Criando uma aplicação Java utilizando JSF, JPA e CDI

# Desenvolvendo um sistema para gerenciamento de biblioteca

Se olharmos para a plataforma e para a linguagem Java desde as primeiras versões podemos notar uma enorme evolução. Com o passar dos anos foram surgindo diversas tecnologias, frameworks e especificações que fazem parte desse processo evolutivo.

Nesse artigo iremos conhecer um pouco mais sobre três importantes especificações Java – JSF, JPA e CDI – e desenvolver uma aplicação que utilize implementações dessas especificações para vermos como funciona a integração entre elas. Iremos entender porque surgiram, quais vantagens trazem e quais tipos de problemas resolvem.

A aplicação que será desenvolvida ao longo do artigo serve para fazer o gerenciamento de uma biblioteca. Nela poderão ser gerenciadas por exemplo as informações das editoras, dos livros, dos leitores, dos funcionários da biblioteca e dos empréstimos dos livros.

|  |
| --- |
| **Fique por dentro** |
| Será desenvolvida do zero uma aplicação para gerenciamento de biblioteca, uitlizando JSF, JPA e CDI, mostrando como integrar as três coisas em um projeto. Serão abordados recursos como o PhaseListener do JSF e as anotações @Inject, @Produces, @Disposes e @Interceptor relacionadas a CDI, a JSR 330 e a especificação de Interceptors dentre outras coisas. Iremos utilizar como servidor de aplicação o Jboss Wildfly e será mostrado como configurá-lo. Usaremos Java 8 e será mostrado como utilizar uma das classes da nova API de data, a LocalDate, dentro das entidades e como contornar o problema de que ainda não existe uma forma de mapear esse tipo de classe na JPA. |

# Especificações Java

Foram citadas três especificações Java no texto introdutório do artigo, mas quem está começando a programar em Java pode não estar familiarizado com o termo “especificação”. O que seria isso? Ao longo do tempo surgiram tecnologias Java, algumas podemos até chamar de frameworks, que tem mais de uma implementação. Podemos pegar como exemplo os frameworks ORM(Object/Relational Mapping) como é o caso do Hibernate, EclipseLink, OpenJPA entre outros. Para que exista uma padronização entre essas implementações é criada uma especificação, que nada mais é do que um documento, que garante todas elas tenham um comportamento igual ou bem parecido e que contemplem as funcionalidades especificadas. No caso dos frameworks ORM foi criada a especificação JPA. Dessa forma se a implementação que está sendo usada em algum projeto começa a apresentar bugs ou é descontinuada, podemos trocá-la por outra sem muito esforço. As vezes as implementações surgem antes da especificação, que vem depois para padronizar tudo, e nesse caso as implementações tem que se adquar ao que foi especificado. Um exemplo seria o Hibernate que surgiu antes da especificação JPA.

Para quem tiver interesse em saber mais sobre o processo de criação das especificações, pode acessar o site da Java Community Process(JCP), e na Wikipédia também pode ser encontrada uma boa explicação resumida(veja os endereços na seção **Links**).

# Sobre o JSF

A especificação JSF especifica como deve funcionar o framework JSF. O JSF, acrônimo de JavaServer Faces, pode ser definido como um framework Java MVC padrão para construir interfaces com o usuário baseadas em componentes para aplicações web. Portanto cada implementação do JSF como o Mojarra e o MyFaces, implementa as funcionalidades do framework.

O JSF surgiu de uma necessidade de termos uma forma mais simples, eficáz e produtiva de criarmos interfaces gráficas para aplicações web. Atualmente inclusive, conseguimos criar aplicações web com recursos visuais complexos de forma simples ao utilizar por exemplo o Primefaces, uma das bibliotecas de componentes mais famosas para JSF 2. O Primefaces possui centenas de componentes ricos que podem ser adicionados facilmente a aplicação, porém nesse artigo não vamos utilizá-los pois iremos nos ater aos componentes padrões do JSF.

Além do JSF facilitar muito a criação da interface gráfica, ele provê uma maneira padrão de solucionar problemas comuns que encontramos durante o desenvolvimento de aplicações web como: validações, conversões, criação de templates, etc.

Antes de chegarmos a origem do JSF vamos voltar um pouco mais no tempo e olhar para as tecnologias que vieram antes.

API de Servlets

A API de Servlets contribuiu para a popularização do Java pois era amplamente usada no desenvolvimento de aplicações web. Até hoje as Servlets funcionam “por baixo dos panos” no caso dos frameworks mais novos, inclusive no caso do JSF, e fazem parte da infraestrutura das aplicações web, mas naquela época eram usadas diretamente para gerar dinamicamente as páginas de resposta as requisições dos usuários.

No entanto essa forma de criar conteúdo dinâmico era propensa a erros e complicada pois misturava código HTML com código Java considerando que uma Servlet é uma classe Java. Veja um exemplo de código que poderia aparecer em uma Servlet:

out.println(“<body text=\”black\” bgcolor=\”blue\” link=\”yellow\”>”);

Para piorar a situação no caso acima ainda temos que usar o caracter de “escape” barra invertida(\) para “escaparmos” as ocorrências de aspas duplas(“).

JavaServer Pages

O próximo passo na evolução das tecnologias Java web foi a introdução das JavaServer Pages. Essa tecnologia consistia em criar páginas com extensão .jsp, que aceitavam tanto conteúdo HTML como porções de código Java(scriptlets, declarations e expressions).

Apesar da tecnologia JavaServer Pages ter significado uma grande melhoria na forma de gerar conteúdo dinâmico para ser apresentado ao usuário, pois as páginas JSP eram praticamente páginas HTML com tagas especiais para incluir código Java, ainda assim continuávamos com códigos HTML e Java misturados no mesmo arquivo.

Visando resolver esse problema surgiu a JSTL(Java Standard Tag Library) e a EL(Expression Language). Assim os códigos Java poderiam ser eliminados das páginas JSP.

Com a introdução da JSTL e da EL era possível implementar o MVC nas aplicações Java web e separar as responsabilidades de cada componente. Classes Java comuns representavam a camada Model, as páginas JSP representavam a camada View e as Servlets representavam a camada Controller.

No entanto apesar de implementar o MVC manualmente dessa forma ter se tornado comum na época e deixar as aplicações mais organizadas, ainda era muito trabalhoso desenvolver usando JSP e Servlets. Eis que surgiram os frameworks MVC.

Frameworks MVC

Ao falar de frameworks MVC, podemos dar destaque ao Struts, que por um bom tempo foi muito utilizado. Uma das razões pela grande adoção do Struts pela comunidade Java foi que ele resolvia de forma inteligente o problema relacionado a separar a aplicação em camadas. Antes do Struts, com JSP e Servlets sem um framework, muitos desenvolvedores, principalmente os inexperientes, acabavam criando o máu hábito de colocar código Java na camada de visualização. Já com o Struts, devido a sua arquitetura MVC, os desenvolvedores eram empelidos a separar o código de apresentação(View) do restante do código(Controller e Model).

Apesar da boa arquitetura do Struts, ainda era muito trabalhoso e custoso desenvolver as interfaces gráficas. Então foi criado o JSF para que se tivesse um framework que também fosse MVC mas que facilitasse o trabalho de criação da camada de apresentação. Na época que o JSF surgiu o Struts era o framework mais utilizado, mas o JSF aos poucos foi ganhando seu espaço.

Para quem está acostumado com outros frameworks como o Struts, Spring MVC ou Vraptor por exemplo, pode estranhar um pouco a forma como o JSF funciona, pois estes outros frameworks são considerados Action Based enquanto o JSF é considerado Component Based. Mas não se preocupe, na parte prática do artigo, irá ser apresentado o “jeitão” JSF de ser.

# Sobre a JPA

A especificação Java Persistence API, ou simplesmente JPA, foi criada para padronizar a forma como os frameworks ORM devem trabalhar. A especificação define uma série de anotações e interfaces, e os frameworks ORM devem implementar essas interfaces, para que todos eles possam ser utilizados através de uma única API. A especificação também dita o que é esperado que cada método faça, e a implementação fica a cargo de cada JPA provider.

Agora vamos ver o que motivou a criação dos frameworks de mapeamento objeto-relacional.

A grande maioria das aplicações atualmente usam algum banco de dados relacional. Quando falamos de bancos de dados relacionais logo pensamos em tabelas, relacionamentos, chaves primárias, chaves estrangeiras, etc. Já quando o assunto é linguagem orientada a objetos logo pensamos em classes, composição, herança, polimorfismo, etc. Ao utilizarmos JDBC para converter o conteúdo de um ResultSet em objetos, conseguimos notar as similaridades entre uma classe e uma tabela, mas apesar das semelhanças existem também diversas diferenças entre o modelo OOP e o relacional. Muitas vezes alguma coisa que existe no paradigma relacional não existe no orientado a objetos, e vice versa, o que remete a uma certa complexidade para fazer os dois modelos se conversarem, exigindo que o desenvolver contorne essas diferenças programaticamente fazendo diversos ajustes no código. A essas diferenças fundamentais entre os dois paradigmas é dado o nome de Impedância (impedance mismatch).

Em Java, uma das formas de se trabalhar com bancos de dados é utilizar a API JDBC. Um dos problemas ao utilizarmos JDBC é que precisamos escrever código SQL misturado com nosso código Java. Se for necessário alterar alguma instrução SQL será necessário recompilar a classe na qual foi feita a alteração. Diante disso, alguns programadores adotaram estratégias como remover os códigos SQL da aplicação, extraindo-os para um arquivo txt ou xml. Surgiram até frameworks como o IBatis para ajudar a realizar esse isolamento.

Outro problema que podemos enfrentar ao utilizar JDBC e escrever os SQLs diretamente em nossas classes Java é se precisarmos mudar de banco de dados. Apesar de existir o padrão ANSI, instruções SQL podem apresentar diferenças significativas dependendo do SGBD utilizado, o que dificultaria a tarefa de migrar de um banco para outro.

Se olharmos para as dificuldades e problemas descritos, podemos concluir que faria sentido se trabalhássemos apenas com a nossa linguagem de programação, usando somente o paradigma OOP, e que fosse abstraído de nós a responsabilidade de escrever SQL ou fazer conversões entre o modelo orientado a objetos e o relacional. Foi justamente para isso que foram criados os frameworks ORM como o Hibernate, EclipseLink, Apache OpenJPA dentre outros. A sigla ORM significa Object/Relational Mapping, ou Mapeamento Objeto/Relacional em português, e como o nome sugere é feito um mapeamento entre um modelo e outro, para que possa haver uma conversão automática entre os dois modelos(OOP e relacional). A idéia é que utilizando esse tipo de framework não precisaremos mais nos preocupar em escrever instruções SQL, passando a trabalhar somente com objetos.

