionar uma mensagem no `FacesMessage` vamos precisar fazer uso de um outro objeto do JSF, o `FacesContext`, que é um objeto que nos dá todo o contexto do JSF e nos permitirá adicionar nossa mensagem, assim, usamos o `FacesContext` para adicionar um `FacesMessage`. Abra o bean `AdminLivrosBean` e no método `salvar()` vamos adicionar o trecho a seguir:

```
@Transactional
public String salvar() {
    for (Integre autorID : autoresId) {
        livro.getAutores().add(new Autro(autorID));
    }
    dao.salvar(livro);
    FacesContext.getCurrentInstance()
        .addMessage(null, new FacesMessage("Livro cadastrado com sucesso!"));

    return "/livros/lista?faces-redirect=true";
}
```

Observe que colocamos uma mensagem dentro do `.addMessage()`. Fizemos uso do `FacesContext` chamando o método estático dele

`getCurrentInstance()` e a partir dele, chamamos o `addMessage()` que é onde efetivamente adicionamos o objeto `FacesMessage`.

Em seguida, vamos precisar alterar o arquivo `lista.xhtml`, responsável pela exibição da lista. O JSF tem uma tag chamada `<h:messages />`, que exibe todas as mensagens adicionadas ao `FacesMessage`. Vamos adicionar esta tag logo abaixo da tag `<html>`.

```
<!DOCTYPE html PUBLIC "-//w3c//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:h="http://xmlns.jcp.org/jsf/html"
 xmlns:f="http://xmlns.jcp.org/jsf/coret"
 xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

<h:messages />
```

Após fazermos o *Full Publish* do projeto, se realizarmos um teste agora, perceberemos que a mensagem ainda não é exibida na lista.



Título	Descrição	Páginas	Preço	Autores
Java 8 Prático	Java 8 Prático	120	59.00	Paulo Silveira, Guilherme Silveira,
Java 8 Prático 3	Java 8 Prático 3	120	59.00	Paulo Silveira, Sérgio Lopes, Guilherme Silveira,
Android BÁsico	Android BÁsico	210	59.00	Paulo Silveira,
Java 8 Prático	Java 8 Prático	120	59.00	Sérgio Lopes, Guilherme Silveira, Alberto Souza,
TDD no Mundo Real	TDD no Mundo Real	320	59.00	Sérgio Lopes,
SOLID	Solid do Aniche	240	78.00	Sérgio Lopes, Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 2	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 2	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 2	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 2	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 2	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 2	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 2	120	59.00	Guilherme Silveira,

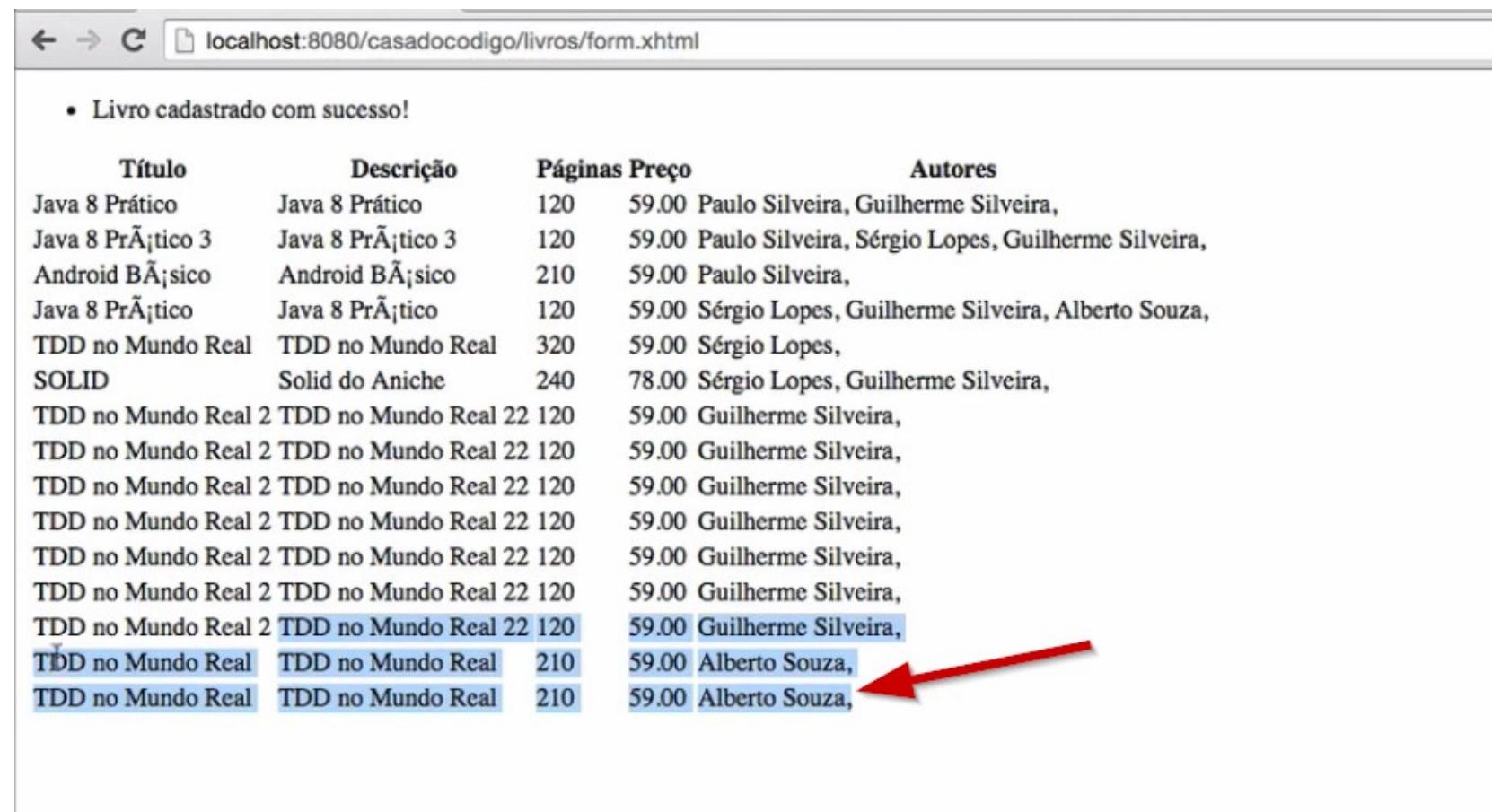
Elá ainda não é exibida, porque toda mensagem do JSF, por padrão é adicionada ao *Request* atual. Como estamos fazendo um `faces-redirect=true`, um novo *Request* é gerado e com isso perdemos a mensagem.

Veremos o que acontece quando removemos o `faces-redirect=true` do nosso `AdminLivrosBean`. Com a alteração, o trecho do código

ficará assim:

```
@Transactional  
public String salvar() {  
    for (Integre autorID : autoresId) {  
        livro.getAutores().add(new Autro(autorID));  
    }  
    dao.salvar(livro);  
    FacesContext.getCurrentInstance()  
        .addMessage(null, new FacesMessage("Livro cadastrado com sucesso!"));  
  
    return "/livros/lista;  
}
```

Dessa vez a mensagem será exibida, no entanto, quando recarregamos a página, voltaremos ao problema anterior.



The screenshot shows a web browser window with the URL `localhost:8080/casadocodigo/livros/form.xhtml`. At the top, there is a success message:

- Livro cadastrado com sucesso!

 Below the message is a table displaying book information. The table has columns: Título, Descrição, Páginas, Preço, and Autores. The data in the table is as follows:

Título	Descrição	Páginas	Preço	Autores
Java 8 Prático	Java 8 Prático	120	59.00	Paulo Silveira, Guilherme Silveira,
Java 8 Prático 3	Java 8 Prático 3	120	59.00	Paulo Silveira, Sérgio Lopes, Guilherme Silveira,
Android Básico	Android Básico	210	59.00	Paulo Silveira,
Java 8 Prático	Java 8 Prático	120	59.00	Sérgio Lopes, Guilherme Silveira, Alberto Souza,
TDD no Mundo Real	TDD no Mundo Real	320	59.00	Sérgio Lopes,
SOLID	Solid do Aniche	240	78.00	Sérgio Lopes, Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 2	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 2	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 2	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 2	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 2	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 2	210	59.00	Alberto Souza,
TDD no Mundo Real	TDD no Mundo Real	210	59.00	Alberto Souza,
TDD no Mundo Real	TDD no Mundo Real	210	59.00	Alberto Souza,

A red arrow points to the last row of the table, specifically to the 'Autores' column of the second-to-last row, which contains the name 'Alberto Souza,'.

Temos registros do livro de TDD do autor `Alberto Souza`. Para resolver esse problema, precisaremos mudar o tipo de mensagem. Vamos fazer com que ela dure mais de um request e com isso, conseguiremos passar a mensagem da tela `form.xhtml` para a

lista.xhtml. O JSF e qualquer framework MVC atual possui um escopo especial (muito rápido) que chamamos de Flash Scope. O Flash Scope começa em um request e termina no request seguinte.

Indo um pouco mais a fundo no Flash Scope, ele consegue aumentar o tempo de vida de um objeto usando a sessão do usuário, adicionando no primeiro request o objeto na sessão e ao fim do segundo request o próprio JSF se encarrega de remover o objeto da sessão. Para fazer uso do Flash Scope, precisamos setar uma informação no contexto do JSF. Dentro do método salvar, faremos o seguinte:

```
// chamada do livroDao.salvar acima
FacesContext.getCurrentInstance().getExternalContext()
    .getFlash().setKeepMessages(true); // Aqui estamos ativando o FlashScope
FacesContext.getCurrentInstance()
    .addMessage(null, new FacesMessage("Livro cadastrado com sucesso!"));
```

Chamamos novamente o FacesContext e pegamos objetos que são externos a ele. Trataremos o contexto com o getFlash(). Usamos o método setKeepMessages(true) para ativar o contexto do Flash. Perceba que a forma como adicionamos a mensagem não mudou, apenas setamos a propriedade keepMessages do objeto Flash que o ExternalContext do JSF nos entregou.

Reiniciamos o servidor e agora nossa mensagem continua, mesmo quando mudamos de tela.

← → C localhost:8080/casadocodigo/livros/lista.xhtml

- Livro cadastrado com sucesso!

Título	Descrição	Páginas	Preço	Autores
Java 8 Prático	Java 8 Prático	120	59.00	Paulo Silveira, Guilherme Silveira,
Java 8 Prático 3	Java 8 Prático 3	120	59.00	Paulo Silveira, Sérgio Lopes, Guilherme Silveira,
Android Básico	Android Básico	210	59.00	Paulo Silveira,
Java 8 Prático	Java 8 Prático	120	59.00	Sérgio Lopes, Guilherme Silveira, Alberto Souza,
TDD no Mundo Real	TDD no Mundo Real	320	59.00	Sérgio Lopes,
SOLID	Solid do Aniche	240	78.00	Sérgio Lopes, Guilherme Silveira,
SOLID 2	Solid 2	400	210.00	Sérgio Lopes, Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real	TDD no Mundo Real	210	59.00	Alberto Souza,
TDD no Mundo Real	TDD no Mundo Real	210	59.00	Alberto Souza,

Nosso objetivo era ter o escopo de Flash na aplicação. Mais adiante, iremos melhorá-la ainda mais.



02 Conhecendo o novo escopo de Flash

[PRÓXIMA ATIVIDADE](#)

40%

ATIVIDADES
2 de 13FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

87.5k xp



Dica: Caso precise do projeto que foi feito na ultima aula, para seguir com seus estudos daqui, basta baixar o código aqui : <https://github.com/alura-cursos/java-ee-webapp/archive/75c4c353f235e8e71d8b1504239179dca18426cf.zip>
[\(https://github.com/alura-cursos/java-ee-webapp/archive/75c4c353f235e8e71d8b1504239179dca18426cf.zip\)](https://github.com/alura-cursos/java-ee-webapp/archive/75c4c353f235e8e71d8b1504239179dca18426cf.zip)

Seu desafio nesse exercício é salvar uma mensagem que dure mais de um request, que chamamos de FlashScope

Para isso, lembre-se do que usamos no vídeo:

- Usar o `FacesContext.getCurrentInstance().addMessage(null, new FacesMessage("Livro cadastrado com sucesso!"));` no método que salva um novo livro.
- Adicionar a mensagem na nossa view, temos que usar uma tag do JSF
- Precisamos aumentar o tempo de vida dessa mensagem, já que por padrão ela dura apenas uma requisição, temos que usar então o FlashScope, usamo algo assim :

```
FacesContext.getCurrentInstance().getExternalContext().getFlash().setKeepMessages(true);
```

Opinião do instrutor

Ao final, seu método `salvar` da classe `AdminLivrosBean` deve ter ficado assim:



40%

ATIVIDADES
2 de 13FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

87.5k xp



```
@Transactional
public String salvar() {
    for (Integer autorId : autoresId) {
        livro.getAutores().add(new Autor(autorId));
    }
    dao.salvar(livro);

    // código novo aqui
    FacesContext.getCurrentInstance().getExternalContext()
        .getFlash().setKeepMessages(true);
    FacesContext.getCurrentInstance()
        .addMessage(null, new FacesMessage("Livro cadastrado"));

    return "/livros/lista?faces-redirect=true";
}
```

[COPIAR CÓDIGO](#)

E na view `lista.xhtml` bastou adicionar o componente `messages`:

```
<!-- Demais declarações acima -->

<h:messages />

<!-- Abaixo a datatable de lista de livros -->
```

[COPIAR CÓDIGO](#)



03

Escopo de Flash



40%

De que serve o Escopo de Flash que temos no JSF?

ATIVIDADES
3 DE 13

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

**A**

Serve para guardar valores no request e que será perdido no segundo request. É apenas para ser usado dentro do mesmo request.

Não exatamente, pois o valor não fica salvo no request e sim na session, sendo removido automaticamente pelo próprio JSF ao final do segundo request.



Serve para guardar valores na sessão, porém duram apenas de um request para o outro, sendo automaticamente retirados da sessão.

Exato, o escopo é controlado pelo JSF que remove logo após a conclusão do segundo request.

C

Serve para guardar valores na sessão do servidor bem rapidamente. Diferente da sessão normal que é bem lenta.

Na verdade o nome `flash` não tem necessariamente haver com velocidade de gravação, mas sim com o tempo que o valor permanecerá guardado. No flash o valor dura apenas um único *request*, enquanto que na sessão normal, fica enquanto o usuário estiver usando a aplicação.



87.5k xp



Serve para guardarmos valores que são temporários, mas que precisam ficar vivos durante um segundo *request*, geralmente feito com `?faces-redirect=true`.

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 3

- Atividade 4 Produzindo o FacesContext | Alura

Video Player is loading.

Current Time 0:00

Duration 7:50

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Pensando um pouco em como melhorar nossa estrutura, veremos que estamos utilizando o `FacesContext` para no contexto externo, manter as mensagens de `flash` configurando o `setKeepMessages` como `true`. Depois, estamos adicionando a mensagem de `Livro cadastrado com sucesso`.

Perceba que em ambos casos, estamos recorrendo ao mesmo contexto. Podemos melhorar isso fazendo com que o contexto seja armazenado em uma variável, em vez de recuperá-lo sempre. Nossa código que antes estava dessa maneira:

```
@Transactional  
public String salvar() {  
    for(Integer autorId : autoresId){  
        livro.getAutores().add(new Autor(autorId));  
    }  
  
    dao.salva(livro);  
  
    FacesContext.getCurrentInstance()  
        .getExternalContext().getFlash().setKeepMessages(true);  
    FacesContext.getCurrentInstance()  
        .addMessage(null, new FacesMessage("Livro Cadastrado com sucesso"));  
  
    return "/livros/lista?faces-redirect=true";  
}
```

Ficará dessa forma:

```
@Transactional  
public String salvar() {  
    for(Integer autorId : autoresId){  
        livro.getAutores().add(new Autor(autorId));  
    }  
  
    dao.salva(livro);  
  
    context = FacesContext.getCurrentInstance();  
    context.getExternalContext().getFlash().setKeepMessages(true);  
    context.addMessage(null, new FacesMessage("Livro Cadastrado com sucesso"));  
  
    return "/livros/lista?faces-redirect=true";  
}
```

Podemos fazer também com que o objeto `context` seja reaproveitável em outros métodos transformando-o em um atributo da classe. Desta forma nosso código que antes criava um novo objeto, apenas faria sua atribuição.

```
context = FacesContext.getCurrentInstance();
```

Para isso precisaremos criar apenas este atributo.

```
public class AdminLivrosBean {  
    private FacesContext context;
```

```
// restante de código  
}
```

O problema de deixar o código exatamente como está agora é precisar utilizar o atributo contexto fora do método `salvar()`. No método `getAutores()` por exemplo, caso utilizássemos o objeto, teríamos problemas porque ele não foi inicializado. Para resolver o problema, moveremos a atribuição do contexto para dentro do construtor da classe da seguinte forma:

```
public class AdminLivrosBean {  
    // código anterior  
  
    public AdminLivrosBean(){  
        context = FacesContext.getCurrentInstance();  
    }  
  
    // restante de código  
}
```

Desta forma, não precisaremos de nenhuma atribuição de contexto dentro dos métodos, visto que o construtor já fará isso automaticamente. Essa solução funciona porém podemos utilizar uma outra, que é pedir para o CDI injetar o objeto para nós. Foi isso que fizemos com os outros objetos. Por que não fazer com este também? Desta forma, descartaremos o construtor e depois, simplesmente anotar o objeto com o `@Inject`.

```
public class AdminLivrosBean {  
    @Inject  
    private FacesContext context;  
    // restante de código  
}
```

O problema é que o CDI e o JSF ainda não estão totalmente integrados e, por isso, a solução não funciona diretamente. Para que funcione, indicaremos para o CDI como criar o objeto que será injetado em nosso código. Por hora, ele não sabe criar o `context` sozinho, mas podemos ensiná-lo.