# Sobre a CDI

A especificação CDI, cujo nome completo é Context and Dependency Injection for Java EE, padroniza o processo de injeção de depêndencias(em inglês Dependency Injection, ou simplesmente DI) dentro do Java. Antes da CDI existiam vários frameworks que realizavam injeção de dependências como o Spring, Jboss Seam, Google Guice, PicoContainer, dentre outros, mas cada um deles funcionava de um jeito diferente e não existia uma forma padrão de fazer a injeção das dependências. A especificação inclusive sofreu várias influências dos frameworks que já existiam, principalmente do Jboss Seam 2 e do Google Guice.

Uma vantagem no uso de CDI se comparado ao uso do Jboss Seam 2, é que a injeção das dependências é feita de modo type safe, e no caso do Seam 2 é feita baseada em Strings.

A CDI(JSR 299) foi construída em cima da JSR 330(Dependency Injection for Java) e as duas especificações são complementares. A JSR 330 define um conjunto de anotações(@Inject, @Named, @Qualifier, @Scope e @Singleton) relacionadas a injeção de dependências.

Dependency Injection é uma das formas de se aplicar uma técnica muito mais ampla chamada Inversão de Controle(em inglês Inversion of Control, ou simplemente IoC). A idéia principal da IoC é inverter o controle das chamadas dos métodos da aplicação em relação a programação convencional, ou seja, ela não é determinada diretamente pelo programador. Alguns frameworks usam o conceito de inversão de controle e controlam as chamadas de métodos específicos, que no caso do Java geralmente são marcados com alguma anotação. Quando usamos DI também ocorre inversão de controle, mas de uma forma um pouco diferente da citada acima, pois nesse caso invertemos o controle da criação das dependências que passa a não ser feita mais por nós, ou seja, já recebemos as dependências instanciadas e prontas para serem usadas.

A principal vantagem de se usar injeção de de dependências é que esse design pattern diminui o acoplamento entre as classes e suas dependências. Felizmente ao utilizar CDI acabamos utilizando esse padrão de projeto sem mesmo perceber, de forma transparente, e podemos focar nas regras de negócios da aplicação que estamos desenvolvendo e não precisamos nos preocupar em como as dependências serão satisfeitas.

Para que o leitor entenda a importância de se diminuir o acoplamento entre as classes(e a implementação da CDI que escolhermos irá nos ajudar nisso) precisa entender o significado de acoplamento. Então vamos revisar dois conceitos importantes na orientação a objetos que são o acoplamento e a coesão.

A coesão está relacionada ao que uma classe sabe fazer, se a classe foi criada de forma que tenha um único e bem focado propósito dizemos que ela é coesa, no entanto se a classe sabe fazer muitas coisas dizemos que ela é pouco coesa. A alta coesão é algo desejável enquanto a baixa coesão é algo que devemos evitar.

O acoplamento está relacionado a forma como uma classe se relaciona e interage com as outras. Vejamos alguns exemplos para ajudar no entendimento. Se a classe A tem um método que recebe um objeto do tipo da classe B podemos dizer que esse acoplamento entre as duas classes é menor do que se a classe A tivesse como membro uma variável de instância do tipo da classe B; Teríamos um acoplamento ainda maior que nos dois casos anteriores se a classe A herdasse da classe B. Outra situação seria se a classe A conhece a classe B unicamente por meio do que a classe B expôs através de sua interface, então poderíamos dizer que o acoplamento entre as duas é fraco. Mas se a classe A conhece detalhes de implementação da classe B e depende de partes da classe B que não fazem parte do que foi exposto em sua interface, então poderíamos dizer que o acoplamento entre elas é mais forte do que deveria.

O interessante é que a coesão e o acoplamento se interrelacionam de certa forma. À medida que nos esforçamos para termos classes com alta coesão em nossa aplicação, deixamos de ter classes que sabem fazer muitas coisas, e consequentemente teremos mais relacionamentos entre nossas classes especializadas, o que propicia o aumento do acoplamento. Porém se nos esforçarmos em excesso para diminuirmos o acoplamento, consequentemente voltaremos a ter classes que sabem fazer muitas coisas, ou seja, teremos menos classes se relacionando mas classes pouco coesas.

Devemos sempre tentar manter a alta coesão e o baixo acoplamento.

Suponhamos que estamos desenvolvendo nossa aplicação para gerencimento de bibliotecas e criamos uma classe chamada AutorBean que tem as seguintes linhas de código:

EntityManager entityManager = ...código para conseguir um entityManager aqui

AutorDao autorDao = new AutorDaoJpa(entityManager);

Note que a classe AutorBean depende de um AutorDao, ou seja, o AutorDao é sua dependência. Porém a classe AutorBean está resolvendo sua dependência e escolhendo uma implementação para AutorDao, que seria a classe AutorDaoJpa. Observe também que a classe AutorDaoJpa tem uma dependência que seria um entityManager, que também está sendo resolvida manualmente. Se já é ruim para a classe AutorBean gerenciar suas próprias dependências, imagine nesse caso que ela está gerenciando uma dependência de sua dependência.

Um problema que pode surgir também é se alguém alterar a assinatura do construtor da classe AutorDaoJpa modificando a lista de argumentos que ela recebe, pois nesse caso a classe AutorBean irá parar de compilar.

Outro agravante que podemos ter com esse código é se um dia por algum motivo quisermos trocar a implementação de AutorDao e decidirmos usar um AutorDaoJdbc no lugar do AutorDaoJpa. Se as dependências são gerenciadas pela nossa aplicação, como nesse exemplo, teremos que alterar o código inteiro manualmente, procurando todo lugar onde é usado um AutorDaoJpa e substituir por um AutorDaoJdbc.

Os problemas que observamos acima são decorrentes do forte acoplamento entre nossa classe de exemplo e sua dependência, pois a classe conhece a implementação concreta da dependência e ainda precisa instância-la. Tudo poderia ficar ainda pior se crescesse o número de dependências. Se pensarmos bem a responsabilidade de instanciar as dependências nem deveria recair sobre o programador, pois não fazem parte da lógica da aplicação.

Para solucionarmos esses problemas poderíamos utilizar Dependency Injection, e é ai que CDI entra na jogada. Se fóssemos fazer um código equivalente ao apresentado, mas utilizando CDI, poderíamos injetar um AutorDao na classe AutorBean:

@Inject

AutorDao autorDao;

Na classe AutorDaoJpa poderíamos injetar o entityManager direto no construtor:

@Inject

AutorDaoJpa(EntityManager entityManager){

...

}

Para que a injeção do entityManager acima funcione é preciso configurar uma classe produtora de entityManagers, mas veremos mais sobre isso na hora de construir nossa aplicação para bibliotecas.

Ao criarmos nossas classes utilizando DI é comum configurarmos a injeção das dependências pelo construtor, como no código acima. Com isso, a classe apenas espera que alguém injete as dependências que ela precisa. Isso facilita muito na hora de criarmos os testes de unidade, pois podemos injetar manualmente mock objects – ver **BOX 1** – pelo construtor da classe utilizando bibliotecas como o Mockito(ver seção **LInks**), então em suma esse tipo de estratégia aumenta a testabilidade de nossas classes.

|  |
| --- |
| **Box 1.** mock objects |
| mock objects são objetos falsos que simulam o comportamento de objetos reais de forma controlada. É possível “ensinar” um mock object a se comportar da forma que precisamos. São normalmente criados para testar o comportamento de outros objetos. |

Vimos duas principais vantagens de se usar injeção de dependências(diminuir o acoplamento entre as classes e aumentar sua testabilidade), e consequentemente temos essas vantagens ao utilizar CDI. Existem outras vantagens no uso de DI, mas não vamos entrar nesse mérito.

Na próxima seção abordaremos o desenvolvimento da aplicação para gerenciamento de biblioteca.

# Desenvolvendo a aplicação

Ao desenvolvermos uma aplicação é comum termos muitos cadastros parecidos, muitas classes que fazem parte do DAO parecidas, etc. Não iremos exibir os códigos de todos arquivos por esse motivo, pois são códigos que se repetem e não agregam grande valor ao artigo. Nas situações que não foi possível omitir o arquivo todo, omitimos parte do código. Ao longo do artigo iremos apresentar os códigos mais importantes dos principais arquivos para que o leitor possa rodar o projeto com o mínimo necessário e no final ter um sistema de biblioteca que execute as duas principais operações que seriam empréstimo e devolução de livros. Uma versão reduzida da aplicação igual a apresentada no artigo está disponível para download no Github, no entanto uma versão completa com todos arquivos na íntegra também está disponível para download no Github(os endereços de ambas versões da aplicação constam na **seção Links**).

# Configurando o ambiente de desenvolvimento

Downloads e instalações

Para desenvolvermos nossa aplicação iremos usar a versão 8 do Java, portanto precisará baixar e instalar o JDK 8(vide seção **Links**). Como IDE utilizaremos o Eclipse Luna para Java EE(vide seção **Links**). O servidor de aplicação escolhido para usarmos foi o Jboss Wildfly versão 8.1.0.Final(vide seção **Links**). A implementação da especificação JSF que usaremos é a Mojarra, mas não será necessário fazer o download pois ela já vem junto com o Wildfly. A implementação da especificação JPA que usaremos é o Hibernate, que também já vem com o Wildfly. A implementação da especificação CDI que usaremos é o Weld, que também já vem com o Wildfly. Usaremos o MySQL como banco de dados(vide seção **Links**). Também será necessário fazer o download do driver do MySQL(vide seção **Links**).

Após baixar e instalar os softwares acima poderá seguir para o próximo passo que é a configuração do Jboss.

Configurando o Jboss WildFly

O Wildfly passui uma arquitetura baseada em módulos. Primeiramente iremos criar um módulo no Jboss que irá armazenar o driver do MySQL. Na pasta wildfly-8.1.0.Final\modules\ deverá ser criada a seguinte estrutura de pastas: com\mysql\main. É obrigatório que tenhamos um diretório main, caso contrário o Wildfly não irá invocar nosso módulo. O diretório com\mysql\main deverá conter o driver do MySQL(por exemplo mysql-connector-java-5.1.32-bin.jar) e um arquivo module.xml com o conteúdo presente na **Listagem 1**.

**Listagem 1.** module.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<module xmlns=*"urn:jboss:module:1.0"* name=*"com.mysql"*>

<resources>

<resource-root path=*"mysql-connector-java-5.1.32-bin.jar"*/>

</resources>

<dependencies>

<module name=*"javax.api"*/>

<module name=*"javax.transaction.api"*/>

</dependencies>

</module>

No atributo path da tag resource-root deve ser informado o nome do arquivo jar do MySQL(o driver) que está no mesmo diretório.