Vamos criar uma nova classe chamada `FacesContextProducer` no pacote `conf`. Nesta classe, teremos o método `getFacesContext` que retornará uma instância de `FacesContext` para cada `request`, ou seja, no escopo da requisição.

```
br.com.casadocodigo.loja.conf  
  
public class FacesContextProducer{  
  
    @RequestScoped  
    public FacesContext getFacesContext(){  
        return FacesContext.getCurrentInstance();  
    }  
}
```

}

Isto é tudo que a nossa classe que cria objetos de contexto precisa fazer, porém precisamos indicar para o CDI que este método faz exatamente isso: produz um objeto. Para isso utilizamos a anotação `@Produces`.

```
br.com.casadocodigo.loja.conf  
public class FacesContextProducer{  
    @RequestScoped  
    @Produces  
    public FacesContext getFacesContext(){  
        return FacesContext.getCurrentInstance();  
    }  
}
```

Se testarmos agora, veremos que tudo continua funcionando e de forma simples, ganhamos a possibilidade de obter o `FacesContext` em qualquer outra classe da nossa aplicação. Precisaremos apenas declarar o atributo como sendo deste tipo e utilizando a anotação `@Inject`. Isso é injeção de dependência. Não precisamos mais fazer atribuições e também não precisaremos de construtores.



05 Criando o produtor de FacesContext

[PRÓXIMA ATIVIDADE](#)

44%

Nesse exercício você deve permitir injetar o `FacesContext` nos Beans, facilitando o seu uso

ATIVIDADES
5 de 13

Crie o produtor de `FacesContext` no pacote `br.com.casadocodigo.loja.conf`, de modo que o JSF possa se integrar ao CDI e sejamos capazes de usar apenas `@Inject` para obter o `FacesContext`.

FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

Altere a classe `AdminLivrosBean` para injetar o `FacesContext` e dentro do método `salvar` apenas usá-lo.



Opinião do instrutor



Ao final, seu produtor deve ter ficado parecido com:

```
public class FacesContextProducer {  
  
    @RequestScoped  
    @Produces  
    public FacesContext getFacesContext() {  
        return FacesContext.getCurrentInstance();  
    }  
}
```



87.7k xp





}

[COPIAR CÓDIGO](#)

44%

Agora que temos o produtor, você deve ter alterado o método `salvar` da classe `AdminLivrosBean`. Seu método deve ter ficado parecido com:

ATIVIDADES
5 de 13FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD
MODO
NOTURNO
ABRIR
CADERNO

```
@Transactional
public String salvar() {
    for (Integer autorId : autoresId) {
        livro.getAutores().add(new Autor(autorId));
    }
    dao.salvar(livro);

    context.getExternalContext()
        .getFlash().setKeepMessages(true);
    context
        .addMessage(null, new FacesMessage("Livro cadastrado"));

    return "/livros/lista?faces-redirect=true";
}
```

[COPIAR CÓDIGO](#)

87.7k xp



Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 3

- Atividade 6 Validando nosso formulário | Alura

Video Player is loading.

Current Time 0:00

Duration 7:12

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Validando nosso formulário

Vamos realizar um teste, entre no formulário e tente cadastrar um livro selecionando apenas os autores. O que acontece? Exato, o cadastro é feito. E não faz muito sentido cadastrar um livro sem título, páginas e até o preço.

Precisamos melhorar isso.

Abra o arquivo `form.xhtml`, e vamos olhar nosso form. Para cada `input` na tela, temos os componentes `<h:inputText>` no código, e este componente possui um atributo, que ainda não usamos, chamado `required`. Quando setamos o valor dele para `true`, ele torna o campo obrigatório. Com isso conseguimos impedir o envio do formulário sem os campos necessários preenchidos. Mas como o usuário vai saber qual campo ele precisa preencher?

Vamos precisar também de um `<h:messages />` para que o usuário saiba o que foi validado e possa preencher o campo obrigatório com o valor necessário.

Mude no `form.xhtml` apenas o `<h:inputText />` do título conforme abaixo, suba seu servidor e tente cadastrar o livro *sem informar o título*.

```
<h:inputText value="#{adminLivrosBean.livro.titulo}" required="true" />
```

Perceba que de fato, o campo foi validado, mas a mensagem exibida não foi tão amigável, pois `j_idt2:j_id7: Erro de validação: o valor é necessário.` não parece fazer muito sentido. E que valor é necessário?

O componente `<h:inputText />` tem um outro atributo chamado `requiredMessage` que nos permite definir a mensagem que devemos exibir para o usuário, assim, podemos ser específicos sobre a obrigatoriedade do campo.

Além disso, existem campos que precisam de uma validação específica. Por exemplo, os campos `preço` e `número de páginas`. Imagine cadastrar um Livro com páginas 0 (zero)? Faz sentido? Ou mesmo, podemos definir um valor mínimo para o preço. Para isso usamos outra tag do `jsf core` chamada `<f:validateLongeRange />` e `<f:validateDoubleRange />`. Como os nomes já indicam, `validateLongeRange` é para valores do tipo `Long` e `validateDoubleRange` para valores do tipo `Double` / `BigDecimal`.

Vamos alterar nosso código, deixando o formulário como a seguir:

```
<h:form>
    <h:messages />
    <div>
        <h:outputLabel value="Título" />
        <h:inputText value="#{adminLivrosBean.livro.titulo}"
                    required="true" requiredMessage="O Título é um campo obrigatório!" />
    </div>
    <div>
        <h:outputLabel value="Descrição"/>
        <h:inputTextare rows="4" cols="20" requiredMessage="A Descrição é Obrigatória"
                      required="true" value="#{adminLivrosBean.livro.descricao}" />
    </div>
    <div>
        <h:outputLabel value="Número de Páginas"/>
```

```
<h:inputText value="#{adminLivrosBean.livro.numeroPaginas}"  
             required="true" requiredMessage="O Número de Páginas é Obrigatório">  
    <f:validateLongRange minimum="80" />  
</h:inputText>  
</div>  
<div>  
    <h:outputLabel value="Preço"/>  
    <h:inputText value="#{adminLivrosBean.livro.preco}"  
                 required="true" requiredMessage="O Preço é Obrigatório">  
        <f:validateDoubleRange minimum="20" maximum="150" />  
    </h:inputText>  
</div>  
<div>  
    <h:outputLabel value="Autores" />  
    <h:selectManyListbox value="#{adminLivrosBean.autoresId}"  
                          converter="javax.faces.Integer">  
        <f:selectItems value="#{adminLivrosBean.autores}"  
                      var="autor"  
                      itemValue="#{autor.id}" itemLabel="#{autor.nome}" />  
    </h:selectManyListbox>  
</div>  
    <h:commandButton value="Cadastrar" action="#{adminLivrosBean.salvar}" />  
</h:form>
```

Se tentarmos enviar o formulário vazio agora, receberemos as mensagens que definimos pelo `requiredMessage` e também as validações de tamanho. Agora nosso formulário só é enviado se preenchermos corretamente todos os campos.

Realize o *Full Publish* novamente e teste se sua aplicação está funcionando corretamente.

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 3

- Atividade 7 Message Bundles | Alura

Message Bundles

As mensagens até que estão ok, se não preenchermos o título a mensagem será amigável já que definimos a mensagem, mas qual é a mensagem que aparece se colocarmos o valor 10 no campo preço? Aparece uma mensagem nada amigável. Um outro problema que podemos ter é quando tivermos um formulário muito grande teremos que colocar requiredMessage em todos os campos. Para isso existem os arquivos de mensagem que são os message bundles que podem ser definidos em um local e usados na aplicação toda.

Vamos criar o arquivo `jsf_messages.properties` dentro da pasta `/src/main/resources`. Com o seguinte conteúdo:

```
javax.faces.component.UIInput.REQUIRED={0}: Campo Obrigatório
javax.faces.converter.IntegerConverter.INTEGER="{2}" deve ser um número inteiro
javax.faces.converter.BigDecimalConverter.DECIMAL="{2}" deve ser um valor separado apenas por ","
```

Vamos entender o que está acontecendo no código acima. Começando pelo `javax.faces.component.UIInput` que se refere ao `<h:inputText />`. Estamos dizendo que quando o ele não for preenchido a mensagem a frente deve ser lançada. O mesmo acontece para as demais propriedades, `javax.faces.converter.IntegerConverter.INTEGER` e `javax.faces.converter.BigDecimalConverter.DECIMAL` que será lançado quando o JSF não conseguir converter o valor digitado pelo usuário para esses tipos.

Outro ponto interessante são os número entre chaves. O valor `{0}` é o nome do campo que foi validado, o valor `{2}` é o valor do campo.

Agora precisamos falar para o JSF que este arquivo deve ser usado como nosso arquivo de mensagens, isso é definido no arquivo `faces-config.xml` que fica dentro da pasta `src/main/webapp/WEB-INF/`.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" vers

    <application>
        <message-bundle>jsf_messages</message-bundle>
    </application>

</faces-config>
```

Adicionamos ao nosso `faces-config.xml` as tags `<application>` e `<message-bundle>` onde informamos o nome do nosso *properties* de mensagens.

Voltando para nosso `form.xhtml`, vamos remover todos os `requiredMessage` de nosso formulário já que as mensagens ficarão por conta do JSF. Faça novamente um *Full Publish* e tente cadastrar um Livro, deixando os campos em branco e colocando valores inválidos nos campos numéricos.

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 3

- Atividade 8 Validação do formulário | Alura

Meu caderno do curso (beta)

Este é seu caderno de anotações aqui na Alura. Todo o seu conteúdo fica visível apenas para você. Próximos passos para essa funcionalidade no nosso [FAQ](#)

Nesse exercício você deve validar o formulário, não permitindo valores inválidos e tratando as mensagens mais comuns no arquivo de mensagens

- Lembre-se de lançar um `<h:messages />` no `form.xhtml`
- Para validar um campo, use o atributo `required` do **HTML5** nos campos do nosso formulário no arquivo `form.xhtml`.
- Lembre-se de usar `<f:validateLongeRange />` e `<f:validateDoubleRange />` para validar os campos numéricos

Além disso, crie o arquivo de mensagens do **JSF** conforme vimos na aula, com o nome `jsf_messages.properties` dentro da pasta `/src/main/resources`. Lembre-se de colocar as mensagens mais comuns para serem validadas de forma genérica, conteúdo do nosso `properties` deve ser mais ou menos dessa forma:

```
javax.faces.component.UIInput.REQUIRED={0}: Campo Obrigato\u0301rio  
javax.faces.converter.IntegerConverter.INTEGER="{2}" deve ser um número inteiro  
javax.faces.converter.BigDecimalConverter.DECIMAL="{2}" deve ser um valor separado apenas por ".."
```

Para que nosso arquivo de mensagens funcione, temos que adicioná-lo no `faces-config.xml`. Abra o arquivo e coloque a configuração abaixo:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>  
<faces-config xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" vers  
  
    <application>  
        <message-bundle>jsf_messages</message-bundle>  
    </application>  
  
</faces-config>
```

Opinião do instrutor

Nosso arquivo de formulário (`form.xhtml`) deve ter ficado parecido com código abaixo:

```
<h:form>
    <h:messages />
    <div>
        <h:outputLabel value="Título" />
        <h:inputText value="#{adminLivrosBean.livro.titulo}"
            required="true" requiredMessage="O Título é um campo obrigatório!" />
    </div>
    <div>
        <h:outputLabel value="Descrição"/>
        <h:inputTextarea rows="4" cols="20" requiredMessage="A Descrição é Obrigatória"
            required="true" value="#{adminLivrosBean.livro.descricao}" />
    </div>
    <div>
        <h:outputLabel value="Número de Páginas"/>
        <h:inputText value="#{adminLivrosBean.livro.numeroPaginas}"
            required="true" requiredMessage="O Número de Páginas é Obrigatório">
            <f:validateLongRange minimum="80" />
        </h:inputText>
    </div>
    <div>
        <h:outputLabel value="Preço"/>
        <h:inputText value="#{adminLivrosBean.livro.preco}"
            required="true" requiredMessage="O Preço é Obrigatório">
            <f:validateDoubleRange minimum="20" maximum="150" />
        </h:inputText>
    </div>
    <div>
        <h:outputLabel value="Autores" />
        <h:selectManyListbox value="#{adminLivrosBean.autoresId}"
            converter="javax.faces.Integer">
            <f:selectItems value="#{adminLivrosBean.autores}"
                var="autor"
                itemValue="#{autor.id}" itemLabel="#{autor.nome}" />
        </h:selectManyListbox>
    </div>
    <h:commandButton value="Cadastrar" action="#{adminLivrosBean.salvar}" />
</h:form>
```

Após ter criado o arquivo de mensagem `jsf_messages.properties`, dentro dele você deve ter as seguintes chaves:

```
javax.faces.component.UIInput.REQUIRED={0}: Campo Obrigatório
javax.faces.converter.IntegerConverter.INTEGER="{2}" deve ser um número inteiro
javax.faces.converter.BigDecimalConverter.DECIMAL="{2}" deve ser um valor separado apenas por ".."
```

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 3 - Atividade 9 Usando a Bean Validation | Alura

Video Player is loading.

Current Time 0:00

Duration 13:46

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Usando a Bean Validation

Mas vamos pensar com calma. O que realmente queremos validar no título, só queremos validar se ele é obrigatório? Imagine outro cenário em que o usuário só preenche um espaço para driblar nossa validação. Como resolvemos isso?

Faremos então uma validação específica que o Java EE já possui e que é feita diretamente na classe Java, esta validação é chamada de **Bean Validation**. A `Bean Validation` é uma especificação do JavaEE e toda especificação precisa ter uma implementação, o **WildFly** já tem um framework que implementa esta especificação internamente no próprio servidor, que é o **Hibernate Validator**.

Usando o **Hibernate Validator**, quando queremos dizer que o Título não pode ser nulo, nem vazio e nem possuir espaços em branco, usamos a annotation `@NotNull` em cima do atributo.

Porém, existem outras coisas que queremos validar, por exemplo, o valor mínimo aceito pelo preço e número de páginas, o tamanho mínimo preenchido para o campo descrição, e por aí vai. Para vários desses casos, temos annotations específicas da **Bean Validation**, e para os casos não cobertos pela **Bean Validation** o **Hibernate Validator** veio para cobrir.

Vamos modificar a nossa classe para usar as anotações de validação:

```
@Entity
public class Livro {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    @NotBlank // Valida já vazio e espeços em branco
    private String titulo;

    @Lob
    @Length(min=10) // Número mínimo de caracteres que o campo pode ter
    @NotBlank
    private String descricao;

    @DecimalMin("20") // Valor decimal mínimo
    private BigDecimal preco;

    @Min(50) // Valor inteiro mínimo
    private Integer numeroPaginas;

    @ManyToMany
    @Size(min=1) // número mínimo de elementos na lista
    @NotNull // A lista não pode ser nula
    private List<Autor> autores = new ArrayList<>();

    // getters e setters abaixo

}
```

Faça *Full Publish* e teste nossa aplicação. Observe que as mensagens já estão em português porque o próprio

Hibernate Validator já consegue transformar. Tente cadastrar também espaço em branco e veja como o formulário se comporta.