Devemos registrar o driver no arquivo wildfly-8.1.0.Final\standalone\configuration\standalone.xml. Entre as tags <drivers> e </drivers> colocaremos uma referência ao módulo que criamos anteriormente que contém o driver do MySQL conforme **Listagem 2**.

**Listagem 2.** Trecho do arquivo standalone.xml com a configuração do driver MySQL

<drivers>

<driver name=*"com.mysql"* module=*"com.mysql"*>

<xa-datasource-class>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</xa-datasource-class>

</driver>

</drivers>

O próximo passo é criarmos um DataSource para podermos conectar na base de dados através dele. No mesmo arquivo wildfly-8.1.0.Final\standalone\configuration\standalone.xml entre as tags <datasources> e </datasources> deve ser colocada a configuração do DataSource conforme **Listagem 3**.

**Listagem 3.** Trecho do arquivo standalone.xml com a configuração do DataSource

<datasources>

<datasource jta=*"false"* jndi-name=*"java:/bibliotecaDS"* pool-name=*"bibliotecaPool"* enabled=*"true"* use-java-context=*"true"*>

<connection-url>jdbc:mysql://localhost:3306/bibliotecaDb</connection-url>

<driver>com.mysql</driver>

<pool>

<min-pool-size>10</min-pool-size>

<max-pool-size>100</max-pool-size>

<prefill>true</prefill>

</pool>

<security>

<user-name>root</user-name>

<password>root</password>

</security>

</datasource>

</datasources>

Repare que na tag driver na configuração do DataSource deve ser inserido o mesmo nome que foi definido na quando registramos o driver do MySQL nesse mesmo arquivo.

Criando o banco de dados

Depois de instalado o MySQL poderemos utilizar o “MySQL Command-Line Tool” (vide seção **Links**), que é um programa que funciona no Prompt de Comando do Windows e serve para conectar no servidor MySQL. A partir de uma janela do Prompt de comando realizamos a conexão com o comando abaixo:

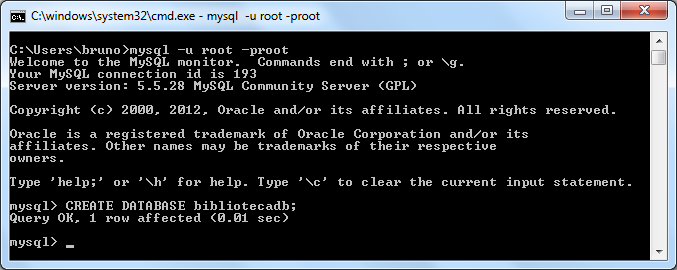
mysql –u usuario –psenha

Podemos substituir a palavra usuario por root, que é o usuário com mais privilégios no MySQL, e substituir a palavra senha pela senha do root que foi definida na instalação do MySQL.

Depois de conectar, criamos o banco de dados que iremos utilizar em nosso projeto através da instrução SQL abaixo:

CREATE DATABASE bibliotecadb;

A **Figura 1** mostra a conexão sendo estabelecida e o banco de dados sendo criado.

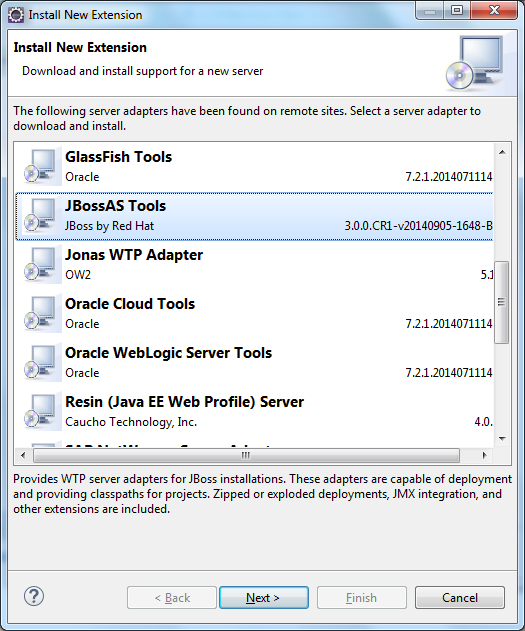


**Figura 1.** Banco de dados sendo criado através do “MySQL Command-Line Tool”

Instalando o JbossAS Tools no Eclipse

Temos que fazer a instalação do JbossAS Tools para podermos trabalhar com o Wildfly dentro do Eclipse.

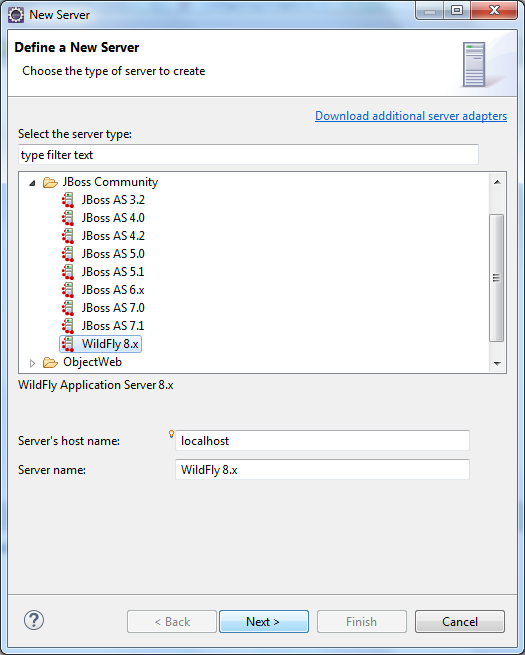
Na aba “Servers” adicionaremos um novo servidor(botão direito -> New -> Server). Na tela que se abre temos que clicar no opção “Download additional server adapters”. Iremos procurar pelo “JbossAS Tools”(veja **Figura 2**). Devemos prosseguir com a instalação e aceitar os termos da licença. Após a instalação o Eclipse será reiniciado.



**Figura 2.** Instalação do JbossAS Tools

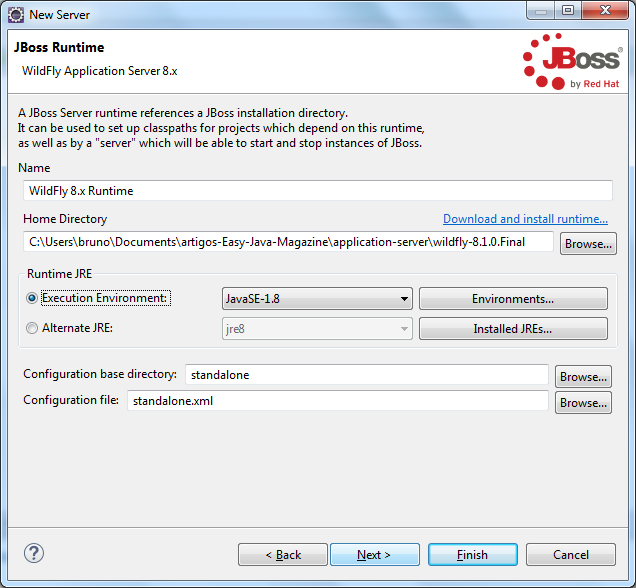
Adicionando o Jboss Wildfly no Eclipse

Após a instalação do JbossAS Tools conseguiremos adicionar o Wildfly na aba “Servers” (botão direito -> New -> Server). Dessa vez já irá aparecer a opção “Wildfly 8.x” que é a que usaremos(veja **Figura 3**).



**Figura 3.** Adicionando Wildfly 8.x no Eclipse

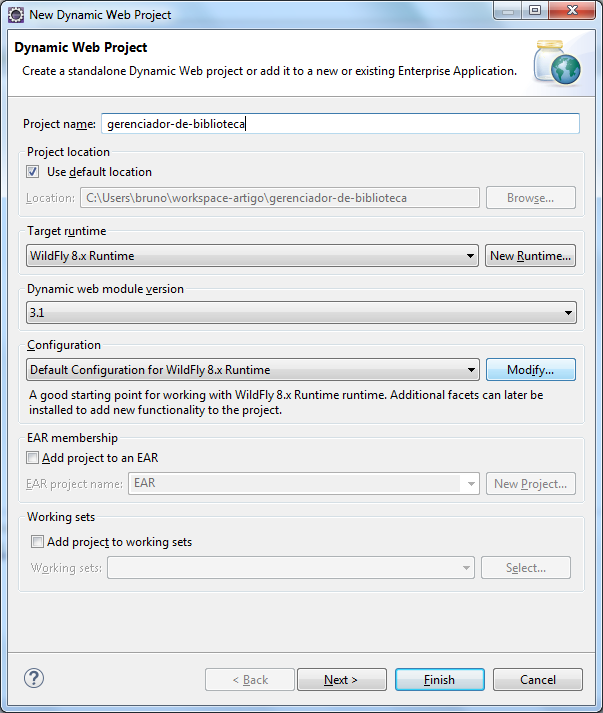
Em uma das etapas será solicitado que informemos o Home Directory do Jboss Wildfly 8.x, e como já o baixamos diretamente do site do Wildfly podemos apenas informar o caminho onde ele está(veja **Figura 4**). O resto das opções podemos manter as configurações padrão e prosseguirmos até o término.



**Figura 4.** Informando o caminho do Jboss Wildfly(seu Home Directory)

Criando o projeto no Eclipse

Criaremos um novo Dynamic Web Project. Na primeira tela do wizard, iremos dar o nome de gerenciador-de-biblioteca ao nosso projeto e configurá-lo conforme **Figura 5**.



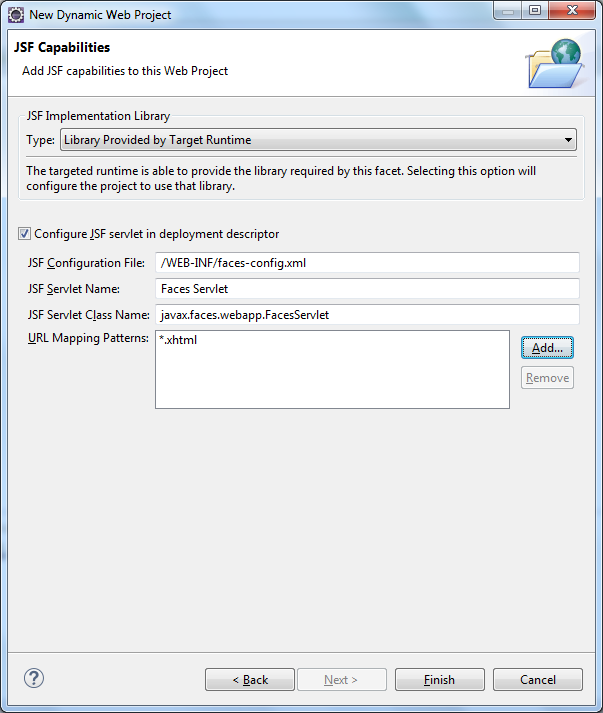
**Figura 5.** Configuração do projeto

Ainda na primeira tela do wizard, devemos clicar no botão “Modify...” na seção “Configuration” e será aberta a tela “Projet Facets” onde devemos manter 3 opções marcadas:

* “Dynamic Web Module” versão 3.1
* “Java” versão 1.8
* “JavaServer Faces” versão 2.2

Na etapa “Web Module” do wizard marcaremos a opção “Generate web.xml deployment descriptor”.

Na etapa “JSF Capabilities” a configuração deve ficar igual a da **Figura 6**. Note que na opção “JSF Implementation Library” informamos que os jars que contém a implementação do JSF serão fornecidos pelo servidor de aplicação. Também perceba que na opção “URL Mapping Patterns” removemos o mapeamento /faces/\* e inserimos o mapeamento \*.xhtml, pois nossas páginas JSF terão extensão xhtml. Depois de configurada essa etapa podemos finalizar.



**Figura 6.** Configurações do JSF do projeto

# Configurando o persistence.xml

O nosso persistence provider, no caso o Hibernate, precisa de um arquivo chamado persistence.xml que contém os dados para conexão com o banco de dados ou no nosso caso qual DataSource iremos utilizar, as classes que representam nossas entidades, dentre outras coisas.

Dentro da pasta src de nosso projeto devemos criar uma pasta chamada META-INF, e dentro dela criar o arquivo persistence.xml que deverá ter o conteúdo mostrado na **Listagem 4**.

**Listagem 4.** Conteúdo do arquivo persistence.xml

<persistence xmlns=*"http://java.sun.com/xml/ns/persistence"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd"*

version=*"2.0"*>

<persistence-unit name=*"bibliotecaPersistence"* transaction-type=*"RESOURCE\_LOCAL"*>

<provider>org.hibernate.ejb.HibernatePersistence</provider>

<jta-data-source>java:/bibliotecaDS</jta-data-source>

<class>br.com.easyjavamagazine.entidades.Autor</class>

<class>br.com.easyjavamagazine.entidades.Categoria</class>

<class>br.com.easyjavamagazine.entidades.Editora</class>

<class>br.com.easyjavamagazine.entidades.Emprestimo</class>

<class>br.com.easyjavamagazine.entidades.Endereco</class>

<class>br.com.easyjavamagazine.entidades.FuncionarioBiblioteca</class>

<class>br.com.easyjavamagazine.entidades.Leitor</class>

<class>br.com.easyjavamagazine.entidades.Livro</class>

<class>br.com.easyjavamagazine.entidades.Pessoa</class>

<class>br.com.easyjavamagazine.entidades.Telefone</class>

<class>br.com.easyjavamagazine.entidades.Usuario</class>

<class>br.com.easyjavamagazine.conversores.LocalDateConversor</class>

<properties>

<property name=*"hibernate.dialect"* value=*"org.hibernate.dialect.MySQL5InnoDBDialect"* />

<property name=*"hibernate.show\_sql"* value=*"true"* />

<property name=*"hibernate.format\_sql"* value=*"false"* />

<property name=*"hibernate.hbm2ddl.auto"* value=*"create"* />

</properties>

</persistence-unit>

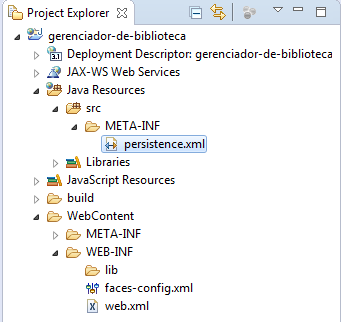
</persistence>

Declaramos no persistence.xml algumas entidades que ainda não foram criadas, mas não se preocupe, isso será a próxima coisa que faremos.

Devemos entender para que serve a propriedade hibernate.hbm2ddl.auto e sempre dar uma atenção especial a essa propriedade, principalmente quando a aplicação for para produção. Podemos configurar os seguintes valores:

* create: cria as tabelas necessárias para persistirmos nossas entidades, destruindo qualquer dado existente.
* create-drop: cria as tabelas igual a opção create, porém remove as tabelas no final da sessão.
* update: faz com que as alterações feitas nas entidades sejam refletidas nas tabelas do banco, por exemplo ao adicionar um novo atributo em uma entidade será criada a respectiva coluna na tabela.
* validate: valida se as tabelas da base de dados estão de acordo com nossas entidades, porém não efetua nenhuma alteração no banco, então podemos considerar essa a opção mais segura para ser usada em ambiente de produção.

A estrutura do projeto nesse momento está conforme a **Figura 7**.



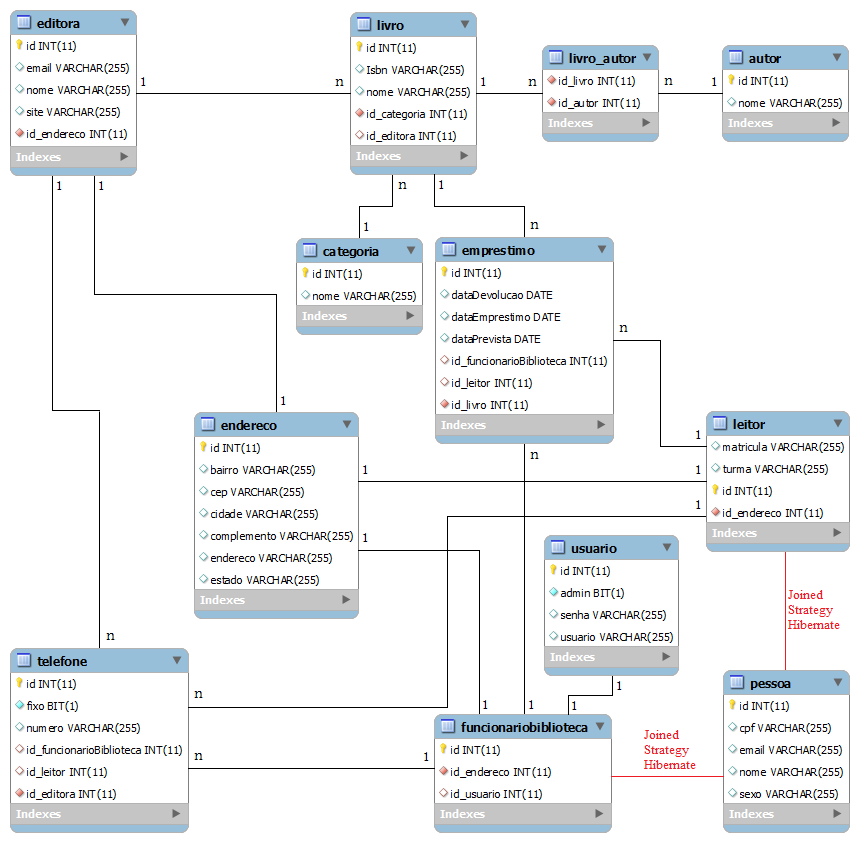
**Figura 7.** Estrutura do projeto

# Criando e mapeando as entidades

Dentro da JPA existe o conceito de entidade. Uma entidade, nada mais é do que um POJO(Plain Old Java Object) cujos atributos que não são marcados com a anotação @Transient devem ser persistidos em um banco de dados relacional através de um EntityManager. Em outras palavras uma entidade representa os dados armazenados em um banco de dados relacional. Dessa forma trabalhamos na grande parte do tempo com nossos objetos e não com SQL, e as mudanças feitas em nossas objetos irão refletir no banco de dados. Assim a JPA provê um mecanismo simples de manipularmos os dados que estão em nosso banco.

Devemos indicar quais de nossas classes são entidades e fazer o mapeamento objeto/relacional, para que o JPA provider usado saiba como fazer as conversões entre o mundo OOP e o mundo relacional. Podemos fazer isso usando anotações ou por meio de configurações em arquivos XML(orm.xml e persistence.xml). Iremos utilizar anotações.

A **Figura 8** demonstra um diagrama que representa como deverá ficar nosso banco de dados depois de pronto. São mostradas as tabelas( assim como suas colunas e suas chaves primárias e estrangeiras), os relacionamentos entre as tabelas e a cardinalidade de cada relacionamento.



**Figura 8.** Diagrama de como deve ficar nosso banco de dados pronto

Olhando para um diagrama como esse fica mais fácil criarmos e mapearmos nossas entidades.

Veja que os relacionamentos entre as tabelas ocorrem por meio de chaves primárias(primary keys) e estrangeiras(foreign keys) normalmente. No entanto as tabelas pessoa, leitor e funcionariobiblioteca possuem apenas primary keys e o Hibernate irá relaciona-las apenas por meio dessas chaves, pois nesse caso o mapeamento será feito usando a anotação @Inheritance(com a estratégia JOINED) juntamente com herança, mas iremos ver mais detalhes adiante.