Um problema que encontramos agora é que a mensagem não está informando o campo a que ela se refere. Vamos melhorar isso. Abra o arquivo `jsf_messages.properties` que criamos e vamos adicionar mais uma linha:

```
# Mantenha as demais mensagens acima.  
javax.faces.validator.BeanValidator.MESSAGE={0}
```

Ao informar `{0}` estamos informando que só queremos exibir a mensagem. Mas ainda assim teríamos o problema de ter a mensagem e não sabermos o campo. Para resolver este problema, uma prática muito usada no mercado é exibir a mensagem na frente do próprio campo, assim fica mais claro para o usuário o que aconteceu no campo que ele preencheu. Vamos voltar ao nosso formulário e adicionar um `<message>` logo após o `input`.

```
<div>  
    <h:outputLabel value="Título" />  
    <h:inputText value="#{adminLivrosBean.livro.titulo}"  
        required="true" />  
    <h:message for="titulo" />  
</div>
```

Vamos tratar primeiro o campo título. Adicionamos um componente `<h:message />` para exibir somente uma mensagem de erro, que é diferente do componente `<h:messages />` (plural) que exibe uma lista de erros. A outra diferença é no atributo `for` do componente `<h:message>`, onde vinculamos a mensagem ao campo a que ele se refere. Mas falta uma coisa: quem é título?

Perceba que no `input` não dizemos o nome dele, e o JSF precisa identificar o `input` para colocar a mensagem correta no campo. Obtemos esse resultado com o atributo `id`. Praticamente toda tag do JSF possui o atributo `id`, e no caso dos inputs, também servem para referenciar a mensagem com o `input` dela. Assim, o código completo do campo Título será:

```
<div>  
    <h:outputLabel value="Título" />  
    <h:inputText value="#{adminLivrosBean.livro.titulo}"  
        required="true" id="titulo" />  
    <h:message for="titulo" />  
</div>
```

Faça o mesmo para os demais campos do nosso formulário.

Agora já não precisamos mais do componente `<h:messages />` que ficava no início do formulário, já que estamos apresentando a mensagem campo a campo, e não precisamos mais das validações específicas para número de páginas e preço, já que a **Bean Validation** está cuidando disso.

Se você testar a aplicação nesse momento, perceberá que todas as validações funcionam, mas quando não selecionamos nenhum autor, recebemos um erro `ConstraintValidationException`. O que está acontecendo é que a nossa classe Livro não está vinculada diretamente com o Autor e sim com uma lista de Integers, que são os ids dos autores. Apenas no momento de salvar de fato no banco de dados, é que ocorre a validação e por isso recebemos um erro tão feio, ao invés das validações elegantes que já temos nos demais campos. Mas vamos tratar deste caso mais a frente em nosso curso.

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 3 - Atividade 10 Utilizando a Bean Validation para uma validação baseada na classe | Alura

Meu caderno do curso (beta)

Este é seu caderno de anotações aqui na Alura. Todo o seu conteúdo fica visível apenas para você. Próximos passos para essa funcionalidade no nosso [FAQ](#)

Nesse desafio, você precisa validar o mesmo formulário, porém com regras baseadas na entidade

Como queremos reaproveitar a validação para vários locais onde estamos usando o `Livro`, vamos fazê-la na classe `Livro`. Coloque nos atributos as validações correspondentes da *Bean Validation* ou mesmo específicas da *Hibernate Validator*.

Além disso, vamos precisar alterar o formulário para que ele seja validado, porém removendo as mensagens e validações específicas do JSF e permitindo que tudo seja feito pela *Bean Validation*. Vamos deixar apenas o atributo `required="true"`.

Por fim, você deve colocar um atributo `id` em cada campo do formulário para que possamos usá-lo com o componente `<h:message/>`, e já o adicionar para cada elemento do formulário.

Opinião do instrutor

Na sua classe `Livro` você deve ter adicionado as *annotations* da *Bean Validation* e da *Hibernate Validator* parecida com as que segue:

```
@Entity  
public class Livro {  
  
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private Integer id;  
  
    @NotBlank  
    private String titulo;
```

```
@Lob  
@Length(min=10)  
@NotBlank  
private String descricao;  
  
@DecimalMin("20")  
private BigDecimal preco;  
  
@Min(50)  
private Integer numeroPaginas;  
  
@ManyToMany  
@Size(min=1)  
@NotNull  
private List<Autor> autores = new ArrayList<>();  
  
// getter's e setter's abaixo  
  
}
```

Agora que nosso `Livro` já possui as regras de validação, seu `form.xhtml` deve ter ficado parecido com código abaixo:

```
<h:form>  
    <div>  
        <h:outputLabel value="Título" />  
        <h:inputText value="#{adminLivrosBean.livro.titulo}"  
            required="true" id="titulo" />  
        <h:message for="titulo" /><!-- ADICIONAMOS MENSAGEM ESPECÍFICA PARA CADA CAMPO -->  
    </div>  
    <div>  
        <h:outputLabel value="Descrição"/>  
        <h:inputTextarea rows="4" cols="20"  
            required="true" value="#{adminLivrosBean.livro.descricao}"  
            id="descricao" />  
        <h:message for="descricao" />  
    </div>  
    <div>  
        <h:outputLabel value="Número de Páginas"/>  
        <h:inputText value="#{adminLivrosBean.livro.numeroPaginas}"  
            required="true" id="numeroPaginas" />  
        <h:message for="numeroPaginas" />  
    </div>  
    <div>  
        <h:outputLabel value="Preço"/>  
        <h:inputText value="#{adminLivrosBean.livro.preco}"  
            required="true" id="preco" />  
        <h:message for="preco" />  
    </div>
```

```
<div>
    <h:outputLabel value="Autores" />
    <h:selectManyListbox      value="#{adminLivrosBean.autoresId}"
        converter="javax.faces.Integer" id="autores">
        <f:selectItems value="#{adminLivrosBean.autores}"
            var="autor"
            itemValue="#{autor.id}" itemLabel="#{autor.nome}" />
    </h:selectManyListbox>
    <h:message for="autores" />
</div>
<h:commandButton value="Cadastrar" action="#{adminLivrosBean.salvar}" />
</h:form>
```

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 3

- Atividade 11 Melhorando as mensagens da Bean Validation | Alura

Melhorando as mensagens da Bean Validation

Vamos terminar os ajustes na apresentação das mensagens em nosso formulário, abra o arquivo `jsf_messages.properties` e perceba que antes, sempre mostrávamos para qual campos era feita a validação. Não precisamos mais informar o campo ao exibirmos um erro de campo obrigatório, já que a mensagem está a frente do campo, então vamos retirar o `{0}` de nossa configuração. Ficando apenas a mensagem.

```
# Mantenha as demais mensagens
javax.faces.component.UIInput.REQUIRED=Campo Obrigato\u0301rio
```

As mensagens ligadas aos componentes do JSF estão melhores, porém as mensagens de validação da **Bean Validation** ainda estão genéricas. Seria muito bom poder deixá-las mais específicas para nosso sistema. Lembrando que elas só aparecem em português por que o **Hibernate Validator** possui internacionalização para várias línguas, porém mesmo assim, ainda não está como gostaríamos.

Vamos acessar o endereço <http://bit.ly/1DsZKvf>. Nesse endereço, encontramos o arquivo `.properties` do **Hibernate Validator**. Como queremos exatamente esses nomes, vamos copiar todas as mensagens e salva-las em um novo arquivo dentro da pasta `src/main/resources/`.

Uma vez dentro do `github`, clique na opção `Raw` para ver só o arquivo texto, sem a interface do Github e copie todo o conteúdo. Volte para o Eclipse e dentro de `resources`, clique com o botão direito, vá em `New > File`, e coloque o nome do arquivo de `ValidationMessages.properties`. Agora cole o conteúdo obtido do `Github` dentro desse novo arquivo.

Dê uma navegada neste arquivo e veja que todas as nossas mensagens já estão prontas, mensagens de mínimo, máximo, notnull e muito mais. Temos inclusive mensagens para validação de CPF, CNPJ e Título Eleitoral. Podemos altera-las ou inserir novas de acordo com a nossa necessidade.

Assim, encerramos nossa aula de *Validação e Mensagens*. Vá para os exercícios e bons estudos.



12 Formas de validação do JavaEE

[PRÓXIMA ATIVIDADE](#)

51%

Qual a diferença entre validar os dados da sua aplicação no JSF ou no Bean Validation?

ATIVIDADES
12 de 13

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD


MODO
NOTURNO


ABRIR
CADERNO

**A**

A Bean Validation é limitada a validação de campos requerido, o que torna a Bean Validation muito mais limitada que as validações do JSF.

É exatamente o contrário. Com a Bean Validation temos uma gama muito maior de validações e possibilidades, enquanto que no JSF ficamos um pouco mais limitados.

O JSF valida os componentes de uma forma mais simples, enquanto a Bean Validation consegue tipos de validações mais completos e por ser ligado a entidade, mais facilmente reaproveitável.

Correto! A Bean Validation torna as validações reaproveitáveis por estarem na entidade e também



possui tipos de validações mais complexos como CPF/CNPJ, Boleto, Cartão de Crédito e etc.



51%

C A validação do JSF é reaproveitável, já que o componente pode ser copiado e colado para outra tela, tornando a validação do JSF mais flexível.

ATIVIDADES
12 de 13

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD


MODO
NOTURNO


ABRIR
CADERNO



Copiar e colar não pode ser considerado flexibilidade. A Bean Validation possui maior flexibilidade pois podemos reutilizar a entidade em várias partes do sistema sem precisar de *copy/paste*.

Validação no JSF é ligada ao componente porém possui algumas limitações a tipos de validações que o JSF já possui, enquanto que na Bean Validation temos uma variedade maior de validações, inclusive para regras de negócio como CPF, Título de Eleitor e até número do cartão de crédito. Por isso a Bean Validation acaba sendo mais poderosa e você ainda fica desacoplado do JSF.

[PRÓXIMA ATIVIDADE](#)



13 Melhorando as mensagens da Bean Validation

[PRÓXIMA ATIVIDADE](#)

54%

Neste exercício seu desafio é fazer as mensagens da Bean Validation ficarem mais específicas

ATIVIDADES
13 de 13FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

Para isso, devemos criar o arquivo `ValidationMessages.properties` dentro da pasta `src/main/resources/`. O conteúdo dele pode ser obtido no endereço <http://bit.ly/1DsZKvf> (<http://bit.ly/1DsZKvf>)

Também vamos modificar nosso arquivo `jsf_messages.properties` para remover o ID do campo, já que nossas mensagens estão logo a frente do campo que pode causar o erro.



Opinião do instrutor



Seu arquivo de mensagens da *Bean Validation* deve ficar parecido com:

```
javax.validation.constraints.AssertFalse.message = must be
javax.validation.constraints.AssertTrue.message = must be
javax.validation.constraints.DecimalMax.message = must be
javax.validation.constraints.DecimalMin.message = Deve se
javax.validation.constraints.Digits.message = numeric
javax.validation.constraints.Future.message = must be
```



88.1k xp





54%

ATIVIDADES
13 de 13FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

88.1k xp



```
javax.validation.constraints.Max.message      = must be
javax.validation.constraints.Min.message     = must be
javax.validation.constraints.NotNull.message = may not
javax.validation.constraints.Null.message    = must be
javax.validation.constraints.Past.message   = must be
javax.validation.constraints.Pattern.message= must be
javax.validation.constraints.Size.message   = size mi
```

```
org.hibernate.validator.constraints.CreditCardNumber.message
org.hibernate.validator.constraints.EAN.message
org.hibernate.validator.constraints.Email.message
org.hibernate.validator.constraints.Length.message
org.hibernate.validator.constraints.LuhnCheck.message
org.hibernate.validator.constraints.Mod10Check.message
org.hibernate.validator.constraints.Mod11Check.message
org.hibernate.validator.constraints.ModCheck.message
org.hibernate.validator.constraints.NotBlank.message
org.hibernate.validator.constraints.NotEmpty.message
org.hibernate.validator.constraints.ParametersScriptAssert
org.hibernate.validator.constraints.Range.message
org.hibernate.validator.constraints.SafeHtml.message
org.hibernate.validator.constraints.ScriptAssert.message
org.hibernate.validator.constraints.URL.message

org.hibernate.validator.constraints.br.CNPJ.message
org.hibernate.validator.constraints.br.CPF.message
org.hibernate.validator.constraints.br.TituloEleitoral.message
```

[COPIAR CÓDIGO](#)

Seu arquivo `jsf_messages.properties` deve ficar como abaixo:



```
javax.faces.component.UIInput.REQUIRED=Campo Obrigato\u00e3o  
javax.faces.converter.IntegerConverter.INTEGER="{2}" deve  
javax.faces.converter.BigDecimalConverter.DECIMAL="{2}" de  
javax.faces.validator.BeanValidator.MESSAGE={0}
```



54%

[COPIAR CÓDIGO](#)

ATIVIDADES
13 de 13

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD



88.1k xp



Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 4

- Atividade 1 Adicionando a Data de Publicação | Alura

Video Player is loading.

Current Time 0:00

Duration 7:49

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Adicionando a Data de Publicação

Vamos adicionar mais uma informação no cadastro que atualmente está assim:

The screenshot shows a web-based form for adding a book. The URL in the browser bar is `localhost:8080/casadocodigo/livros/form.xhtml`. The form includes fields for Title, Description, Number of Pages, Price, and Authors. The Authors field is a dropdown menu with four options: Paulo Silveira, Sérgio Lopes, Guilherme Silveira, and Alberto Souza. A 'Cadastrar' button is visible at the bottom left of the authors section.

Dentro do nosso arquivo `Livro.java`, criaremos mais um atributo no `Livro`: Abra a classe `Livro` e logo após o atributo `numeroPaginas` insira o atributo `dataPublicacao` que será do tipo `Calendar`. Lembrando que o **JPA** precisa entender o tipo de data que queremos gravar, se é *DataHora*, só *Hora* ou só *Data*. Fazemos isso com a annotation `@Temporal` informando o `TemporalType`. No nosso caso, será apenas a data, então usaremos assim:

```
@DecimalMin("20")
private BigDecimal preco;
@Min(50)
private Integer numeroPaginas;

@Temporal(TemporalType.DATE)
private Calendar dataPublicacao;
```

Selecionaremos `dataPublicacao` e com o comando `Control + 1`, vamos gerar os `getters` e `setters`, que serão adicionados na parte de baixo do arquivo, depois do `toString()`:

```
@Override
public String toString() {
    return "Livro [id=" + id + ", titulo=" + titulo + ", descricao=" + descricao + ", preco=" + preco
           + ", numeroPaginas=" + numeroPaginas + ", autores=" + autores + "]";
}
public Calendar getDataPublicacao() {
    return dataPublicacao;
}
```

```
public void setDataPublicacao(Calendar dataPublicacao) {  
    this.dataPublicacao = dataPublicacao;  
}
```

Uma vez criado o atributo, precisamos de um componente no nosso formulário para que o usuário possa preencher a data, abra o arquivo `form.xhtml`.

Vamos colocar o nosso componente no final do `form`, logo antes do botão "Cadastrar". Usaremos o componente `<h:inputText />`, e também o `<h:message for="" />` como fizemos nos campos anteriores.

```
<div>  
    <h:outputLabel value="Data de Publicação" />  
    <h:inputText value="#{adminLivrosBean.livro.dataPublicacao}"  
        id="dataPublicacao" />  
    <h:message for="dataPublicacao" />  
</div>
```

Para que nenhuma mensagem seja suprimida, adicionamos o `id` seguido pelo `for="dataPublicacao"`.

Realize um *Full Publish* no projeto e tente cadastrar um novo Livro com a data de publicação.

The screenshot shows a web browser window with the URL `localhost:8080/casadocodigo/livros/form.xhtml;jsessionid=B3kwYZ5ztHuh-y9btX1QLpE1AOXKg_7n_lV9W45O.aluras-r`. The page displays a form for adding a new book. The fields are as follows:

- Título: Especial de Vendas
- Descrição: Especial de Vendas
- Número de Páginas: 210
- Preço: 49
- Autores:
 - Paulo Silveira
 - Sérgio Lopes
 - Guilherme Silveira
- Data de Publicação: 10/05/2016

The 'Data de Publicação' field contains the value '10/05/2016'. The 'Cadastrar' button at the bottom left of the form is highlighted with a cursor.

Quando tentamos inserir a data `10/05/2016` recebemos um erro: *'Erro de conversão ao definir o valor '10/05/2016' para 'null Converter'*. Quando trabalhamos com data, ela não tem um conversor automático, não tem como o JSF saber se queremos uma data no formato brasileiro, internacional ou outro qualquer. Temos então que informar ao

JSF qual conversor usaremos.