Antes de criarmos nossas entidades é necessário entendermos que tipos de relacionamentos podem existir entre elas. Os relacionamentos podem ser unidirecionais ou bidirecionais. Vejamos alguns exemplos para melhor entendimento:

* OneToOne unidirecional: o relacionamento entre uma entidade Editora e uma entidade Endereco. Em nossa aplicação é certo que gostaríamos de poder recuperar o endereço de uma editora(editora.getEndereco()), mas não precisaremos recuperar a editora de um endereço(endereco.getEditora()).
* OneToOne bidirecional: o relacionamento entre uma entidade Usuario(que contém por exemplo um usuário e senha para logar no sistema) e uma entidade FuncionarioBiblioteca. Em nossa aplicação através do usuário logado poderemos descobrir qual o funcionário da biblioteca que está realizando alguma operação no sistema(usuario.getFuncionarioBiblioteca()). Também é interessante podermos fazer o inverso e saber qual o usuário de um determinado funcionário da biblioteca(funcionarioBiblioteca.getUsuario()), pois poderíamos por exemplo ter uma tela onde buscamos o funcionário pelo nome e mostramos seu usuário juntamente com os outros dados, ou permita que ele altere a senha do seu usuário.
* OneToMany unidirecional: o relacionamento entre entidades do tipo Leitor e do tipo Telefone. Faz sentido recuperarmos os telefones de um leitor(leitor.getTelefones()), mas não faz muito sentido recuperarmos o leitor de um telefone(telefone.getLeitor()).
* OneToMany bidirecional e ManyToOne bidirecional: o relacionamento entre entidades do tipo Livro e do tipo Emprestimo. Se o relacionamento for bidirecional podemos por exemplo gerar um relatório informando em quais datas o livro foi tomado emprestado ou devolvido(podemos usar livro.getEmprestimos() e iterar na lista de empréstimos recuperarndo as datas de empréstimo e devolução). Também poderemos recuperar todos empréstimos e descobrir qual livro foi emprestado em cada um dos empréstimos(emprestimo.getLivro()). É importante perceber que o tipo de relacionamento OneToMany bidirecional é no fundo a mesma coisa que o tipo de relacionamento ManyToOne bidirecional. São duas perspectivas diferentes de um mesmo conceito.
* ManyToOne unidirecional: o relacionamento entre entidades do tipo Livro e do tipo Editora. Em nosso sistema será possível recuperarmos a editora de um determinado livro(livro.getEditora()), porém não há necessidade de recuperarmos todos livros de uma determinada editora(editora.getLivros()). Caso existisse essa necessidade poderia ser implementado um relacionamento ManyToOne bidirecional.
* ManyToMany unidirecional: o relacionamento entre entidades do tipo Livro e do tipo Autor. Em nossa aplicação será possível recuperarmos os autores de um determinado livro(livro.getAutores()), porém não há necessidade de recuperarmos os livros de um determinado autor(autor.getLivros()).
* ManyToMany bidirecional: podemos citar novamente o relacionamento entre entidades do tipo Livro e do tipo Autor. Em nossa aplicação será implementado um relacionamento unidirecional entre essas duas entidades, mas nada impediria que configurássemos um relacionamento bidirecional se houvesse essa necessidade.

Como talvez já tenham percebido, se um relacionamento deve ser unidirecional ou bidirecional depende muito da necessidade da aplicação. Se o relacionamento bidirecional não é uma necessidade da aplicação é melhor é evitá-lo, pois com esse tipo de relacionamento o código fica um pouco mais complicado pois temos sempre que “amarrar” as duas pontas, mantendo um estado consistente para o nosso modelo. Por exemplo se fóssemos realizar um empréstimo, teríamos que setar o objeto livro em nosso objeto emprestimo, mas também teríamos que adicionar o objeto emprestimo a lista de empréstimos do objeto livro:

emprestimo.setLivro(livro);

livro.getEmprestimos().add(emprestimo);

Voltando a olhar para o nosso diagrama(**Figura 8**) conseguimos imaginar quais entidades precisaremos ter. São elas: Autor, Categoria, Editora, Emprestimo, Endereco, FuncionarioBiblioteca, Leitor, Livro, Pessoa, Telefone e Usuario.

Para organizar o projeto iremos colocar cada tipo de classe em um package diferente, no caso das entidades ficarão todas em br.com.easyjavamagazine.entidades.

**Listagem 5.** Entidade Autor

**package** br.com.easyjavamagazine.entidades;

**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;

//getters e setters

}

Na **Listagem 5** temos o código da entidade Autor, que nada mais é, do que uma classe Java comum com algumas anotações. A anotação @Entity que vai em cima da classe indica que essa classe é uma entidade. Como não estamos usando a anotação @Table para especificar o nome da tabela, será usado o nome da entidade que por padrão é o nome não qualificado da classe, portanto será criada uma tabela chamada autor para essa entidade. A anotação @Id serve para demarcar a chave primária, ou seja, será criada uma tabela chamada autor no banco de dados que terá uma coluna chamada id que será a chave primária. A anotação @GeneratedValue indica que o valor da chave primária deve ser gerado automaticamente, e de acordo com a stratégia(strategy) passada dentro da anotação o persistence provider irá se orientar para viabilizar isso, por exemplo criando uma coluna do tipo auto-increment/identity ou uma sequence no banco de dados. O provider irá utilizar a estratégia indicada se for suportada pelo banco de dados.

A JPA define quatro tipos de estratégia:

* @GeneratedValue(strategy=GenerationType.AUTO): indica que o persistence provider deve escolher a melhor estratégia para geração dos valores da PK de acordo com o banco de dados. É a opção com maior portabilidade. Quando usamos a anotação sem definirmos explicitamente uma estratégia, essa opção é a default.
* @GeneratedValue(strategy=GenerationType.IDENTITY): especifica o uso de uma coluna auto-increment/identity.
* @GeneratedValue(strategy=GenerationType.SEQUENCE): especifica o uso de uma sequence para geração dos valores.
* @GeneratedValue(strategy=GenerationType.TABLE): especifica que o persistence provider deve atribuir valores as chaves primárias de nossas entidades com base em uma tabela auxiliar que irá garantir que os valores das chaves sejam únicos.

Escolhemos a estratégia IDENTITY para geração automática dos valores da primary key da tabela autor.

**Listagem 6.** Entidade Categoria

**package** br.com.easyjavamagazine.entidades;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

@Entity

**public** **class** Categoria {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

**private** Integer id;

**private** String nome;

//getters e setters

}

**Listagem 7.** Entidade Endereco

**package** br.com.easyjavamagazine.entidades;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

@Entity

**public** **class** Endereco {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

**private** Integer id;

**private** String endereco;

**private** String complemento;

**private** String bairro;

**private** String estado;

**private** String cidade;

**private** String cep;

//getters e setters

}

**Listagem 8.** Entidade Telefone

**package** br.com.easyjavamagazine.entidades;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

@Entity

**public** **class** Telefone {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

**private** Integer id;

**private** String numero;

**private** **boolean** fixo;

//getters e setters

}

As **Listagens 6, 7 e 8** mostram os códigos das entidades Categoria, Endereco e Telefone. Os códigos dessas classes são bem simples e são similares ao código da classe Autor que já foi explicado, inclusive usam as mesmas anotações, então dispensam comentários.

**Listagem 9.** Entidade Editora

**package** br.com.easyjavamagazine.entidades;

**import** java.util.ArrayList;

**import** java.util.List;

**import** javax.persistence.CascadeType;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

**import** javax.persistence.JoinColumn;

**import** javax.persistence.OneToMany;

**import** javax.persistence.OneToOne;

@Entity

**public** **class** Editora {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

**private** Integer id;

**private** String nome;

@OneToMany(cascade={CascadeType.***ALL***})

@JoinColumn(name="id\_editora")

**private** List<Telefone> telefones = **new** ArrayList<Telefone>();

@OneToOne(cascade={CascadeType.***ALL***})

@JoinColumn(name="id\_endereco")

**private** Endereco endereco;

**private** String site;

**private** String email;

//getters e setters

}

Na **Listagem 9** temos o código da entidade Editora. A anotação @OneToMany especifica um relacionamento do tipo um para muitos entre uma editora e seus telefones. Uma editora poderá ter muitos telefones. A anotação @JoinColumn, colocada em cima da lista de telefones, define qual coluna da tabela telefone será a chave estrangeira, que nesse caso será a coluna id\_editora. A anotação @OneToOne especifica um relacionamento do tipo um para um entre uma editora e um endereço, ou seja, uma editora poderá ter apenas um endereço e um endereço poderá pertencer somente a uma editora. A anotação @JoinColumn, colocada em cima do endereço, indica que a coluna id\_endereco da tabela editora será chave estrangeira. Note que tanto na anotação @OneToMany como na anotação @OneToOne foi especificado um atributo cascade e é importante entender para que ele serve. Este atributo foi usado para que as operações executadas, através de um entityManager, nas instâncias de Editora também sejam executadas nas instâncias de Telefone e Endereco. Por exemplo se persistirmos uma editora no banco de dados através do método persist, seus telefones e seu endereço também serão persistidos:

//persistindo uma editora, e seus telefones e seu endereço também serão persistidos automaticamente

Editora editora = new Editora();

editora.getTelefones().add(new Telefone());

editora.setEndereco(new Endereco());

//persiste tudo com uma única invocação do método persist

entityManager.persist(editora);

**Listagem 10.** Entidade Emprestimo

**package** br.com.easyjavamagazine.entidades;

**import** java.time.LocalDate;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

**import** javax.persistence.JoinColumn;

**import** javax.persistence.ManyToOne;

@Entity

**public** **class** Emprestimo {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

**private** Integer id;

**private** LocalDate dataEmprestimo;

**private** LocalDate dataPrevista;

**private** LocalDate dataDevolucao;

@ManyToOne

@JoinColumn(name = "id\_livro")

**private** Livro livro;

@ManyToOne

@JoinColumn(name = "id\_leitor")

**private** Leitor leitor;

@ManyToOne

@JoinColumn(name = "id\_funcionarioBiblioteca")

**private** FuncionarioBiblioteca funcionarioBiblioteca;

//getters e setters

}

A **Listagem 10** mostra a entidade Emprestimo que representa uma empréstimo de um livro. Um empréstimo de um livro é realizado por um funcionário da biblioteca para um leitor, portanto as entidades Emprestimo, Livro, Leitor e FuncionarioBiblioteca se relacionam entre si. Um único livro pode ser emprestado várias vezes, um mesmo leitor pode tomar emprestado vários livros(para cada livro um empréstimo diferente no caso do nosso sistema) e um mesmo funcionário da biblioteca pode realizar vários empréstimos, então teremos três relacionamentos do tipo ManyToOne para expressar essas relações. Foi usada a classe LocalDate do Java 8 para definirmos o tipo da data do empréstimo, da data prevista para devolução e da data efetiva da devolução. Quando temos em nossa classe atributos do tipo java.util.Date ou java.util.Calendar podemos usar a anotação @Temporal da JPA para realizarmos o mapeamento, mas como estamos usando o tipo java.time.LocalDate teremos que criar uma classe que atuará como um conversor(mais detalhes serão explicados quando for mostrado o código dessa classe). Isso é necessário pois a anotação @Temporal por enquanto não suporta o tipo java.time.LocalDate.

**Listagem 11.** Entidade Livro

**package** br.com.easyjavamagazine.entidades;

**import** java.util.List;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

**import** javax.persistence.JoinColumn;

**import** javax.persistence.JoinTable;

**import** javax.persistence.ManyToMany;

**import** javax.persistence.ManyToOne;

**import** javax.persistence.OneToMany;

@Entity

**public** **class** Livro {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

**private** Integer id;

**private** String nome;

@ManyToOne

@JoinColumn(name="id\_editora")

**private** Editora editora;

@ManyToMany

@JoinTable(name="livro\_autor",

joinColumns={@JoinColumn(name="id\_livro")},

inverseJoinColumns={@JoinColumn(name="id\_autor")})

**private** List<Autor> autores;

@ManyToOne

@JoinColumn(name="id\_categoria")

**private** Categoria categoria;

**private** String Isbn;

@OneToMany(mappedBy="livro")

**private** List<Emprestimo> emprestimos;

//getters e setters

}

Na **Listagem 11** temos o código da entidade Livro. A anotação @ManyToOne, em cima da variável editora, indica um relacionamento do tipo muitos para um, pois podemos ter muitos livros publicados por uma única editora. A anotação @ManyToMany define um relacionamento do tipo muitos para muitos, pois muitos livros podem pertencer a um mesmo autor, e um livro pode ser escrito por mais de um autor. Em conjunto com a anotação @ManyToMany usamos a anotação @JoinTable para podermos configurar o nome da tabela de ligação que irá relacionar os livros e autores. Através dos atributos joinColumns e inverseJoinColumns configuramos as colunas da tabela de ligação que serão as chaves estrangeiras. Para ficar mais claro, teremos a tabela livro\_autor que terá as colunas id\_livro e id\_autor que serão as chaves estrangeiras, enquanto as chaves primárias estão nas tabelas livro e autor. Em cima da variável categoria, foi configurada a anotação @ManyToOne, para indicar que muitos livros podem pertencer a uma mesma categoria. Se olharmos na **Listagem 10** e na **Listagem 11** será evidente que temos um relacionamento bidirecional entre as entidades Livro e Emprestimo, e esse relacionamento é definido por meio da anotação @ManyToOne usada na classe Emprestimo e da anotação @OneToMany usada na classe Livro. Dentro de @OneToMany usamos o atributo mappedBy para indicar que a entidade livro é o lado inverso do relacionamento enquanto a entidadeEmprestimo é a dona do relacionamento. No site da DevMedia tem um ótimo post explicando mais sobre a annotation mappedBy(veja o endereço na **seção Links**).

**Listagem 12.** Entidade Usuario

**package** br.com.easyjavamagazine.entidades;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

**import** javax.persistence.OneToOne;

@Entity

**public** **class** Usuario {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

**private** Integer id;

@Column(unique=**true**)

**private** String usuario;

**private** String senha;

**private** **boolean** admin = **true**;

@OneToOne(mappedBy="usuario")

**private** FuncionarioBiblioteca funcionarioBiblioteca;

//getters e setters

}

A **Listagem 12** mostra o código da entidade Usuario. Com as explicações que foram dadas nas listagens anteriores já é possível entender o papel de cada anotação utilizada nesse código, com exceção da anotação @Column que ainda não foi abordada. Estamos usando @Column com o atributo unique para assegurar que não existam dois nomes de usuário iguais e portanto será criada uma constraint no banco que irá garantir que não haja esse tipo de duplicidade.

**Listagem 13.** Entidade Pessoa

**package** br.com.easyjavamagazine.entidades;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

**import** javax.persistence.Inheritance;

**import** javax.persistence.InheritanceType;

@Entity

@Inheritance(strategy=InheritanceType.***JOINED***)

**public** **class** Pessoa {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

**private** Integer id;

**private** String nome;

**private** String email;

**private** String sexo;

**private** String cpf;

//getters e setters

}

**Listagem 14.** Entidade FuncionarioBiblioteca

**package** br.com.easyjavamagazine.entidades;

**import** java.util.ArrayList;

**import** java.util.List;

**import** javax.persistence.CascadeType;

**import** javax.persistence.Entity;

**import** javax.persistence.JoinColumn;

**import** javax.persistence.OneToMany;

**import** javax.persistence.OneToOne;

@Entity

**public** **class** FuncionarioBiblioteca **extends** Pessoa{

@OneToOne(cascade={CascadeType.***ALL***})

@JoinColumn(name="id\_usuario")

Usuario usuario;

@OneToMany(cascade={CascadeType.***ALL***})

@JoinColumn(name="id\_funcionarioBiblioteca")

**private** List<Telefone> telefones = **new** ArrayList<Telefone>();

@OneToOne(cascade={CascadeType.***ALL***})

@JoinColumn(name="id\_endereco")

**private** Endereco endereco;

//getters e setters

}

**Listagem 15.** Entidade Leitor

**package** br.com.easyjavamagazine.entidades;

**import** java.util.ArrayList;

**import** java.util.List;

**import** javax.persistence.CascadeType;

**import** javax.persistence.Entity;

**import** javax.persistence.JoinColumn;

**import** javax.persistence.OneToMany;

**import** javax.persistence.OneToOne;

@Entity

**public** **class** Leitor **extends** Pessoa{

**private** String turma;

**private** String matricula;

@OneToMany(cascade={CascadeType.***ALL***})

@JoinColumn(name="id\_leitor")

**private** List<Telefone> telefones = **new** ArrayList<Telefone>();

@OneToOne(cascade={CascadeType.***ALL***})

@JoinColumn(name="id\_endereco")

**private** Endereco endereco;

//getters e setters

}

No Java é comum usarmos herança e a JPA provê maneiras de fazermos o mapeamento das nossas hierarquias de classes. Em nosso sistema teremos uma classe Pessoa que terá duas subclasses: FuncionarioBiblioteca e Leitor. As **Listagens 13, 14 e 15** apresentam o código dessas classes. Essas três classes serão entidades e cada uma será representada por sua respectiva tabela no banco de dados. Para atingirmos esse resultado usaremos a anotação @Inheritance com a stratégia JOINED. Repare que definimos uma chave primária(usando a anotação @Id) apenas na entidade Pessoa, mas na realidade o JPA provider irá criar as três tabelas(pessoa, funcionariobiblioteca e leitor) com chave primária(representada pela coluna id de cada tabela nesse caso). O JPA provider irá usar o valor dessas chaves primárias para relacionar um registro da tabela pessoa com um registro da tabela funcionariobiblioteca ou um registro da tabela pessoa com um registro da tabela leitor.

Como no persistence.xml configuramos a propriedade hibernate.hbm2ddl.auto com o valor create, se rodarmos o nosso projeto no Jboss Wildfly todas as tabelas serão criadas automaticamente pois nesse momento já temos todas entidades prontas com seus respectivos mapeamentos.

# Populando Banco de Dados

Como foi mencionado anteriormente não iremos exibir o código de todos cadastros, nem de todos arquivos que fazem parte do DAO, pois teríamos listagens muito parecidas devido a similaridade entre os códigos. Se o leitor quiser criar o projeto e seus arquivos a partir das instruções contidas no artigo e executá-lo no servidor de aplicação, será necessário popular o banco de dados usando instruções SQL, pois omitimos os cadastros e não haveria outra forma de cadastrar as informações necessárias ao funcionamento do sistema.

Então após subir o servidor de aplicação e as tabelas serem criadas os comandos SQL da **Listagem 16** devem ser executados via “MySQL Command-Line Tool”.

**Listagem 16.** Instruções SQL para popular a base de dados

INSERT INTO endereco(id, bairro, cep, cidade, complemento, endereco, estado) VALUES(1, 'Pq Sao Rafael', '08311080', 'Sao Paulo', '', 'Avenida Baronesa de Muritiba, num 51', 'SP');

INSERT INTO endereco(id, bairro, cep, cidade, complemento, endereco, estado) VALUES(2, 'Santa Terezinha', '02460000', 'Sao Paulo', '', 'Rua Luís Antonio dos Santos, num 110', 'SP');

INSERT INTO pessoa(id, cpf, email, nome, sexo) VALUES(1, '34087772486', 'iris@meudominio.com', 'Iris', 'F');

INSERT INTO leitor(matricula, turma, id, id\_endereco) VALUES('35012', 'TADS01', 1, 1);

INSERT INTO autor(id, nome) VALUES(1, 'Ricardo R. Lecheta');

INSERT INTO categoria(id, nome) VALUES(1, 'Informatica');

INSERT INTO editora(id, email, nome, site, id\_endereco) VALUES(1, 'novatec@novatec.com.br', 'Novatec', 'www.novatec.com.br', 2);

INSERT INTO livro(id, isbn, nome, id\_categoria, id\_editora) VALUES(1, '9788575222928', 'Google Android para Tablets', 1, 1);

INSERT INTO livro(id, isbn, nome, id\_categoria, id\_editora) VALUES(2, '9788575224014', 'Desenvolvendo para iPhone e iPad', 1, 1);

INSERT INTO livro\_autor(id\_livro, id\_autor) VALUES(1, 1);

INSERT INTO telefone(id, fixo, numero, id\_leitor) VALUES(1, b'0', '99999-9999', 1);

INSERT INTO telefone(id, fixo, numero, id\_leitor) VALUES(2, b'1', '5555-5555', 1);

INSERT INTO telefone(id, fixo, numero, id\_editora) VALUES(3, b'0', '98888-8888', 1);

INSERT INTO telefone(id, fixo, numero, id\_editora) VALUES(4, b'1', '7777-7777', 1);

Depois das tabelas terem sido criadas e populadas com informações, não queremos que da próxima vez que o servidor de aplicação subir elas sejam recriadas e todas informações apagadas. Então devemos alterar a propriedade hibernate.hbm2ddl.auto no persistence.xml colocando o valor validate. Após essa alteração devemos clicar com o botão direito no Jboss que está na aba Servers do Eclipse e escolher a opção “Clean...”. Feito isso não corremos mais o risco das tabelas serem recriadas, da próximo vez que o Jboss iniciar o Hibernate irá só realizar as validações necessárias.

# Criando classe de conversão para LocalDate

**Listagem 17.** Classe de conversão para usar com LocalDate

**package** br.com.easyjavamagazine.conversores;

**import** java.sql.Date;

**import** java.time.LocalDate;

**import** javax.persistence.AttributeConverter;

**import** javax.persistence.Converter;

@Converter(autoApply = **true**)

**public** **class** LocalDateConversor **implements** AttributeConverter<LocalDate, Date> {

@Override

**public** Date convertToDatabaseColumn(LocalDate dataDaEntidade) {

**return** (dataDaEntidade == **null**) ? **null** : Date.*valueOf*(dataDaEntidade);

}

@Override

**public** LocalDate convertToEntityAttribute(Date dataDoBancoDeDados) {

**return** (dataDoBancoDeDados == **null**) ? **null** : dataDoBancoDeDados.toLocalDate();

}

}

Como a entidade Emprestimo(mostrada na **Listagem 10**) tem atributos do tipo LocalDate e na JPA não temos uma anotação que mapeie um atributo desse tipo, usaremos um outro recurso da JPA para fazer as conversões, que consiste em criar uma classe que herde de AttributeConverter e que sobrescreva seus dois métodos. O código dessa classe é mostrado na **Listagem 17**. Levando em conta que o tipo java.sql.Date é compatível com o tipo date das colunas de nosso banco de dados, podemos converter de java.time.LocalDate para java.sql.Date no hora de gravar os dados na base, e fazer o aposto na hora de recuperar os dados do banco e atribuí-los aos atributos de nossas entidades. Os métodos convertToDatabaseColumn e convertToEntityAttribute são responsáveis por essas conversões. Também usamos a anotação @Converter em nossa classe setando o autoApply para true, para que esse tipo de conversão seja feito automaticamente para todos atributos do tipo LocalDate de todas nossas entidades.