Vamos colocar uma nova tag `<f:convertDateTime />` dentro do `<h:inputText />` e informar qual *pattern* queremos usar no converter.

```
<div>
    <h:outputLabel value="Data de Publicação" />
    <h:inputText value="#{adminLivrosBean.livro.dataPublicacao.time}"
        id="dataPublicacao" />
    <f:convertDateTime pattern="dd/MM/yyyy" />
</h:inputText>
    <h:message for="dataPublicacao" />
</div>
```

Se você tentou subir a aplicação agora com o *Converter*, vai perceber que a tag `<f:convertDateTime />` só funciona para objetos do tipo `java.util.Date` e estamos utilizando `java.util.Calendar`. Para obter o `Date` a partir de um `Calendar`, basta chamar o método `getTime()` do `Calendar`, em nossa *Expression Language* dentro do `value` usamos `.time`.

Faça *Full Publish* novamente e tente inserir um livro.

Você deve ter recebido um erro do tipo `PropertyNotFoundException` isso aconteceu porque o nosso atributo `dataPublicacao` do objeto `livro` está `null`, ele ainda não foi inicializado. Vamos resolver isso na nossa classe `Livro`, inicializando o atributo `dataPublicacao` com a data atual.

```
@Temporal(TemporalType.DATE)
private Calendar dataPublicacao = Calendar.getInstance(); // Pegamos uma instância de Calendar
```

Agora, ao abrir o formulário já será carregada com a data de hoje.

The screenshot shows a web browser window with the URL `localhost:8080/casadocodigo/livros/form.xhtml`. The page displays a form for adding a book. The fields are labeled 'Título', 'Descrição', 'Número de Páginas', 'Preço', 'Autores' (with a dropdown menu showing 'Paulo Silveira', 'Sérgio Lopes', 'Guilherme Silveira', and 'Alberto Souza'), 'Data de Publicação' (with the value '19/04/2016'), and a 'Cadastrar' button. A red arrow points to the date field.

Título

Descrição

Número de Páginas

Preço

Autores

Paulo Silveira
Sérgio Lopes
Guilherme Silveira
Alberto Souza

Data de Publicação 19/04/2016

Cadastrar

As informações serão salvas, mas ainda sem a data de publicação.

The screenshot shows a web browser window with the URL `localhost:8080/casadocodigo/livros/lista.xhtml`. The page displays a success message: "• Livro cadastrado com sucesso!" followed by a table of books.

Título	Descrição	Páginas	Preço	Autores
Java 8 Prático	Java 8 Prático	120	59.00	Paulo Silveira, Guilherme Silveira,
Java 8 Prático 3	Java 8 Prático 3	120	59.00	Paulo Silveira, Sérgio Lopes, Guilherme Silveira,
Android Básico	Android Básico	210	59.00	Paulo Silveira,
Android Básico 2	Android Básico 2	829	129.00	Paulo Silveira, Sérgio Lopes, Guilherme Silveira, Alberto Souza,
JavaEE	JavaEEJavaEEJavaEEJavaEE	89	59.00	Paulo Silveira, Sérgio Lopes,
Java 8 Prático	Java 8 Prático	120	59.00	Sérgio Lopes, Guilherme Silveira, Alberto Souza,
TDD no Mundo Real	TDD no Mundo Real	320	59.00	Sérgio Lopes,
SOLID	Solid do Aniche	240	78.00	Sérgio Lopes, Guilherme Silveira,
SOLID 2	Solid 2	400	210.00	Sérgio Lopes, Guilherme Silveira,
Certificação Java 8	Certificação Java 8	250	59.00	Sérgio Lopes, Guilherme Silveira, Alberto Souza,
Especial de Vendas	Especial de Vendas	210	49.00	Sérgio Lopes, Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
TDD no Mundo Real 2	TDD no Mundo Real 22	120	59.00	Guilherme Silveira,
Outro Livro	livro qualquer	210	520.00	Guilherme Silveira, Alberto Souza,
TDD no Mundo Real	TDD no Mundo Real	210	59.00	Alberto Souza,
TDD no Mundo Real	TDD no Mundo Real	210	59.00	Alberto Souza,
JavaEE	Conheça as novidades do JavaEE 7	89	59.00	Alberto Souza,

Nós conseguimos cadastrar o nosso livro, mas tivemos que usar um valor *default* (a data atual). Talvez, quando você cadastre o seu livro, esta seja realmente a data. Será que essa é realmente a melhor opção para nosso negócio? Vamos evoluir o nosso sistema mais adiante.

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 4 - Atividade 2 Guardando a data de publicação do Livro | Alura

Dica: Caso precise do projeto que foi feito na última aula, para seguir com seus estudos daqui, basta baixar o código [aqui](#)

Uma vez que precisamos salvar a data de publicação do livro, precisaremos de um novo campo na classe `Livro`. Abra sua classe `Livro` e adicione o atributo `dataPublicacao` do tipo `Calendar`. Lembre-se de criar o `get` e `set` do novo atributo.

Para que a JPA possa entender o novo atributo, precisamos o marcar com a annotation `@Temporal`. Mas ainda assim temos que decidir o que vamos guardar em relação a data. Podemos guardar *Data*, *Hora* ou os dois *DataHora*, mas guardaremos apenas a data. Assim, informamos dentro da annotation o valor `TemporalType.DATE`.

Para evitar problemas com nosso conversor, precisamos instanciar a data já na criação do `Livro`, adicione um `Calendar.getInstance()` direto no atributo.

Salve seu código, mas ainda não conseguiremos testar. No próximo exercício será possível verificar se está tudo correto com nossa *Data de Publicação* do `Livro`.

Opinião do instrutor

Na sua classe `Livro` você deve ter adicionado o atributo abaixo com o seguinte mapeamento:

```
@Temporal(TemporalType.DATE)
private Calendar dataPublicacao = Calendar.getInstance();

// Crie o getter e o setter no final do arquivo.
```

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 4

- Atividade 3 Cadastrando a data de publicação | Alura

Nossa classe Livro já possui o atributo `dataPublicacao`, agora precisamos permitir que o usuário possa informar a data de publicação na página `form.xhtml`.

Ao final do nosso formulário, bem antes do `<h:commandButton />`, podemos adicionar mais um campo para a *Data de Publicação*. Coloque a `<div>` como já estamos fazendo e também uma `<h:outputLabel>`. Também precisaremos de um `<h:inputText>`.

A grande questão aqui é que o usuário pode digitar uma data de qualquer forma. Mas como estamos no Brasil, queremos que ele informe no formato brasileiro, que seria o `dia/mês/ano`. Assim, vamos adicionar um conversor de data para esse formato, dentro do `<h:inputText>`. O conversor é o `<f:convertDateTime pattern="dd/MM/yyyy" />` já com o padrão brasileiro que precisamos.

Esse `input` deverá apontar para o atributo novo que criamos. Por isso o `value` dele irá para `#{{adminLivrosBean.livro.dataPublicacao.time}}`. Lembrando que o `time` no final informa que usaremos um `java.util.Date`, pois nosso conversor `f:convertDateTime` só trabalha com `Date`.

Salve seu código e teste a inserção de um novo Livro com a Data de Publicação.

Opinião do instrutor

Todo nosso `input` para a *Data de Publicação* deverá ficar da seguinte forma:

```
<div>
    <h:outputLabel value="Data de Publicação" />
    <h:inputText value="#{{adminLivrosBean.livro.dataPublicacao.time}}"
        id="dataPublicacao">
        <f:convertDateTime pattern="dd/MM/yyyy" />
    </h:inputText>
    <h:message for="dataPublicacao" />
</div>
```

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 4

- Atividade 4 Criando nosso próprio Converter | Alura

Video Player is loading.

Current Time 0:00

Duration 14:33

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Criando nosso próprio Converter

Vamos fazer um teste, modificaremos a hora do *Sistema Operacional* para **23:30h** e pediremos para que a hora não seja automaticamente. Feito isto, entraremos no formulário novamente. A data exibida deve estar mostrando do

próximo dia. Isso ocorre porque existe diferença de *time zone* (fuso horário). Quando não especificamos nenhum, o default é UTC-0, e podemos gerar problemas futuros.

Vamos corrigir nossa data, informando o *timeZone* do nosso converter para "America/Sao_Paulo". Você pode usar qualquer outro, dependendo da sua necessidade. Nossa converter ficará assim:

```
<!-- Mantenha os demais componentes intactos -->
<f:convertDateTime pattern="dd/MM/yyyy" timeZone="America/Sao_Paulo"/>
```

Faça *Full Publish* novamente e perceba que ao entrar no formulário, a data está correta, devendo exibir o dia de hoje no campo.

Mas algo ainda não está legal. Imagine se em todos os campos de data tivermos que fazer essas configurações, dizer qual formato usar, *timeZone*, etc. Perderemos muito tempo apenas configurando a mesma coisa em vários lugares.

Para casos como esse, podemos criar nosso próprio *Converter*. Onde já informaremos tudo o que desejamos que seja configurado para um determinado tipo de dado. Mas antes, vamos voltar o *input* de *dataPublicacao* para usar *Calendar* e remover o `<f:convertDateTime />` que usamos antes.

```
<!-- Só o input simples -->
<h:inputText value="#{adminLivrosBean.livro.dataPublicacao}"
    id="dataPublicacao" />
```

Vamos criar uma nova classe chamada *CalendarConverter* no pacote `br.com.casadocodigo.loja.converters`. Nessa nova classe, precisamos informar ao JSF que ela será nosso conversor, utilizamos a anotação `@FacesConverter` para mapear a nossa classe, tornando-a de fato uma classe de conversão. Dentro da annotation, informamos qual o tipo de dado que ele converterá: `@FacesConverter(forClass=Calendar.class)`.

Além da annotation, precisamos implementar uma interface que possui os métodos base de conversão. Esta interface é a `javax.faces.convert.Converter`, que nos obriga implementar dois métodos, o `getAsObject` e o `getAsString` como podemos ver suas assinaturas abaixo:

```
@FacesConverter(forClass=Calendar.class)
public class CalendarConverter implements Converter {

    @Override
    public Object getAsObject(FacesContext context,
        UIComponent component, String dataTexto) {
        return null;
    }

    @Override
    public String getAsString(FacesContext context,
```

```
        UIComponent component, Object dataObject) {  
    return null;  
}  
}
```

Esses dois métodos funcionam assim. Quando a informação está na tela, ela é uma String, nesse ponto o **JSF** chama o método `getAsString()` do nosso Converter. Quando está no ManagedBean é um Objeto que queremos, desta forma o **JSF** chama o método `getAsObject()`.

Agora precisamos implementar de fato nosso converter, faremos uso do mesmo conversor que usamos na tela, só que agora reutilizando o que ele já faz configurado em um único local. Usaremos o `DateTimeConverter`, e deixaremos o trabalho de conversão pesado com ele. Vamos declará-lo como atributo de nossa classe:

```
@FacesConverter(forClass=Calendar.class)  
public class CalendarConverter implements Converter {  
  
    private DateTimeConverter converter = new DateTimeConverter();  
  
    //Mais código abaixo...  
}
```

Na assinatura dos métodos, perceba que a única diferença é o retorno de cada um e o último parâmetro. No `getAsObject()`, o último parâmetro é a data em texto que queremos transformar para `Object`, e no `getAsString()`, o último parâmetro é a data `Object` que queremos transformar em texto (`String`).

Vamos começar pelo método `getAsObject()`. Queremos recuperar a data como um objeto, então chamamos o `converter.getAsObject`, passando os mesmos parâmetros que recebemos:

```
Date data = (Date) converter.getAsObject(context, component, dataTexto);
```

Como nosso converter é para `Date`, vamos realizar a transformação desse `Date` em `Calendar`.

```
Calendar calendar = Calendar.getInstance();  
calendar.setTime(data);  
return calendar;
```

Quando estavamos trabalhando com a data no formulário, nós tínhamos o problema do `timeZone` e o problema do `patter` de formatação da data em texto, precisamos resolver estes problemas aqui também. Desta forma, precisamos que o converter saiba qual `timeZone` usar e qual `pattern` usar quando a data estiver no formato texto.

Tanto o método `getAsObject` como o `getAsString`, farão uso do mesmo `Converter`, o que significa que teríamos que setar os parâmetros de `timeZone` e `pattern` duas vezes. Mas podemos ser mais espertos e criar um construtor para nosso converter, onde já realizamos toda configuração necessária:

```
public CalendarConverter() {
    converter.setPattern("dd/MM/yyyy");
    converter.setTimeZone(TimeZone.getTimeZone("America/Sao_Paulo"));
}
```

Pronto, todas as configurações foram realizada em um único lugar, e poderemos usar nos dois métodos.

Ainda temos que implementar o método `getAsString` que faz justamente o inverso. Aqui queremos recuperar a data no formato texto, então chamamos o `converter.getAsString`, passando os mesmos parâmetros que recebemos no método, com um detalhe, o object é um `Calendar`, e o `DateTimeConverter` recebe um `Date`. Vamos transformar os dados, e como não tem muito segredo, segue o código abaixo:

```
public String getAsString(FacesContext context,
                           UIComponent component, Object dataObject) {
    if (dataObject == null)
        return null;

    Calendar calendar = (Calendar) dataObject;
    return converter.getAsString(context, component, calendar.getTime());
}
```

O único ponto de destaque é a verificação que fazemos antes de tudo. Pois se passarmos um `null` para o converter, ele lançará uma *Exception* e não é o que desejamos.

Nosso converter está pronto. Para que você confira todo o código que temos, segue:

```
@FacesConverter(forClass=Calendar.class)
public class CalendarConverter implements Converter {

    private DateTimeConverter converter = new DateTimeConverter();

    public CalendarConverter() {
        converter.setPattern("dd/MM/yyyy");
        converter.setTimeZone(TimeZone.getTimeZone("America/Sao_Paulo"));
    }

    @Override
    public Object getAsObject(FacesContext context,
                             UIComponent component, String dataTexto) {
        Date data = (Date) converter.getAsObject(context, component, dataTexto);
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(data);
        return calendar;
    }

    @Override
```

```
public String getAsString(FacesContext context,
    UIComponent component, Object dataObject) {
    if (dataObject == null)
        return null;

    Calendar calendar = (Calendar) dataObject;
    return converter.getAsString(context, component, calendar.getTime());
}
```

Pronto, agora sempre que fizermos uso do Calendar teremos nosso converter em ação.

Podemos remover o inicializador de nosso atributo `dataPublicacao` da classe `Livro`, não precisamos inicializar mais nosso atributo. Assim o campo vem vazio na tela, pronto para o usuário preencher, o que seria o esperado. Veja o trecho de código abaixo:

```
public class Livro {

    // Demais atributos acima

    @Temporal(TemporalType.DATE)
    private Calendar dataPublicacao;

    // Demais atributos e métodos abaixo

}
```

O que mais ganhamos com essa implementação do Converter? Ganhamos o uso de Calendar em qualquer entidade do sistema, sendo transformada para texto automaticamente pelo **JSF**, sem que tenhamos que nos preocupar com o formato e `timezone`. Criamos um único objeto que já serve para todo o sistema.

Faça novamente um *Full Publish* teste o cadastro de Livro completo.

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 4

- Atividade 5 Criando um conversor para nossa data | Alura

O converter que estamos usando tem alguns problemas. O primeiro deles é que ele está gerando uma data com valor errado se não usarmos o `TimeZone` correto, e o segundo é o tipo com que ele trabalha. Como vimos no exercício anterior ele trabalha com `java.util.Date` mas nós estamos usando um `Calendar` e queremos trabalhar em cima de `Calendar`.

Para resolver esse dois problemas, podemos criar o nosso próprio converter. Crie a classe `CalendarConverter` no pacote `br.com.casadocodigo.loja.converters`.

Dentro de nossa classe temos que implementar a interface `javax.faces.convert.Converter`, a qual nos obrigará a implementar os métodos `getAsObject(FacesContext context, UIComponent component, String dataTexto)` e o `getAsString(FacesContext context, UIComponent component, Object dataObject)`.

Vamos fazer uso do conversor de `Date` que temos pronto, para nos ajudar. Crie um atributo de instância chamado `converter` do tipo `DateTimeConverter`, e já instancie a classe ai mesmo.

No construtor de nosso `CalendarConverter`, vamos informar o padrão do nosso sistema para o formato da data, e também já deixar o `TimeZone` setado para `TimeZone.getTimeZone("America/Sao_Paulo")`.

Nos métodos, o código será bem simples, no método `getAsObject` basta chamar o próprio converter que temos para `Date` que recebemos um `Date`, o que fica fácil transformar em `Calendar`.

```
Date data = (Date) converter.getAsObject(context, component, dataTexto);
```

E no método `getAsString`, devemos usar o `Calendar` que recebemos como objeto e passar como `Date` para nosso converter. Nesse método, precisamos evitar o `NullPointerException`, então verifique antes de realizar o *casting* se o objeto não é nulo, caso seja, retorne `null`. Após obter o `Calendar`, só precisamos chamar nosso converter para retornar o valor.

```
return converter.getAsString(context, component, calendar.getTime());
```

Agora não precisamos mais nos preocupar com instanciar o atributo `dataPublicacao` na classe `Livro`, assim, remova o `Calendar.getInstance()` que fizemos.

E para finalizar, abra o formulário `livros/form.xhtml` e remova o `<f:convertDateTime ... />` bem como retire o `.time` do `value` do `input`. Ficando simplesmente assim:

```
<h:inputText value="#{adminLivrosBean.livro.dataPublicacao}" id="dataPublicacao" />
<h:message for="dataPublicacao" />
```

Agora temos um converter que já tem o padrão pré-definido e bem como o `TimeZone` e o **JSF** identifica automaticamente o tipo que precisa para realizar a conversão sem que seja necessário informar na tag `input` um converter específico.

Realize um *Full Publish* e teste nossa aplicação.

Opinião do instrutor

Nosso converter, já com `TimeZone` e também com o `pattern` informado, deverá ficar assim:

```
@FacesConverter(forClass=Calendar.class)
public class CalendarConverter implements Converter {

    private DateTimeConverter converter = new DateTimeConverter();

    public CalendarConverter() {
        converter.setPattern("dd/MM/yyyy");
        converter.setTimeZone(TimeZone.getTimeZone("America/Sao_Paulo"));
    }

    @Override
    public Object getAsObject(FacesContext context,
        UIComponent component, String dataTexto) {
        Date data = (Date) converter.getAsObject(context, component, dataTexto);
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(data);
        return calendar;
    }

    @Override
    public String getAsString(FacesContext context,
        UIComponent component, Object dataObject) {
        if (dataObject == null)
            return null;

        Calendar calendar = (Calendar) dataObject;
        return converter.getAsString(
            context, component, calendar.getTime());
    }
}
```



06

Convertendo datas



60%

Porque tivemos que criar um conversor para Calendar se já existe um para Date?

ATIVIDADES
6 DE 9

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

**A**

Porque o JSF foi feito numa época em que apenas o Date existia, então ele não conhece o Calendar.

Não é bem por isso. JSF foi feito quando já existia Calendar, mas preferiram fazer conversor automático apenas para Date, limitando assim a plataforma. Temos que fazer a conversão de Calendar na mão.

B

Porque o conversor de Date não funcionou corretamente.

O conversor de Date funciona perfeitamente, por isso reutilizamos parte dele no nosso próprio converter de Calendar.



88.4k xp

**a**

Porque queremos trabalhar direto com Calendar ao invés de fazer

parse para Date toda hora.

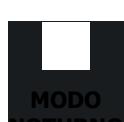


60%

ATIVIDADES
6 DE 9

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD



MODO
NOTURNO



[PRÓXIMA ATIVIDADE](#)



88.4k xp



Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 4

- Atividade 7 Convertendo o Autor | Alura

Video Player is loading.

Current Time 0:00

Duration 0:00

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Convertendo o objeto Autor

Uma vez que aprendemos sobre `Converters` criando a classe `CalendarConverter`, podemos evoluir essa ideia para melhorar o nosso código. Observe que temos um caso parecido, a nossa lista de autores do nosso formulário. Além de

já usarmos um `Converter` que é do próprio **JSF**, a `javax.faces.Integer`, estamos trabalhando sempre com o ID's do `Autor` em vez de utilizar o próprio objeto `Autor`.

Vimos como fazer isto anteriormente. Criaremos a classe `AutorConverter` no pacote `br.com.casadocodigo.loja.converters`, já vamos anotá-la com `@FacesConverter("autorConverter")`. Definimos o nome "autorConverter" para poderemos referenciá-lo em outros locais caso seja necessário. Implementaremos também a interface `Converter`.

Usando o atalho do Eclipse, pressione `ctrl + 1` em cima do nome da classe, e escolha a opção `add unimplemented methods.`

```
@FacesConverter("autorConverter")
public class AutorConverter implements Converter {

    @Override
    public Object getAsObject(FacesContext context,
        UIComponent component, String id) {
        return null;
    }

    @Override
    public String getAsString(FacesContext context,
        UIComponent component, Object autorObject) {
        return null;
    }
}
```

Começaremos pelo método `getAsObject()`... Queremos recuperar o autor como um objeto, então faremos `new Autor` e da instância obtida, chamaremos o `setId` que recebe o `id` - basta transformar de `String` para `Integer` com `Integer.valueOf`. Já temos a ideia principal, só iremos verificar antes, se a `String` recebida não é nula ou vazia. O método ficará assim:

```
public Object getAsObject(FacesContext context, UIComponent component, String id) {
    if (id == null || id.trim().isEmpty())
        return null;

    Autor autor = new Autor();
    autor.setId(Integer.valueOf(id));

    return autor;
}
```

Seguiremos para o método `getAsString()` que fará justamente o inverso. Queremos recuperar o id do autor no formato `String`. Já recebemos o objeto `Autor` como `Object`, agora, faremos um *casting* de volta para `Autor`. Deste, chamaremos o `getId` e depois o `toString()` - para transformá-lo em texto. Se o objeto recebido estiver `null`, teremos uma `NullPointerException`, então, evitaremos esse caso desde o início do método.

Juntando tudo, nosso código do `getAsString` ficará assim:

```
public String getAsString(FacesContext context, UIComponent component, Object autorObject) {  
    if (autorObject == null)  
        return null;  
  
    Autor autor = (Autor) autorObject;  
    return autor.getId().toString();  
}
```

Agora que conseguimos converter de `Autor` com o nome `autorConverter`, podemos voltar ao arquivo `form.xhtml` e trocar o `javax.faces.Integer` simplesmente para `autorConverter`. Além disso, o próprio `value` do componente `select`, precisa de uma lista de ID's, porém, a entidade `Livro` já possui uma lista de autores relacionada. Podemos fazer uso dessa lista diretamente, mudando simplesmente para:

```
value="#{adminLivrosBean.livro.autores}"
```

Dentro da tag, também precisamos realizar um mudança em `<f:selectItems />`, tirando o `id` no `itemValue`, e usando diretamente o `autor`. Com as alterações, o código ficou assim:

```
<div>  
    <h:outputLabel value="Autores" />  
    <h:selectManyListbox value="#{adminLivrosBean.livro.autores}"  
        converter="autorConverter" id="autores">  
        <f:selectItems value="#{adminLivrosBean.autores}" var="autor"  
            itemValue="#{autor}" itemLabel="#{autor.nome}" />  
    </h:selectManyListbox>  
    <h:message for="autores" />  
</div>
```

Além disso, o uso do converter também facilitará nossa vida no `AdminLivrosBean`. Abra a classe e dentro do método `salvar()`, encontraremos um `for` que percorre a lista de `ids` dos autores. Como não estamos mais usando a lista na tela, removeremos seguinte trecho:

```
for (Integer autorId : autoresId) {  
    livro.getAutores().add(new Autor(autorId));  
}
```

E também o atributo `List<Integer> autoresId`, junto com seu *getter* e *setter*: `getAutoresId` e `setAutoresId`.

```
private List<Integer> autoresId = new ArrayList<>();

// deixar os demais métodos e remover os get e set abaixo
public List<Integer> getAutoresId() {
    return autoresId;
}

public void setAutoresId(List<Integer> autoresId) {
    this.autoresId = autoresId;
}
```

Faça novamente um *Full Publish* e tente realizar o cadastro de um Livro para ver o que acontece.

Você deve ter recebido um erro de validação. Trata-se de um erro bem comum no JSF quando estamos usando **Converters**. Isso acontece porque o JSF espera que os valores da lista sejam comparados com os valores passados para o nosso conversor. Assim, o **JSF** precisa comparar o autor convertido com o autor da lista. Uma vez que ele não consegue, ele dará um erro de validação.

Mas por que estamos recebendo esse erro, se sabemos que o autor convertido existe na lista?

Lembre-se que quando não especificamos uma forma de comparação de objetos no *Java*, ele irá comparar os objetos usando `o ==` - mas este irá comparar a referência de memória, o que não servirá neste caso. Então, precisaremos mudar a forma de comparação, e criar nosso método `equals()` na classe `Autor`, para que o **JSF** use o novo método corretamente.

Como o Eclipse já realiza esta ação, peça para o Eclipse gerar o `equals()`. Entre na classe `Autor` e pressione `Ctrl + 3`, em seguida, será aberta uma caixa. Digite `equals` e selecione a opção "Generate hashCode() and equals()", depois, selecione apenas a opção do `id` e desmarque o "nome" caso esteja selecionado. Clique em `OK` e veja o código gerado.

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((id == null) ? 0 : id.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
```

```
Autor other = (Autor) obj;
if (id == null) {
    if (other.id != null)
        return false;
} else if (!id.equals(other.id))
    return false;
return true;
}
```

Vamos pedir para o Eclipse gerar o `toString()`. Com isto, todo o cadastro de Livros já deve estar funcionando e você também deve ser capaz de relacionar os Autores com o Livro, o que é feito internamente usando os objetos em vez de primitivos.

Desta forma, nosso código fica muito mais Orientado a Objetos de fato, e mais elegante.

Realize um *Full Publish* e teste seu cadastro. Depois disso, vá para os exercícios e tire suas dúvidas no fórum.

Bons estudos!

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 4

- Atividade 8 Criando o conversor para Autor | Alura

Considerando o que já fizemos para a *Data de Publicação*, podemos melhorar também a seleção de autores, que hoje é feita salvando os ID's dos autores no `AdminLivrosBean`. Em um sistema O.O. queremos trabalhar com o objeto `Autor`. Vamos criar também um conversor para o `Autor`.

O nome da classe é `AutorConverter` que também deve implementar a interface `Converter`. Para ser encontrada pelo **JSF** usamos a annotation `@FacesConverter("autorConverter")`. Perceba que dessa vez estamos nomeando o conversor ao invés de indicar para qual classe ele irá funcionar, como fizemos no `CalendarConverter`.

Faça o método `getAsObject` retornar um objeto `Autor`, lembrando que a `String` que recebemos como parâmetro terá apenas o `ID` do autor.

Da mesma forma faça o método `AsString` retornar o `ID` do `Autor` como texto.

Com isso teremos o nosso converter pronto, mas diferente do `Calendar`, precisamos dizer onde vamos utilizar o conversor. Abra novamente o formulário `livros/form.xhtml` e procure pelo `<h:selectManyListbox ...>`. Nele, adicione o atributo `converter="autorConverter"` e modifique o valor para `value="#{adminLivrosBean.livro.autores}"`. E dentro dele no `<f:selectItems ...>`, podemos alterar o atributo `itemValue` para ficar `itemValue="#{autor}"`, pois o conversor já irá tratar o que precisa ser tratado do `Autor`.

Lembre-se que para o funcionamento correto do `Converter` precisamos ter os métodos `equals` e `hashCode` sobrescritos corretamente na classe `Autor`.

Outra classe que sofrerá mudanças será a `AdminLivrosBean`. Nela, vamos remover o atributo `autoresId`, pois não precisamos mais dele, e dentro do método `salvar`, remova o `foreach` que cria os autores, também não precisamos mais dele. Ficando o método `salvar` assim:

```
@Transactional  
public String salvar() {  
    dao.salvar(livro);  
  
    context.getExternalContext()  
        .getFlash().setKeepMessages(true);  
    context  
        .addMessage(null, new FacesMessage("Livro cadastrado com sucesso!"));  
}
```

```
        return "/livros/lista?faces-redirect=true";
    }
```

Veja que ficou muito mais simples e agora podemos usar o `Autor` como sendo objeto em nossa `view` e também nos `noossos Beans`.

Faça um novo *Full Publish* e veja se tudo continua funcionando normalmente.

Opinião do instrutor

Após todas as alterações terem sido feitas, seu código do `Converter` deve se parecer com o seguinte:

```
@FacesConverter("autorConverter")
public class AutorConverter implements Converter {

    @Override
    public Object getAsObject(FacesContext context,
        UIComponent component, String id) {
        if (id == null || id.trim().isEmpty()) return null;
        System.out.println("Convertendo para Objeto: " + id);

        Autor autor = new Autor();
        autor.setId(Integer.valueOf(id));

        return autor;
    }

    @Override
    public String getAsString(FacesContext context,
        UIComponent component, Object autorObject) {
        if (autorObject == null) return null;
        System.out.println("Convertendo para String: " + autorObject);

        Autor autor = (Autor) autorObject;
        return autor.getId().toString();
    }
}
```

Também fique atento para alterar o formulário `form.xhtml`, deixando o `selectManyListbox` parecido com o código a seguir:

```
<div>
    <h:outputLabel value="Autores" />
    <h:selectManyListbox      value="#{adminLivrosBean.livro.autores}"
```

```
        converter="autorConverter" id="autores">
    <f:selectItems value="#{adminLivrosBean.autores}"
        var="autor"
        itemValue="#{autor}" itemLabel="#{autor.nome}" />
</h:selectManyListbox>
<h:message for="autores" />
</div>
```

E por fim, não deixe de criar os métodos `equals` e `hashCode` da classe `Autor`.

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((id == null) ? 0 : id.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Autor other = (Autor) obj;
    if (id == null) {
        if (other.id != null)
            return false;
    } else if (!id.equals(other.id))
        return false;
    return true;
}
```

09

Analisando o Converter de Autor



63%

O que ganhamos criando o converter de Autor? (Selecione todas as opções aplicáveis)

ATIVIDADES
9 DE 9

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD


MODO
NOTURNO


ABRIR
CADERNO

L

88.6k xp



Ganhamos flexibilidade na tela, fazendo com que componentes conheçam direto o objeto Autor.

Correto, o sistema fica mais flexível com os componentes do JSF usando objetos ao invés de inteiros.



Ganhamos um converter próprio, que já preenchemos com uma nova instância de um Autor, colocando já o ID selecionado dentro do Autor.

Os converters do JSF apenas convertem primitivos e objetos básicos do Java, com nosso converter, temos o JSF conversando com um objeto do nosso sistema que já recebemos no ManagedBean populado com o ID.

a



Ganhamos mais semântica, uma vez que nosso sistema trabalha diretamente com objetos Autor e não mais com inteiros.



63%

ATIVIDADES
9 DE 9

Correto, é muito mais fácil ler que queremos guardar os valores em autores do que em autoresId, pois pegamos um autor da lista, e o guardamos em uma lista de objetos autores.

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD



Ganhamos flexibilidade na tela, fazendo com que componentes conheçam direto o objeto Autor, também mais semântica, uma vez que nosso sistema trabalha diretamente com objetos Autor e não mais com inteiros e temos um converter próprio, que já preenche a lista com instâncias de objetos Autor, colocando o ID do Autor selecionado no objeto. Assim, temos uma aplicação mais esperta e não temos que tratar listas de Integers .

[PRÓXIMA ATIVIDADE](#)



88.6k xp



Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 5

- Atividade 1 Salvando a Imagem no Servidor | Alura

Video Player is loading.

Current Time 0:00

Duration 11:06

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Salvando a Imagem no Servidor

No site oficial da [Casa do Código](#), podemos ver que todos os livros possuem de fato uma capa. Este será o nosso próximo desafio: fazer o livro ter de fato uma capa.

Para isto, adicionaremos um campo novo no formulário, que nos permitirá selecionar o local do arquivo enviado para o servidor. O JSF 2.2 nos trouxe um componente próprio para isso com o JavaEE 7, esse componente é o `<h:inputFile>`.

No fim do formulário, vamos adicionar o campo **Capa do Livro** apontando para `AdminLivrosBean`. Usaremos um atributo a parte para fazer a ligação com o arquivo que queremos subir. Adicione também um `<h:message>` para possíveis problemas.

```
<div>
    <h:outputLabel value="Capa do Livro" />
    <h:inputFile value="#{adminLivrosBean.capaLivro}" id="capaLivro" />
    <h:message for="capaLivro" />
</div>
```

Um detalhe importantíssimo é que precisamos dizer no nosso formulário que queremos enviar arquivo por ele e não em formato de texto. Para isso, no formulário, adicionaremos o atributo `enctype` dentro do `form`. O código ficará assim:

```
<h:form enctype="multipart/form-data">
```

Iremos manter os demais campos abaixo. Com o nosso formulário pronto, poderemos focar na classe `AdminLivrosBean`. Nela, precisaremos do atributo que declaramos no `xhtml`: o `capaLivro`. O tipo desse atributo é o ponto mais relevante.