# Criando Interceptor e classes DAO

Agora que nossas entidades foram implementadas teremos que implementar as classes que irão compor o nosso DAO. Para facilitar a nossa vida ao implementarmos as classes DAO e as demais classes do projeto, iremos usar injeção de dependências com o Weld, que é a implementação de referência da especificação CDI. Para habilitarmos a injeção de dependências com CDI no projeto basta criarmos um arquivo vazio chamado beans.xml na pasta WEB-INF.

Uma coisa comum de se encontrar em classes DAO são códigos para gerenciamento de transações(ver **Box 2**). Quando isso acontece temos grandes quantidades de código de infraestrutura dentro de nossas classes DAO, sem contar que é código duplicado. Então, antes de começarmos a criar nossas classes DAO vamos criar um mecanismo para gerenciar as transações de nosso sistema, e assim evitar que esse tipo de código fique espalhado dentro do nosso DAO ou espalhado pela aplicação toda. A solução que iremos adotar é baseada em Interceptors e Interceptor bindings(assuntos abordados na especificação de Interceptors e na especificação CDI).

|  |
| --- |
| **Box 2.** Transação |
| Transação pode ser definida como uma série de ações que devem ser executadas uma após a outra até que todas elas tenham sido concluídas com sucesso. Caso alguma dessas ações não possa ser concluída com sucesso, as anteriores devem ser desfeitas, voltando o estado do sistema igual ao que era antes da transação iniciar. Uma transação deve terminar ou com um commit ou com um rollback. |

Com um Interceptor conseguimos interceptar uma chamada de método. A capacidade de interceptar métodos é uma característica encontrada na programação orientada a aspecto(em inglês Aspect Orientated Programing, ou simplesmente AOP). Embora a especificação de Interceptors e a CDI tenham especificado apenas essa pequena parte da AOP, não devemos tirar o mérito dessas especificações, pois essa é a parte da AOP que a maioria dos desenvolvedores usam.

**Listagem 18.** Interceptor que irá gerenciar as transações

**package** br.com.easyjavamagazine.interceptadores;

**import** java.io.Serializable;

**import** javax.inject.Inject;

**import** javax.interceptor.AroundInvoke;

**import** javax.interceptor.Interceptor;

**import** javax.interceptor.InvocationContext;

**import** javax.persistence.EntityManager;

**import** br.com.easyjavamagazine.util.MensagemUtil;

@Interceptor @Transacional

**public** **class** TransacionalInterceptor **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

@Inject

**private** EntityManager entityManager;

@AroundInvoke

**public** Object intercept(InvocationContext context){

Object resultado = **null**;

**try** {

entityManager.getTransaction().begin();

resultado = context.proceed();

entityManager.getTransaction().commit();

} **catch** (Exception e) {

entityManager.getTransaction().rollback();

MensagemUtil.*addMensagemDeErro*("Erro - ",

"Detalhes do erro: " + e.getClass().getName() + " - " + e.getMessage());

e.printStackTrace();

}

**return** resultado;

}

}

A **listagem 18** mostra o código do Interceptor que irá gerenciar as transações da aplicação. Veja que é uma classe comum que leva a anotação @Interceptor. Dentro do Interceptor criado temos um método chamado intercept que tem uma anotação @AroundInvoke e por conta disso será responsável por interceptar as chamadas aos nossos métodos de negócios que precisarem de uma transação. Dentro desse método usamos o entityManager que foi injetado via CDI para iniciarmos uma transação, logo em seguida invocamos o método proceed do contexto para indicar que nesse momento queremos que seja executada a lógica contida no método que estamos interceptando, e por último damos commit na transação. Caso ocorra uma exception é executado rollback e adicionada uma mensagem que será mostrada ao usuário.

**Listagem 19.** Interceptor binding para associar o Interceptor com os beans

**package** br.com.easyjavamagazine.interceptadores;

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

**import** javax.interceptor.InterceptorBinding;

@InterceptorBinding

@Target({ElementType.***TYPE***, ElementType.***METHOD***})

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** Transacional {

}

Agora que já criamos o Interceptor temos que achar um meio de dizer para ele quais métodos precisam de uma transação e devem ser interceptados. Fazemos isso por meio de um Interceptor binding, que é responsável por associar o Interceptor com os métodos que precisam ser interceptados. A **Listagem 19** mostra o Interceptor binding chamado Transacional que foi criado, que nada mais é do que uma anotação marcada com outra anotação chamada @InterceptorBinding. Na **Listagem 18** repare também que marcamos a classe TransacionalInterceptor com a anotação @Transacional que representa nosso Interceptor binding. A partir de agora sempre que um método precisar ser executado dentro de uma transação podemos colocar @Transacional em cima dele.

**Listagem 20.** Arquivo beans.xml

<?xml version=*"1.0"* encoding=*"ISO-8859-1"* ?>

<beans xmlns=*"http://xmlns.jcp.org/xml/ns/javaee"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://xmlns.jcp.org/xml/ns/javaee*

*http://xmlns.jcp.org/xml/ns/javaee/beans\_1\_1.xsd"*

bean-discovery-mode=*"all"*>

<interceptors>

<class>br.com.easyjavamagazine.interceptadores.TransacionalInterceptor</class>

</interceptors>

</beans>

Para que nosso Interceptor funcione precisamos registra-lo no arquivo beans.xml criado dentro de WEB-INF anteriormente. A **Listagem 20** mostra como deve ficar o beans.xml.

**Listagem 21.** Fábrica de Entity Manager

**package** br.com.easyjavamagazine.fabricas;

**import** javax.enterprise.context.ApplicationScoped;

**import** javax.enterprise.context.RequestScoped;

**import** javax.enterprise.inject.Disposes;

**import** javax.enterprise.inject.Produces;

**import** javax.persistence.EntityManager;

**import** javax.persistence.EntityManagerFactory;

**import** javax.persistence.Persistence;

@ApplicationScoped

**public** **class** FabricaDeEntityManager {

**private** **static** EntityManagerFactory *factory* = Persistence.*createEntityManagerFactory*("bibliotecaPersistence");

@Produces @RequestScoped

**public** EntityManager criarEntityManager() {

**return** *factory*.createEntityManager();

}

**public** **void** fecharEntityManager(@Disposes EntityManager manager){

manager.close();

}

}

Através da anotação @Inject injetamos um Entity Manager na classe TransacionalInterceptor criada anteriormente. Pelo fato da criação de um Entity Manager ser um pouco mais complexa do que o normal(não basta dar usar new para criar a instância, é preciso instanciar a classe a partir de uma EntityManagerFactory), precisamos “ensinar” para o CDI provider como criar um Entity Manager para poder injeta-lo quando precisarmos. **A Listagem 21** mostra a classe que ficará encarregada de produzir os Entity Managers. A anotação @Produces indica o método produtor de Entity Manager e a anotação @RequestScoped indica que deve ser produzido um Entity Manager por requisição. No final de cada requisição o Entity Manager que foi criado será fechado através do método fecharEntityManager que receberá o Entity Manager via parâmentro, mas para que isso funcione é necessário usar a anotação @Disposes conforme mostrado na listagem.

**Listagem 22.** Classe MensagemUtil que irá adicionar mensagens para o usuário visualizar

**package** br.com.easyjavamagazine.util;

**import** javax.faces.application.FacesMessage;

**import** javax.faces.context.FacesContext;

**public** **class** MensagemUtil {

**public** **static** **void** addMensagemDeErro(String mensagemDeErro, String detalhes){

*addMensagem*(FacesMessage.***SEVERITY\_ERROR***, mensagemDeErro, detalhes);

}

**public** **static** **void** addMensagemDeAviso(String mensagemDeAviso, String detalhes){

*addMensagem*(FacesMessage.***SEVERITY\_WARN***, mensagemDeAviso, detalhes);

}

**public** **static** **void** addMensagemInformativa(String mensagemInformativa, String detalhes){

*addMensagem*(FacesMessage.***SEVERITY\_INFO***, mensagemInformativa, detalhes);

}

**private** **static** **void** addMensagem(FacesMessage.Severity severidade, String mensagem, String detalhes){

FacesMessage msg = **new** FacesMessage(severidade, mensagem, detalhes);

FacesContext.*getCurrentInstance*().addMessage(**null**, msg);

}

}

A classe TransacionalInterceptor também faz uso do método addMensagemDeErro presente na classe MensagemUtil. A **Listagem 22** mostra o código da classe MensagemUtil que é bastante simples e não requer muitas explicações. A classe disponibiliza alguns métodos públicos que delegam todo o trabalho para o método privado addMensagem que por sua vez usa um método de FacesContext para adicionar uma mensagem que será exibida para o usuário.

**Listagem 23.** implementação do DAO genérico

**package** br.com.easyjavamagazine.daoimpl;

**import** java.util.List;

**import** javax.persistence.EntityManager;

**import** br.com.easyjavamagazine.dao.GenericoDao;

**public** **class** GenericoDaoImpl<T, K> **implements** GenericoDao<T, K> {

**private** Class<T> classeDaEntidade;

**private** EntityManager entityManager;

**public** GenericoDaoImpl(Class classeDaEntidade, EntityManager entityManger) {

**this**.classeDaEntidade = classeDaEntidade;

**this**.entityManager = entityManger;

}

@Override

**public** **void** inserir(T entidade) {

entityManager.persist(entidade);

}

@Override

**public** **void** remover(T entidade) {

entityManager.remove(entityManager.merge(entidade));

}

@Override

**public** **void** atualizar(T entidade) {

entityManager.merge(entidade);

}

@Override

**public** T pesquisarPorID(K id) {

**return** entityManager.find(classeDaEntidade, id);

}

@Override

**public** List<T> listarTodos(){

**return** entityManager.createQuery("select t from " + classeDaEntidade.getSimpleName() + " t").getResultList();

}

}

O nosso próximo passo será criarmos as classes DAO. Para cada classe DAO teremos sua respectiva interface. Dessa forma na hora de injetar qualquer DAO que precisarmos, poderemos usar uma variável do tipo da interface com a anotação @Inject em cima dela, e assim diminuímos o acoplamento pois não estamos declarando a implementação concreta que deve ser usada. Além disso quando definimos uma interface podemos ter mais de uma implementação para ela, no caso de um DAO podemos ter por exemplo uma implementação que use JPA e outra que use JDBC, e usando recursos da CDI podemos escolher qual implementação deve ser injetada(usando Qualifiers, Alternatives ou a anotação @Vetoed conseguimos configurar isso). Para simplificar o artigo os códigos das interfaces serão omitidos, mas de qualquer forma é fácil saber os métodos declarados em cada interface, basta observar na implementação concreta quais métodos foram sobrescritos(os que estão marcados com a anotação @Override).