Desejamos transferir arquivos, mas, anteriormente tínhamos que trabalhar com base em `arrays` de bytes ou `FileInputStream` - tudo feito manualmente. O JavaEE 7 nos trouxe um novo objeto que já tem a capacidade de salvar um arquivo dentro dele: o `Part`. Usaremos o tipo `Part` e, no método `salvar()`, realizaremos a transferência do arquivo recebido pelo formulário para o sistema operacional.

O tipo `Part` possui um método chamado `write()` que recebe uma `String` com o caminho onde queremos salvar o arquivo dentro do `s.o..` Por isso, vamos passar o seguinte caminho: `/casadocodigo/livros/`, concatenando a isso o nome original do arquivo que nos foi enviado e pode ser obtido pelo código `arquivo.getSubmittedFileName()`.

Assim, nossa classe `AdminLivrosBean` ficará assim:

```
public class AdminLivrosBean {
    // Os demais atributos ficam aqui, não alterar!
    private Part capaLivro;
    @Transactional
    public String salvar() throws IOException {
```

```
        dao.salvar(livro);
        capaLivro.write("/casadocodigo/livros" + capaLivro.getSubmittedFileName());
    }
    // Mantenha o restante do método aqui
}
// Demais getter's e setter's, não alterar
}
```

Importantíssimo já criar a pasta `/casadocodigo/livros` no seu sistema operacional. Assim, já poderemos fazer testes na aplicação. Após salvar tudo, execute o *Full Publish* para ver se o arquivo é realmente enviado para o caminho acima do sistema operacional.



02 Criando o campo para seleção de arquivo

[PRÓXIMA ATIVIDADE](#)

67%

ATIVIDADES
2 de 14FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

Dica: Caso precise do projeto que foi feito na ultima aula, para seguir com seus estudos daqui, basta baixar o código [aqui](https://github.com/alura-cursos/java-ee-webapp/archive/ef6dcc48786b230dc0fa67cc85ae8fe0329b6b44.zip) (<https://github.com/alura-cursos/java-ee-webapp/archive/ef6dcc48786b230dc0fa67cc85ae8fe0329b6b44.zip>)

Vamos possibilitar o envio da capa do livro para ser salva em no servidor onde o sistema está rodando. Para isso, faremos uso da nova tag `<h:inputFile ... >` que veio no JSF 2.2.

Adicione a tag no nosso formulário `form.xhtml` bem ao final, antes do `<h:commandButton ... >`. Lembre-se de adiciona na estrutura padrão que já estamos utilizando (`div` e um `<h:outputLabel>`).



O valor do campo, aponte para `value="#{adminLivrosBean.capaLivro}"`. Ainda vamos criar esse atributo em nosso Bean .

Para que o envio de arquivos funcione, precisamos mudar o tipo do nosso formulário. Coloque o formulário para definir o atributo `enctype="multipart/form-data"` .

88.8k xp



Antes de testar, vamos fazer o próximo exercício.



67%

Opinião do instrutor

ATIVIDADES
2 de 14

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD



88.8k xp

O arquivo `form.xhtml` deve ter ficado parecido com o código abaixo:

```
<h:form enctype="multipart/form-data">  
    <!-- Os demais campos continuam aqui! -->  
  
    <div>  
        <h:outputLabel value="Capa do Livro" />  
        <h:inputFile value="#{adminLivrosBean.capa}" />  
        <h:message for="capaLivro" />  
    </div>  
    <h:commandButton value="Cadastrar" action="#{adminLivrosBean.salvar}" />  
</h:form>
```

[COPIAR CÓDIGO](#)



03 Enviando arquivos no JavaEE 7

[PRÓXIMA ATIVIDADE](#)

67%

Como ficou facilitado o upload de arquivos no JavaEE 7?

ATIVIDADES
3 de 14

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD

 MODO
NOTURNO

 ABRIR
CADERNO

**A**

Através da tag `<h:inputfile>` que envia uma array de bytes para o servidor como representação do arquivo.

Não é bem assim. O `<h:inputfile>` envia o arquivo usando o novo objeto `Part` do JavaEE 7. Sendo muito mais fácil tratar arquivos com `Part` do que com um array de bytes.

B

Através do `<input type="file">` do próprio HTML que envia a imagem para o servidor como uma List de bytes.

O JSF possui seu próprio componente de upload, apesar de ser possível usar o do HTML, o do framework evita erro na aplicação. Lembrando que ele não recebeu mais um array ou List de bytes, agora temos o objeto `Part`

para tratar arquivos.



67%

ATIVIDADES
3 de 14



Através da tag `<h:inputFile>` que envia o arquivo selecionado para um objeto do tipo `Part`, facilitando a manipulação de arquivos no Bean.

Exato! O JavaEE 7 trouxe o `<h:inputFile>` e o objeto `Part` como um facilitador de upload de arquivos.

FÓRUM DO
CURSO

VOLTAR
PARA
DASHBOARD



PRÓXIMA ATIVIDADE



04 Recebendo o arquivo no Bean e salvando no disco

[PRÓXIMA ATIVIDADE](#)

69%

ATIVIDADES
4 de 14FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARDABRIR
CADERNO

Para que o arquivo seja enviado para servidor, precisamos preparar o nosso `AdminLivrosBean` também:

- No `AdminLivrosBean` temos que adicionar o novo atributo responsável pelo arquivo, que será `private Part capaLivro` do pacote `javax.servlet.http.Part`;
- Antes de implementá-lo no método, temos que criar o caminho no nosso computador que irá receber o arquivo enviado. Lembre-se de criar em lugar que seja fácil de você localizar. Seguindo o padrão do vídeo será `/casadocodigo/livros`;
- Dentro do método `salvar` vamos inserir a seguinte linha:

```
capaLivro.write("/casadocodigo/livros/" + capaDoLivro.getSubmittedFileName())
```

Lembre-se que esse caminho pode variar dependendo do Sistema Operacional. lembre-se de usar o fórum em caso de problemas.

Opinião do instrutor

O nosso `AdminLivrosBean` : Implementando o atributo, lembre-se de usar o pacote `import javax.servlet.http.Part;`



89.0k xp

```
// Outros atributos
```

```
private Part capaLivro;
```

[COPIAR CÓDIGO](#)

Ainda no nosso bean, dentro do método salva:



69%

ATIVIDADES
4 de 14FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARDMODO
NOTURNOABRIR
CADERNO

89.0k xp



@Transactional

```
public String salvar() throws IOException {  
    String caminhoDoArquivo = "/casadocodigo/livros/";  
    capaLivro.write(caminhoDoArquivo);  
  
    //o resto do método
```

[COPIAR CÓDIGO](#)

Agora se você usa o windows, talvez precise passar a unidade de disco no caminho.

@Transactional

```
public String salvar() throws IOException {  
    String path = "c:/casadocodigo/livros/" + capaLivr  
    capaLivro.write(path);  
  
    //o resto do método
```

[COPIAR CÓDIGO](#)

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 5

- Atividade 5 Salvando o Path da Capa do Livro | Alura

Video Player is loading.

Current Time 0:00

Duration 15:08

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Salvando o Path da Capa do Livro

Podemos perceber no entanto que no banco de dados, o livro cadastrado não tem nenhuma referência com o caminho que informamos. Essas informações hoje estão desconectadas. Essa conexão pode ser criada através de um novo

atributo na classe `Livro`. Como só queremos salvar o caminho onde o arquivo se encontra no servidor, basta criar um atributo com as seguintes características:

```
public class Livro {  
    // Demais atributos acima, não alterar!  
  
    private String capaPath;  
  
    // Crie também o getter e setter para capaPath  
}
```

Agora que já podemos salvar o caminho do arquivo, vamos olhar novamente para o método `salvar` da classe `AdminLivrosBean`. Essa manipulação de arquivo que estamos fazendo pode começar a ficar muito comum no sistema, e ainda podemos considerar que outras partes do sistema também precisarão de upload de arquivos que talvez nem sejam capas de livro ou mesmo imagem.

Por isso, precisamos deixar o código que trata o envio do arquivo para o servidor reutilizável.

Vamos criar uma classe chamada `FileSaver` no pacote `br.com.casadocodigo.loja.infra` que será responsável por escrever esse arquivo no disco. Criaremos o método `write`, recebendo o tipo `Part` que será o arquivo que precisamos escrever no disco, e também uma `String` com o path referente ao negócio que está usando o `FileSaver`. Por exemplo, se estamos tratando de livro, podemos salvar na pasta padrão `/casadocodigo` mas dentro dela o path específico seria `/livros`.

Perceba que a pasta padrão no servidor pode mudar, então vamos deixar ela padronizada em um atributo `static` e `final` dentro do `FileSaver`.

Assim, resumindo nossa nova classe `FileSaver`, teremos o método `write` que receberá o arquivo e o caminho relativo para salvar, e retornará o caminho relativo usado para salvar já concatenado com o nome do arquivo. Ela terá também o caminho `default` do servidor. Ficando, ao final, conforme o código a seguir:

```
public class FileSaver {  
  
    private static final String SERVER_PATH = "/casadocodigo";  
  
    public String write(Part arquivo, String path) {  
        String relativePath = path + "/" + arquivo.getSubmittedFileName();  
        try {  
            arquivo.write(SERVER_PATH + "/" + relativePath);  
  
            return relativePath;  
        } catch (IOException e) {  
        }  
    }  
}
```

```
        throw new RuntimeException(e);
    }
}
```

Feito isso, podemos voltar ao `AdminLivrosBean` e alterar o método `salvar` para usar o `FileSaver`. Ficando ao final, o método `salvar` da seguinte forma:

```
@Transactional
public String salvar() throws IOException {
    dao.salvar(livro);
    FileSaver fileSaver = new FileSaver(); // Nossa nova classe
    livro.setCapaPath(fileSaver.write(capaLivro, "livros")); // Já chamamos o método write e já retornamos o pa

    context.getExternalContext()
        .getFlash().setKeepMessages(true);
    context
        .addMessage(null, new FacesMessage("Livro cadastrado com sucesso!"));

    return "/livros/lista?faces-redirect=true";
}
```

Novamente, execute um *Full Publish* e veja se tudo o que você fez está funcionando. Cadastre um novo livro com uma foto bacana para podermos exibir depois. Aqui estão algumas fotos de capas de livros para download.

<https://s3.amazonaws.com/caelum-online-public/java-ee-webapp/capas.zip>

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 5

- Atividade 7 Recebendo o arquivo no AdminLivrosBean | Alura

Estamos selecionando o arquivo no formulário, porém ainda não estamos pegando esse arquivo. Para capturar o arquivo, vamos abrir a classe `AdminLivrosBean` e adicionar nela um atributo chamado `capaLivro`. Esse atributo será do tipo `Part`, o novo tipo do **JavaEE 7** que representa um arquivo integrado ao `<h:inputFile>` do **JSF 2.2**.

Não queremos tratar esse arquivo no `Bean`, pois isso é código de *Infra*, então vamos criar uma nova classe chamada `FileSaver` no pacote `br.com.casadocodigo.loja.infra`. Nessa classe vamos criar um método `write` que recebe o arquivo do tipo `Part` e também um caminho relativo do tipo `String` onde desejamos salvar o arquivo. O método deve retornar uma `String` com o caminho relativo mais o nome do arquivo. Por enquanto não vamos nos preocupar com o que o método faz, vamos apenas criá-lo. *Faremos a implementação no próximo exercício.*

Dentro do método `salvar` do `AdminLivrosBean`, vamos adicionar algumas linhas que servirão para chamar o `FileSaver` e também para colocar o caminho do arquivo no `Livro`. Você pode colocar essa informação antes ou depois do código `dao.salvar(livro)`, pois o `salvar` do `Bean` só enviará o objeto para o banco de dados ao final da transação.

```
FileSaver fileSaver = new FileSaver();
livro.setCapaPath(fileSaver.write(capaLivro, "livros"));
```

Ainda não conseguimos testar, pois não colocamos código relevante no `FileSaver`. Vamos fazer isso no próximo exercício.



08 Criando o método write do FileSaver

[PRÓXIMA ATIVIDADE](#)

74%

Já estamos fazendo uso do `FileSaver`, agora podemos criar o método `write` que terá a responsabilidade de escrever o arquivo no disco.

ATIVIDADES
8 de 14

O método `write` tem a seguinte assinatura:

```
public String write(Part arquivo, String path) { ... }
```

[COPIAR CÓDIGO](#)

FÓRUM DO CURSO

VOLTAR PARA DASHBOARD



Para o caminho absoluto, crie uma "*constante*" chamada `SERVER_PATH` que terá o caminho do sistema de arquivos, onde você deseja salvar.



Dentro do método `write`, pegue o arquivo e utilize o método `write` do objeto `arquivo` para salvar o arquivo no disco, utilizando o `SERVER_PATH` concatenado ao caminho relativo que você recebeu por parâmetros e ao `submittedFileName` do objeto `arquivo`. Também coloque um `try / catch` para pegar a `IOException`, não permitindo que ela quebre o encapsulamento do `FileSaver`.



89.3k xp

Lembre-se de retornar o caminho relativo, que será o `path` recebido concatenado ao `submittedFileName` do `arquivo`.





Opinião do instrutor



74%

ATIVIDADES
8 de 14FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARD

```
public class FileSaver {  
  
    private static final String SERVER_PATH = "/casadocodi  
  
    public String write(Part arquivo, String path) {  
        String relativePath = path + "/" + arquivo.getSubPartName();  
        try {  
            arquivo.write(SERVER_PATH + "/" + relativePath);  
  
            return relativePath;  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

89.3k xp



Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 5

- Atividade 9 Obtendo a Capa do FileSystem | Alura

Video Player is loading.

Current Time 0:00

Duration 18:00

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

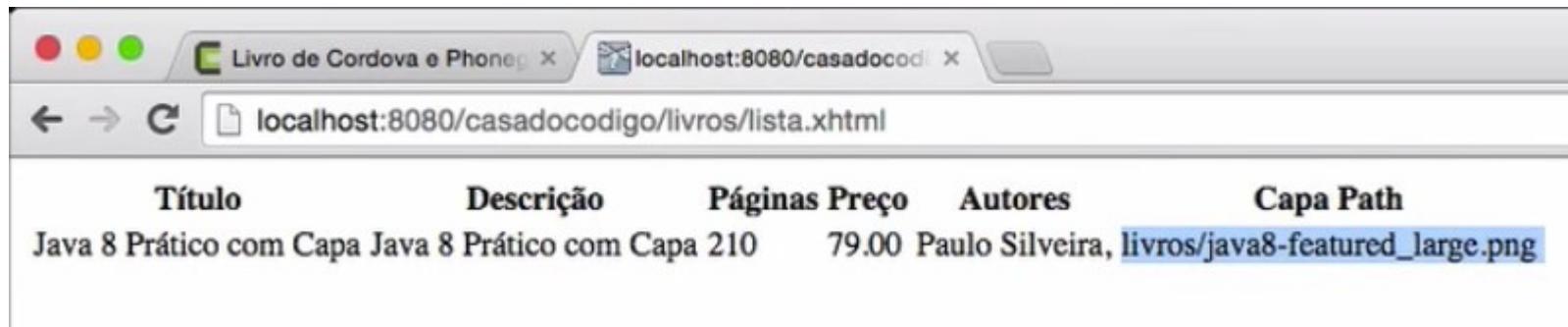
Uma vez que já estamos enviando o arquivo para o servidor, precisamos de uma forma de recuperar os dados do mesmo.

Acessaremos a página `lista.xhtml`, e em `<dataTable>`, adicionaremos mais uma coluna, esta será o *Path* da capa do

Livro. Antes de , vamos inserir o seguinte código:

```
<h:column>
    <f:facet name="header">Capa Path</f:facet>
    #{livro.capaPath}
</h:column>
```

Faça *Full Publish* da aplicação e veja como ficou a exibição da capa na tela.



Título	Descrição	Páginas	Preço	Autores	Capa Path
Java 8 Prático com Capa	Java 8 Prático com Capa	210	79.00	Paulo Silveira,	livros/java8-featured_large.png

Aparentemente a foto foi salva, mas na hora de exibir, não faz sentido exibir apenas o path relativo da foto. O ideal é exibir a própria foto. Mas acabamos caindo em uma limitação, pois estamos salvando a foto fora do *Application Server*, em uma pasta específica do **S.O.**. Então mesmo que usássemos o caminho relativo, ele não iria funcionar.

Para que o sistema pegue esse arquivo de dentro do **S.O.**, precisaremos tratar isso dentro da aplicação. Faremos isto, utilizando *Servlet* normal da especificação de *Servlets*. Em seguida, criaremos uma nova classe no pacote `br.com.casadocodigo.servlets` e daremos o nome da classe de `FileServlet`. Faça a nova classe herdar de `HttpServlet` e já sobrescrever o método `service()`. Todo *Servlet* precisa de um mapeamento, no nosso caso, faremos via *annotation* usando o `@WebServlet` e informando o caminho que desejamos mapear. No nosso caso, usaremos o mapeamento `/file`.

Um detalhe interessante é a forma como nosso *servlet* será chamado. Do jeito que fizemos até aqui, para chamar nosso *Servlet*, precisaremos passar algum parâmetro com o nome do arquivo que queremos carregar, por exemplo:

```
http://localhost:8080/casadocodigo/file?p=livros/java-8-featured_large.png
```

Mas desta forma, o carregamento ficará estranho . Seria melhor fazer:

```
.../file/livros/java-8-featured_large.png
```

Assim, ficaria parecendo que estamos acessando realmente o arquivo. Conseguiremos esse resultado simplesmente

adicionado um /* (barra asterisco) no fim do mapeamento já feito. Assim a base de nosso *Servlet* ficará assim:

```
@WebServlet("/file/*")
public class FileServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        // código do tratamento do arquivo irá aqui!
    }

}
```

Dentro do método `service()`, conseguimos pegar o que vier pelo * por meio do objeto `req.getRequestURI()`. O método nos retorna toda a URL, mas só nos interessa o que vier depois de /file, então faremos um `split("/file")` e do Split teremos dois lados, valor [0] será o que vem antes do /file e [1] o que vem depois. Queremos este último, uma vez que é isso que temos salvo no banco de dados. Desta forma, já teremos o *path*.

```
String path = req.getRequestURI().split("/file")[1];
```

Para que o arquivo seja enviado no `response` do *Servlet*, precisaremos informar o `contentType` do arquivo. Neste caso específico, sabemos que se trata de uma imagem, porém queremos deixar o `FileServlet` mais genérico. Usaremos a API nova do **Java de NIO** para pegar o `contentType` direto do arquivo `Paths.get("caminho completo do arquivo")`. No entanto, temos que acessar de fato o arquivo e, até o momento, só temos o caminho relativo do arquivo. Onde foi que guardamos o arquivo dentro do servidor?

Podemos juntar o `path` que temos com o caminho fixo do nosso `FileSaver` para conseguirmos o caminho completo do arquivo. Passaremos as duas informações para o `Paths` e assim obtemos um `Path` como sendo a fonte do nosso arquivo.

```
Path source = Paths.get(FileSaver.SERVER_PATH + "/" + path);
```

O `Path` servirá como fonte para o `FileNameMap` conseguir chegar no arquivo e obter o `contentType`. O `FileNameMap` pode ser obtido usando a classe `URLConnection` chamando o método estático da classe `getFileNameMap`.

```
FileNameMap fileNameMap = URLConnection.getFileNameMap();
```

Pelo `fileNameMap`, chamaremos o método `getContentTypeFor()` passando nosso `source`. Só temos antes que informar que o protocolo de acesso é `file:` para que o `FileNameMap` possa pegar o arquivo corretamente. Assim sendo, temos:

```
String contentType = fileNameMap.getContentTypeFor("file:"+source);
```

Agora sim, poderemos setar o `contentType` no nosso `response`:

```
res.setContentType(contentType);
```

Mas parece que tivemos tanto trabalho apenas para setar um valor no `response`. Será que valeu a pena?

Todo navegador verifica sempre o *Header* da resposta do servidor para saber o que ele deve fazer. Quando você abre um PDF no `Chrome` por exemplo, ele possui um *leitor* interno de PDF's, e já é possível ao usuário ler o arquivo sem ter que abri-lo no seu computador na mão. Isso vale para outros tipos de arquivos, imagens, vídeos, e etc. Assim, é muito importante dizer ao navegador qual o tipo de conteúdo que estamos enviando para ele, e ele se ajustará a esse tipo de conteúdo.

Outro *Header* que é importante informar, é o tamanho do arquivo ou `Content-Length`, o que também ajuda o navegador a baixar corretamente o arquivo. Usaremos outra classe da API de **NIO do Java**, que é a classe `Files`.

```
res.setHeader("Content-Length", String.valueOf(Files.size(source)));
```

E por fim, o *Header* que coloca o nome correto do arquivo que estamos baixando, que é o `Content-Disposition`.

```
res.setHeader("Content-Disposition",
    "filename=\""+source.getFileName().toString() + "\"");
```

Ainda usaremos esse `Content-Disposition` mais adiante. Por enquanto, são esses os *Headers* que precisamos informar. Parece trabalhoso fazer tudo isso manualmente, mas não temos escolha... Até o momento, o **JavaEE** não possui nenhuma forma de obter esses dados automaticamente.

Usando **JSF** ainda temos mais uma coisa a fazer, que é limpar o `response` antes de setar qualquer cabeçalho. Lembre-se que o **JSF** pode ter recebido esse `request` e assim, já podemos ter colocado alguma informação no `response`, por isso é importante limpá-lo sempre que usarmos o `response`, evitando resultados inesperados.

```
res.reset();
// Antes de setar qualquer Header.
```

Agora estamos prontos. Mas ainda não transferimos o arquivo de fato, apenas preparamos o `response` para isso. Usaremos o `FileSaver` que já temos e ele cuidará da operação. Como queremos transferir o arquivo do servidor para o `response`, criaremos um método estático dentro de `FileSaver` chamado `transfer()`. Esse método tem a seguinte assinatura:

```
public static void transfer(Path source, OutputStream outputStream) {
    // código que manipula o arquivo
}
```

Já podemos chamá-lo no `FileServlet`, então vamos adicionar a chamada dele e logo depois veremos como implementar a transferência. Nosso método `service()` do *Servlet* ficou assim:

```
protected void service(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    String path = req.getRequestURI().split("/file")[1];

    Path source = Paths.get(FileSaver.SERVER_PATH + "/" + path);
    FileNameMap fileNameMap = URLConnection.getFileMap();
    String contentType = fileNameMap.getContentTypeFor("file:"+source);

    res.reset();
    res.setContentType(contentType);
    res.setHeader("Content-Length", String.valueOf(Files.size(source)));
    res.setHeader("Content-Disposition",
        "filename=\"" + source.getFileName() + "\"");
    FileSaver.transfer(source, res.getOutputStream());
}
```

Agora, voltando ao nosso `FileSaver`, o primeiro passo da transferência é realizar a entrada do arquivo, do servidor para o sistema. Usaremos a classe `FileInputStream` para isso. Passaremos o `FileInputStream` para a classe `Channels.newChannel` que abre um canal direto com o arquivo, retornando o objeto `ReadableByteChannel`.

Além do canal de entrada, precisaremos de um canal de saída. Vamos usar o `Channels.newChannel` novamente e passaremos o `OutputStream` que recebemos como parâmetro. O canal nos retornará um `WritableByteChannel`.

Como os dois recursos `ReadableByteChannel` e `WritableByteChannel` são canais ligados direto ao arquivo e ao `response` do servidor, precisaremos fechar os recursos. Desde o **Java 7**, temos a possibilidade de usar o `try-with-resources` que automaticamente já fecha um recurso após o fim do `try`. O código ficará assim:

```
FileInputStream input = new FileInputStream(source.toFile());
try( ReadableByteChannel inputChannel = Channels.newChannel(input);
      WritableByteChannel outputChannel = Channels.newChannel(outputStream)) {
    // código que transfere o arquivo.
}
```

Além da entrada e saída, a transferência de um lado para o outro deve ser feita sempre usando um `Buffer`. Nesta, os arquivos serão transferidos em pedaços. No caso, vamos transferir 10kb por vez. Para criar nosso `Buffer` usaremos a classe `ByteBuffer.allocateDirect` passando como parâmetro `1024 * 10` que representa os 10Kb.

```
public static void transfer(Path source, OutputStream outputStream) {
    try {
        FileInputStream input = new FileInputStream(source.toFile());
        try( ReadableByteChannel inputChannel = Channels.newChannel(input);
              WritableByteChannel outputChannel = Channels.newChannel(outputStream)) {
            ByteBuffer buffer = ByteBuffer.allocateDirect(1024 * 10);

            } catch (IOException e) {
```

```
        throw new RuntimeException(e);
    }
} catch (FileNotFoundException e) {
    throw new RuntimeException(e);
}
}
```

Vamos começar a ler do nosso canal de entrada e transferir para o buffer. Faremos isto, enquanto existirem bytes para serem lidos. Usaremos um `while` para isso, e o próprio `inputChannel` possui um método chamado `read()` que nos retornará o valor `-1` quando não houver mais bytes a serem lidos.

```
while(inputChannel.read(buffer) != -1) {
    outputChannel.write(buffer);
    buffer.clear();
}
```

Depois de adicionarmos o `while()`, o trecho ficará da seguinte maneira:

```
public static void transfer(Path source, OutputStream outputStream) {
    try {
        FileInputStream input = new FileInputStream(source.toFile());
        try( ReadableByteChannel inputChannel = Channels.newChannel(input);
             WritableByteChannel outputChannel = Channels.newChannel(outputStream)) {
            ByteBuffer buffer = ByteBuffer.allocateDirect(1024 * 10);

            while(inputChannel.read(buffer) != -1) {
                outputChannel.write(buffer);
                buffer.clear();
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    } catch (FileNotFoundException e) {
        throw new RuntimeException(e);
    }
}
```

Com os bytes no buffer, podemos enviar para o `outputChannel`, só que dessa vez não queremos ler, e sim escrever na saída. Faremos isto, usando o método `write()` e passando para ele o `buffer` com os bytes lidos. Depois que escrevermos no `buffer`, queremos que ele seja limpo usando o `buffer.clear()`. Desta forma, ele poderá voltar a ler mais informações do arquivo.



10 Criando coluna para a Capa do Livro

[PRÓXIMA ATIVIDADE](#)

77%

ATIVIDADES
10 de 14FÓRUM DO
CURSOVOLTAR
PARA
DASHBOARDMODO
NOTURNOABRIR
CADERNO

89.4k xp



Com a capa do livro sendo salva no sistema de arquivo, podemos fazer a exibição dessa capa. Para isso vamos usar a tag `` normal do HTML.

Abra o arquivo `lista.xhtml` e ao final da `<h:datatable>`, vamos adicionar uma nova coluna. Nela, nomeie o `header` da coluna de `Capa Path` e dentro coloque a tag `` com o valor do `src` para `/file/#{livro.capaPath}`. Para a imagem não ficar gigante, você pode colocar a altura de 30% da imagem `height="30%"`.

Nossa lista ainda não vai exibir a imagem corretamente, pois não existe o endereço `/file/...` que informamos. No próximo exercício vamos criá-lo.

Opinião do instrutor

A nova coluna da listagem, deve ficar assim:

```
<h:column>
    <f:facet name="header">Capa</f:facet>
    
</h:column>
```

[COPIAR CÓDIGO](#)

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 5

- Atividade 11 Criando o Servlet de File | Alura

Atender a requisição `/file/...` é simples, mas para isso, precisamos criar um `Servlet`.

Crie uma nova classe no pacote `br.com.casadocodigo.servlets` chamada `FileServlet`. Faça a classe herdar de `HttpServlet` e *sobrecreva* o método `service`.

O primeiro passo é mapear essa `Servlet`. Anote essa classe com `@WebServlet`. Dentro da *annotation*, coloque o valor `"/file/*"`. Esse valor significa que qualquer coisa começando com `/file/` será atendido pelo nosso `Servlet`.

Dentro do método `service`, obtenha a URL requisitada, dividindo o método do `request` pelo `/file`, e do array retornado, pegamos a posição **1** que é a posição após o `/file`.

Crie o `Path` e use o `FileNameMap` para obter o `contentType`. Para criar o `Path`, precisamos do caminho absoluto para o arquivo. Já temos o caminho relativo, só precisamos concatenar o local do servidor onde o arquivo está salvo. No `FileSaver` já temos o caminho do servidor, mas está em `private`. Vamos mudar a constante `SERVER_PATH` para ser `public`, o que não é um problema para nós, já que ela não pode ser alterada.

Os próximos passos são um pouco mais "burocráticos", já que teremos que tratar o `response` na mão.

Primeiro devemos limpar o `response` para remover possíveis sujeiras, depois informamos qual o `contentType`, *setamos* o Header para o tamanho `Content-Length` e por fim dizemos qual será o nome do arquivo com o Header `Content-Disposition`.

Mas ainda não pegamos o arquivo do disco para colocar no `response`. Para isso, vamos usar o `FileSaver` novamente, chamando `FileSaver.transfer`, e para ele, chame passe o `Path` que criamos antes e o `OutputStream` do `response`.

Ainda não criamos o método `transfer` do `FileSaver`, vamos fazer isso no próximo exercício.

Opinião do instrutor

O `FileServlet` que você criou, deve ficar parecido com o código abaixo.

```
@WebServlet("/file/*")
public class FileServlet extends HttpServlet {
```

```
@Override  
protected void service(HttpServletRequest req, HttpServletResponse res)  
    throws ServletException, IOException {  
    String path = req.getRequestURI().split("/file")[1];  
  
    Path source = Paths.get(FileSaver.SERVER_PATH + "/" + path);  
    FileNameMap fileNameMap = URLConnection.getFileMap();  
    String contentType = fileNameMap.getContentTypeFor("file:"+source);  
  
    res.reset();  
    res.setContentType(contentType);  
    res.setHeader("Content-Length", String.valueOf(Files.size(source)));  
    res.setHeader("Content-Disposition",  
        "filename=\""+source.getFileName().toString() + "\"");  
    FileSaver.transfer(source, res.getOutputStream());  
}  
}
```

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 5

- Atividade 12 Transferindo o arquivo do servidor para o response | Alura

Quase tudo pronto, só precisamos do método `transfer` da nossa classe `FileSaver` funcionando. Vamos lá.

Abra a classe `FileSaver` e nela crie um método público e `static`, que não deve possuir retorno chamado `transfer`. Os parâmetros que vamos passar são os que a classe `FileServlet` já usa. Por isso, receba um `Path` que chamamos de `source` e um `OutputStream` que chamamos de `outputStream`.

Dentro do método, vamos chamar o `new FileInputStream(source.toFile())` e já o colocar em uma variável. Agora temos que manipular os arquivos, e isso não é tão simples, por isso, segue abaixo o código do `try-with-resources`.

```
try {
    FileInputStream input = new FileInputStream(source.toFile());
    try( ReadableByteChannel inputChannel = Channels.newChannel(input);
        WritableByteChannel outputChannel = Channels.newChannel(outputStream) ) {
        ByteBuffer buffer = ByteBuffer.allocateDirect(1024 * 10);

        while(inputChannel.read(buffer) != -1) {
            buffer.flip();
            outputChannel.write(buffer);
            buffer.clear();
        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
} catch (FileNotFoundException e) {
    throw new RuntimeException(e);
}
```

Assim, esse código é bem tenso, e conforme explicamos no vídeo, ele pega o arquivo físico e passa para o `OutputStream` do `HttpServletResponse`.

Com isso, fechamos o método `transfer`.

Opinião do instrutor

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 5

- Atividade 13 Exibindo a Capa do Livro na Listagem | Alura

Video Player is loading.

Current Time 0:00

Duration 4:34

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Depois de escrito no `outputChannel`, precisamos limpar o `buffer` para que ele possa ler novamente quando voltar ao fluxo do `while` que criamos.

Como já lemos novamente os bytes do `buffer` e entramos novamente no `while`, temos mais um passo a fazer. Todo

buffer possui um ponteiro que após temos escrito no buffer pode ter ficado no final do mesmo, e para garantir que escreveremos no `outputStream` tudo que está no buffer, precisamos chamar o método `buffer.flip` que colocará o ponteiro na posição zero novamente. Assim, a primeira linha dentro do `while` será `buffer.flip()`

Nosso `while` completo ficou bem simples, graças ao `Channels` de entrada e de saída.

```
while(inputChannel.read(buffer) != -1) {  
    buffer.flip();  
    outputChannel.write(buffer);  
    buffer.clear();  
}
```

Além do `try-with-resources`, o `FileInputStream` nos obriga a verificar uma `Exception`, então faremos o `catch()` dele fora do nosso `try-with-resources` e apenas jogaremos a exceção pra cima, porém encapsulada em uma `RuntimeException()`.

Nosso método `transfer()` ficou assim:

```
public static void transfer(Path source, OutputStream outputStream) {  
    try {  
        FileInputStream input = new FileInputStream(source.toFile());  
        try( ReadableByteChannel inputChannel = Channels.newChannel(input);  
             WritableByteChannel outputChannel = Channels.newChannel(outputStream)) {  
            ByteBuffer buffer = ByteBuffer.allocateDirect(1024 * 10);  
  
            while(inputChannel.read(buffer) != -1) {  
                buffer.flip();  
                outputChannel.write(buffer);  
                buffer.clear();  
            }  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
    } catch (FileNotFoundException e) {  
        throw new RuntimeException(e);  
    }  
}
```

Aparentemente temos tudo pronto, mas para que possamos ver de fato nossa capa, vamos abrir novamente nossa `lista.xhtml` e no lugar de exibir apenas o caminho da nossa capa, vamos exibir de fato a imagem de capa do livro, mas redimensionando para um tamanho que não quebre nossa lista de Livros.

Adicione a tag `` e no atributo `src` coloque o valor como sendo `"#{request.contextPath}/file/#{livro.capaPath}"`. Ajuste o atributo `altura` para `height="30%"` e o texto alternativo como sendo o título

do livro alt="#{livro.titulo}"` . Assim, o código de exibição da imagem de capa ficou:

```
<h:column>
    <f:facet name="header">Capa Path</f:facet>
    
</h:column>
```

Faça novamente um *Full Publish* e cadastre um livro com uma imagem real e depois verifique na listagem se sua imagem realmente aparece corretamente na tela.

Ganhamos muita flexibilidade no nosso sistema, cadastrando as capas do livro e lendo esse arquivo de capa depois. Além disso, ainda fizemos a exibição dos arquivos como parte da URL, ao invés de ficar passando parâmetros, sujando nossas URL's do sistema.

Agora vá para os exercícios e tire suas dúvidas no fórum. Bons estudos!

Java EE parte 1: Crie sua loja online com CDI, JSF, JPA: Aula 6

- Atividade 1 Adicionando a Home Page da CDC | Alura

Video Player is loading.

Current Time 0:00

Duration 17:38

1x

- 2x
- 1.75x
- 1.5x
- 1.25x
- 1x, selected
- 0.75x
- 0.5x
- 0.25x

Transcrição

Adicionando a Home Page da CDC

Nosso sistema administrativo e o formulário já estão muito bons. Então vamos tentar fazer com que nossa HomePage comece a ser exibida, e tenhamos uma página inicial bem atrativa.

The screenshot shows a web browser displaying the website <https://www.casadocodigo.com.br>. The page features a header with the site's logo and navigation links. Below the header, a large orange banner with the text 'Últimos lançamentos' is visible. Four book cards are displayed in a row:

- Zend Certified Engineer**: Descomplicando a certificação PHP. It features a purple seal with an elephant icon and the text 'Certified PHP'. The authors are MATHEUS MARABESI and MICHAEL DOUGLAS.
- Web Services REST**: com ASP.NET Web API e Windows Azure. It shows a blue cloud icon with a hand cursor and several lowercase letters (a, z, u, r, e, c) falling from it. A yellow 'COMPRAR' button is at the bottom.
- jQuery Mobile**: Desenvolva interfaces para múltiplos dispositivos. It shows a smartphone connected via a cable to a computer monitor, both displaying mobile interface prototypes.
- Elastic**: Consumindo com ELK. Only the start of the title and a green decorative bar are visible.

Faremos a exibição dos livros em destaque separando-os dos demais livros, assim como é feito na loja da Casa do Código. Queremos a mesma home, porém não iremos criar tudo novamente, pois seria muito trabalhoso. Você pode baixar o pacote com a home da *CDC* no link abaixo:

<https://s3.amazonaws.com/caelum-online-public/java-ee-webapp/casadocodigo-javaee-home.zip>

Você já encontrará cerca de 15 livros cadastrados no pacote, com as capas, descrição e nome dos autores.

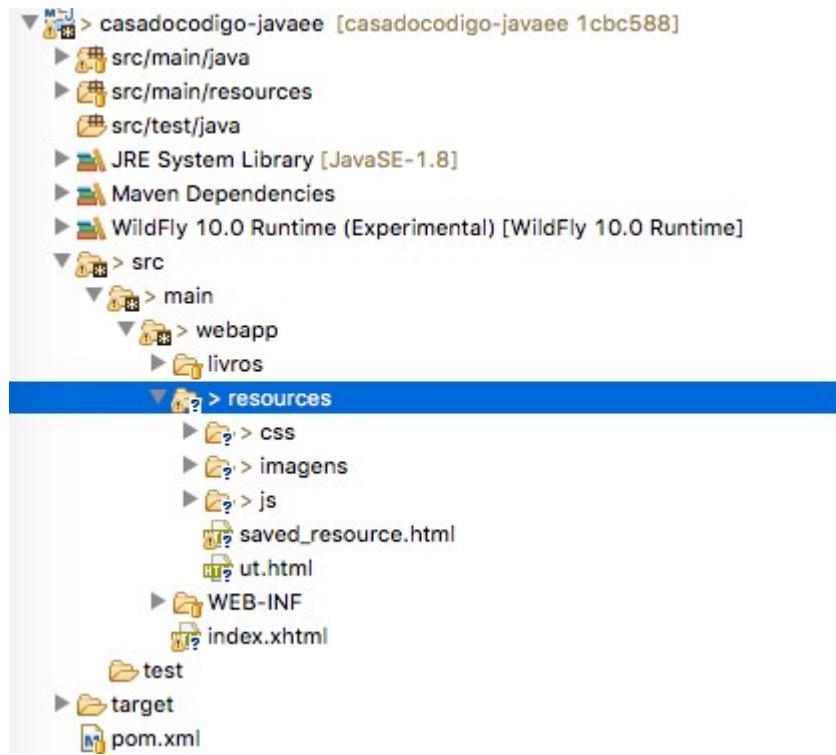
The screenshot shows a web browser window with the URL localhost:8080/casadocodigo/livros/lista.xhtml. The page displays a list of books:

- Introdução à Computação**: "Em uma sociedade global que esbanja desenvolvimento tecnológico, aprender a programar passa a ser um conhecimento est transformados em linhas de código se tornam programas, jogos, sites e aplicativos. O homem pensa em ir até Marte e mergu ciência. Na fronteira de tudo isso, está o código. Neste livro, Guilherme Silveira ensina as principais matérias introdutórias c computação, fazendo você criar seus primeiros jogos de computador. Entenda como seu programa toma decisões e domine c com recursão. Decifre toda a sequência de um programa de computador entendendo a sua pilha de execução e domine a lóg software."
- Spring MVC**: "O Spring é o principal concorrente da especificação JavaEE. Com uma plataforma muito estável e com integração fina entre fornece um ambiente muito propício para que o programador foque nas regras de negócio e esqueça dos problemas de infra será construída uma aplicação baseada na loja da Casa do Código e você terá a chance de utilizar diversas das funcionalidad pelo framework. Usaremos o Spring MVC como alicerce da nossa aplicação web e para implementar todas as funcionalidad integrações, como: Spring JPA, para facilitar o acesso ao banco de dados; Spring Security, para segurança da aplicação; dife com a parte de Profiles; respostas assíncronas para melhorarmos a escalabilidade; e ainda detalhes, como cache e suporte às estilo REST. Tudo isso sem uma linha de XML, todas configurações serão feitas baseadas em anotações e código Java."
- Explorando APIs e bibliotecas Java**: "Uma vez que você aprende Orientação a Objetos e q básico do Java, é necessário se tornar fluente em suas APIs, que não sã programador Java eficaz conhece o que há disponível e sabe quando pode usar cada uma das possibilidades que a linguagem queremos para você. Rodrigo Turini ensina como lidar com as diferentes APIs do Java com exemplos práticos e que são relé Você vai aprender as diferentes APIs para fazer IO, trabalhar com threads da forma correta, se conectar com banco de dados ferramentas de build e muito mais."

Após baixar o arquivo, delete o atual arquivo `index.html` que temos na nossa aplicação e vamos importar um novo arquivo com uma estrutura já preparada para o nosso projeto.

Ainda no Eclipse, clique com o botão direito sobre o projeto e selecione a opção `Import`. Vá em `General` e depois em `Archive File` e depois vá em `Next`. Nessa tela, cliquem em `Browse...` que fica logo a frente do campo `From archive file:` e navegue até o local onde você baixou o `zip` da home da CDC, clique em `OK`. Você deve estar vendo uma estrutura de pastas no lado esquerdo e no lado direito o arquivo `index.xhtml`. Antes de finalizar, na caixa `Into folder` procure pelo caminho `src/main/webapp` e depois clique `OK`. Por fim, clique em `Finish` para finalizar o processo.

Você já deve estar vendo os arquivos importados.



Vamos testar se nosso `index.xhtml` já está funcionando. *Full Publish* do projeto e acesse: <http://localhost:8080/casadocodigo/index.xhtml>. Navegue um pouco na aplicação para brincar, mas lembre-se que alguns links estão enviando para a loja oficial da Casa do Código.

Como todos os livros estão com a mesma capa, percebemos que ainda não está sendo considerada os livros cadastrados no banco de dados. Vamos fazer com que nossos livros cadastrados apareçam de fato na Home.

Abra o `index.xhtml` e vá para a linha 217. A partir da segunda tag `` do arquivo, selecione todos os demais `` até antes do fechamento da tag `` de modo que sobre ao menos 1 tag `` para servir de repetição do nosso `for`.

Logo acima do `` vamos colocar uma tag `<ui:repeat value="" var="livro">`. Já colocamos o nome do `var` como sendo **livro** e vamos realizar algumas modificações no meio do código.

Dentro da tag `<a>` procure pelo atributo `title` e modifique para `title="#{livro.titulo}"`. No atributo `` procure pelos atributos `alt` e `title` e coloque os valores `alt="#{livro.titulo}" title="#{livro.titulo}"` e dentro da tag `` também modifique o título do livro atual para `#{{livro.titulo}}`.

Como estamos trabalhando na parte de últimos lançamentos da Home, vamos criar um `ManagedBean` que será

responsável por trazer os 5 livros que fazem parte dos últimos lançamentos. Chamando esse Bean de HomeBean e podemos ter um método chamado ultimosLancamentos. Antes de criar efetivamente nosso Bean, podemos já adicionar essa informação ao nosso ui:repeat. Nossa código de repetição dos últimos lançamentos ficará assim:

```
<ui:repeat value="#{homeBean.ultimosLancamentos()}" var="livro">
    <li class="livroNaVitrine vitrineDestaque-produto">
        <a href="https://www.casadocodigo.com.br/products/livro-certificacao" class="livroNaVitrine-link"
            title="#{livro.titulo}">
            <div class="livroNaVitrine-imagemContainer" role="presentation">
                
            </div>
            <span class="livroNaVitrine-nome">#{livro.titulo}</span>
        </a>
    </li>
</ui:repeat>
```

Já referenciamos o HomeBean, então vamos criá-lo. Após usarmos o comando Ctrl + N, selecionaremos Class e daremos o nome da classe de HomeBean. Depois, colocaremos a classe no pacote de beans. Anote a classe com @Model como já fizemos anteriormente e também crie um método chamado ultimosLancamentos que retorna um List<Livro>. Dentro do nosso método vamos precisar de um LivroDao, então já vamos injetar o LivroDao no nosso Bean e chamamos dentro do nosso método dao.ultimosLancamentos(). Nossa HomeBean até o momento ficará assim:

```
package br.com.casadocodigo.loja.beans;

import javax.enterprise.inject.Model;

@Model
public class HomeBean {

    @Inject
    private LivroDao dao;

    public List<Livro> ultimosLancamentos() {
        return dao.ultimosLancamentos();
    }
}
```

Como o LivroDao ainda não possui o método ultimosLancamentos, vamos criá-lo pressionando Ctrl + 1 e escolhendo a opção "Create method ...". O Eclipse deve ter criado para você o método, já com a assinatura tudo certinho.

Dentro do método criado, vamos colocar um JPQL para obter os livros do banco de dados. Além disso, como queremos apenas os últimos 5 resultados, vamos pedir para o EntityManager realizar essa limitação, utilizando o

setMaxResults. O método `ultimosLancamentos()` completo ficará assim:

```
public List<Livro> ultimosLancamentos() {  
    String jpql = "select l from Livro l order by l.id desc";  
    return manager.createQuery(jpql, Livro.class)  
        .setMaxResults(5)  
        .getResultList();  
}
```

Observe que o `order by l.id desc`, com o qual estamos buscando os livros ordenados inversamente pelo ID do Livro. Assim garantiremos que os últimos cadastros serão retornados pela query. Poderíamos usar a data de lançamento para isso, mas como no nosso caso não fará diferença, sinta-se à vontade pra usar qualquer um dos dois.

Vamos testar nossa aplicação. Faremos um *Full Publish* e veremos se nova *Home* está aparecendo corretamente e listando os últimos cinco livros.



Voltando para a `index.xhtml`, vamos realizar uma alteração para os demais livros. Dentro do `vitrineDaColecao`

(próximo da linha 238), removeremos todos os `` que estiverem dentro de `` mantendo apenas 1 que servirá para repetirmos. Logo acima do `` que sobrou, adicione a tag `<ui:repeat>` da mesma forma que acima, mudando apenas o `value` para:

```
value="#{homeBean.demaisLivros()}"
```

Nosso código final ficará assim:

```
<ul class="vitrineDaColecao-lista">
    <ui:repeat value="#{homeBean.demaisLivros()}" var="livro">
        <li class="livroNaVitrine vitrineDaColecao-produto">
            <a href="https://www.casadocodigo.com.br/products/livro-certificacao" class="livroNaVitrine-link"
                title="#{livro.titulo}">
                <div class="livroNaVitrine-imagemContainer" role="presentation">
                    
                </div>
                <span class="livroNaVitrine-nome">#{livro.titulo}</span>
            </a>
        </li>
    </ui:repeat>
</ul>
```

Esse novo método no `Bean` precisa ser criado. Vamos criar um novo método na classe `HomeBean` chamado `demaisLivros()` retornando `List<Livro>` também. Dentro dele, apenas a chamada para `return dao.demaisLivros();` que ainda não existe no `LivroDao`. Seguiremos o mesmo esquema de antes, `Ctrl + 1` e escolha a opção `Create method` Já dentro do método na classe `LivroDao`, criaremos a **JPQL** que cuidará do carregamento dos demais livros, e será exatamente igual a anterior. Mas se ela é igual a anterior, o que mudará dos demais livros para os últimos livros?

Nós limitamos os últimos lançamentos aos cinco primeiros resultados. Se já temos os cinco, não queremos que eles se repitam nos demais livros, certo? Então, vamos mudar os parâmetros do `EntityManager`. Vamos retirar a limitação dos últimos resultados, pois agora queremos todos. Mas queremos todos **sem os cinco primeiros**, e para isso vamos informar a propriedade `setFirstResult(5)`, que é onde dizemos que os resultados considerados precisam começar a partir do sexto item encontrado, apesar de informarmos o número 5, pois a indexação começa do 0 (zero). E como já temos os cinco primeiros, realmente queremos apenas do sexto até o último.

```
public class LivroDao {
    // Demais métodos acima

    public List<Livro> demaisLivros() {
        String jpql = "select l from Livro l order by l.id desc";
        return manager.createQuery(jpql, Livro.class)
    }
}
```

```
        .getResultList();
    }

}
```

Faça novamente um *Full Publish* e vamos entrar na nossa aplicação pelo navegador. Tudo deve estar funcionando corretamente.

Nossa aplicação está bem elegante, mas ainda queremos que as capas dos livros exibidas sejam de fato as capas que cadastramos. Para essa simples alteração, basta abrir o `index.xhtml` e dentro do `` procurar pela tag ``. Perceba que essa tag está apontando para algum caminho interno como `resources/....` O que temos que fazer é alterar para *invocar* o nosso `FileServlet` que já criamos na aula anterior. Para isso, altere o atributo `src` para `src="#{request.contextPath}/file/#{livro.capaPath}"`. A tag `` completa ficará assim:

```

```

Lembre-se de alterar para os **Últimos Lançamentos** bem como para os **Demais Livros**.

Faça novamente um *Full Publish* do projeto e ao subir o servidor, navegue pela **Home** da aplicação. Todas as capas devem estar corretas e os títulos também correspondentes. Assim, nossa `Home` está praticamente idêntica a da **CDC** e agora temos uma página bem elegante.