Para que tenhamos reutilização de código iremos usar um DAO genérico conforme mostrado na **Listagem 23**. É comum que desenvolvedores criem primeiro o DAO genérico e depois criem as demais classes DAO fazendo-as herdar do DAO genérico. Iremos adotar uma estratégia diferente usando composição ao invés de herança. As nossas outras classes DAO terão um DAO genérico ao invés de serem um DAO genérico. Como nossas classes terão um DAO genérico, elas irão delegar as tarefas para ele. Com essa estratégia diminuimos o acoplamento. Além disso não herdaremos os comportamentos do DAO genérico e iremos implementar somente os métodos que fazem sentido em cada classe DAO. Vamos imaginar uma situação hipotética apenas para ficar mais claro o problema de herdar um comportamento nesse caso. Imagine que precisamos gravar no banco de dados alguns logs de atividades realizadas dentro do sistema e para isso poderíamos ter uma classe LogDao que herdaria de GenericoDao e ganharia de bandeja todos os comportamentos do DAO genérico, inclusive o comportamento do método remover. Mas imagine que um dos requisitos do sistema é que esses logs não sejam deletados da base de dados. Nessa situação poderíamos até sobrescrever o método remover e lançar uma Exception caso alguém tentasse usá-lo, mas se os logs não devem ser apagados faz sentido termos um método remover na classe LogDao? Não faz sentido. Usando composição ao invés de herança não teremos esse problema.

**Listagem 24.** implementação do empréstimo DAO

**package** br.com.easyjavamagazine.daoimpl;

**import** java.io.Serializable;

**import** javax.inject.Inject;

**import** javax.persistence.EntityManager;

**import** br.com.easyjavamagazine.dao.EmprestimoDao;

**import** br.com.easyjavamagazine.dao.GenericoDao;

**import** br.com.easyjavamagazine.entidades.Emprestimo;

**public** **class** EmprestimoDaoImpl **implements** EmprestimoDao, Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** GenericoDao<Emprestimo, Integer> genericoDao;

@Inject

EmprestimoDaoImpl(EntityManager entityManager){

genericoDao = **new** GenericoDaoImpl<Emprestimo, Integer>(Emprestimo.**class**, entityManager);

}

@Override

**public** **void** inserir(Emprestimo emprestimo) {

genericoDao.inserir(emprestimo);

}

@Override

**public** **void** atualizar(Emprestimo emprestimo) {

genericoDao.atualizar(emprestimo);

}

}

A **Listagem 24** exibe a classe EmprestimoDaoImpl que implementa a interface EmprestimoDao, ou seja, é a implementação do DAO de empréstimo. Nessa classe usamos injeção de dependência via construtor. Como colocamos a anotação @Inject no construtor, o CDI provider irá instanciar a classe EmprestimoDaoImpl quando necessário passando um Entity Manager como argumento. Lembrando que o Entity Manager será produzido pela fábrica que criamos anteriormente que usa @Produces. O Entity Manager injetado pelo construtor é passado para o DAO genérico. Os métodos dessa nossa classe DAO apenas delegam o trabalho para os métodos do DAO genérico.

**Listagem 25.** implementação do leitor DAO

**package** br.com.easyjavamagazine.daoimpl;

**import** java.io.Serializable;

**import** java.util.List;

**import** javax.inject.Inject;

**import** javax.persistence.EntityManager;

**import** br.com.easyjavamagazine.dao.GenericoDao;

**import** br.com.easyjavamagazine.dao.LeitorDao;

**import** br.com.easyjavamagazine.entidades.Leitor;

**public** **class** LeitorDaoImpl **implements** LeitorDao, Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** GenericoDao<Leitor, Integer> genericoDao;

@Inject

LeitorDaoImpl(EntityManager entityManager){

genericoDao = **new** GenericoDaoImpl<Leitor, Integer>(Leitor.**class**, entityManager);

}

@Override

**public** Leitor pesquisarPorId(Integer id) {

**return** genericoDao.pesquisarPorID(id);

}

@Override

**public** List<Leitor> listarLeitores() {

**return** genericoDao.listarTodos();

}

//restante dos métodos omitidos

}

O código mostrado na **Listagem 25** se refere a implementação do leitor DAO. A implementação dessa classe é muito semelhante a implementação do empréstimo DAO visto anteriormente e por isso dispensa comentários.

**Listagem 26.** implementação do funcionário biblioteca DAO

**package** br.com.easyjavamagazine.daoimpl;

**import** java.io.Serializable;

**import** java.util.List;

**import** javax.inject.Inject;

**import** javax.persistence.EntityManager;

**import** br.com.easyjavamagazine.dao.GenericoDao;

**import** br.com.easyjavamagazine.dao.FuncionarioBibliotecaDao;

**import** br.com.easyjavamagazine.entidades.FuncionarioBiblioteca;

**public** **class** FuncionarioBibliotecaDaoImpl **implements** FuncionarioBibliotecaDao, Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** GenericoDao<FuncionarioBiblioteca, Integer> genericoDao;

@Inject

FuncionarioBibliotecaDaoImpl(EntityManager entityManager){

genericoDao = **new** GenericoDaoImpl<FuncionarioBiblioteca, Integer>(FuncionarioBiblioteca.**class**, entityManager);

}

@Override

**public** **void** inserir(FuncionarioBiblioteca funcionarioBiblioteca) {

genericoDao.inserir(funcionarioBiblioteca);

}

@Override

**public** **void** remover(FuncionarioBiblioteca funcionarioBiblioteca) {

genericoDao.remover(funcionarioBiblioteca);

}

@Override

**public** **void** atualizar(FuncionarioBiblioteca funcionarioBiblioteca) {

genericoDao.atualizar(funcionarioBiblioteca);

}

@Override

**public** List<FuncionarioBiblioteca> listarFuncionariosBiblioteca() {

**return** genericoDao.listarTodos();

}

@Override

**public** **boolean** existeFuncionarioCadastrado() {

List<FuncionarioBiblioteca> listaDeFuncionarios = listarFuncionariosBiblioteca();

**if**(listaDeFuncionarios != **null** && listaDeFuncionarios.size() > 0){

**return** **true**;

}

**return** **false**;

}

}

A **Listagem 26** apresenta o funcionário biblioteca DAO, que é muito semelhante as classes DAO vistas nas listagens anteriores. A maioria dos métodos dessa classe apenas invocam algum outro método no DAO genérico, com exceção do método existeFuncionarioCadastrado que contém uma lógica que verifica se existe algum funcionário cadastrado no banco de dados, e em caso afirmativo retorna true, caso contrário false. Mais adiante veremos o código da classe FuncionarioEUsuarioVerificadorBean onde esse método será usado.

**Listagem 27.** implementação do usuário DAO

**package** br.com.easyjavamagazine.daoimpl;

**import** java.io.Serializable;

**import** java.util.List;

**import** javax.inject.Inject;

**import** javax.persistence.EntityManager;

**import** javax.persistence.NoResultException;

**import** javax.persistence.Query;

**import** br.com.easyjavamagazine.dao.GenericoDao;

**import** br.com.easyjavamagazine.dao.UsuarioDao;

**import** br.com.easyjavamagazine.entidades.Usuario;

**public** **class** UsuarioDaoImpl **implements** UsuarioDao, Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** GenericoDao<Usuario, Integer> genericoDao;

**private** EntityManager entityManager;

@Inject

UsuarioDaoImpl(EntityManager entityManager){

genericoDao = **new** GenericoDaoImpl<Usuario, Integer>(Usuario.**class**, entityManager);

**this**.entityManager = entityManager;

}

@Override

**public** List<Usuario> listarUsuarios() {

**return** genericoDao.listarTodos();

}

@Override

**public** **boolean** existeUsuarioCadastrado() {

List<Usuario> listaDeUsuarios = listarUsuarios();

**if**(listaDeUsuarios != **null** && listaDeUsuarios.size() > 0){

**return** **true**;

}

**return** **false**;

}

@Override

**public** Usuario pesquisarUsuario(Usuario usuario) {

Usuario usuarioEncontrado = **null**;

Query query = entityManager.createQuery("select u from Usuario u where u.usuario = :usuarioParam and u.senha = :senhaParam")

.setParameter("usuarioParam", usuario.getUsuario())

.setParameter("senhaParam", usuario.getSenha());

**try**{

usuarioEncontrado = (Usuario)query.getSingleResult();

}**catch**(NoResultException e){

// não faz nada, se não achar o usuário deixa retornar null

}

**return** usuarioEncontrado;

}

}

Na **Listagem 27** temos a implementação do usuário DAO. Nessa classe temos três métodos. O método listarUsuarios retorna uma lista de objetos que são do tipo Usuario, e essa lista é preenchida com o conteúdo presente na tabela usuario do banco de dados. O método existeUsuarioCadastrado verifica se existe algum usuário cadastrado na base de dados retornando true se já existir algum ou false se não existir nenhum. Esse método será usado pela classe FuncionarioEUsuarioVerificadorBean como veremos adiante. Por último temos o método pesquisarUsuario que será usado para validar se o usuário e senha digitados na tela de login são válidos. A lógica usada é bastante simples: é passada uma String JPQL para o método createQuery do Entity Manager que retorna um objeto do tipo Query. Esse objeto do tipo Query, criado a partir da JPQL informada, será usado para que seja feita uma consulta no banco de dados em busca de um registro que seja compatível com o usuário e senha digitados na tela de login. Se algum registro for encontrado será retornado um objeto do tipo Usuario preenchido com os dados desse registro, caso contrário será retornado null.

**Listagem 28.** implementação do livro DAO

**package** br.com.easyjavamagazine.daoimpl;

**import** java.io.Serializable;

**import** java.util.ArrayList;

**import** java.util.List;

**import** javax.inject.Inject;

**import** javax.persistence.EntityManager;

**import** br.com.easyjavamagazine.dao.GenericoDao;

**import** br.com.easyjavamagazine.dao.LivroDao;

**import** br.com.easyjavamagazine.entidades.Emprestimo;

**import** br.com.easyjavamagazine.entidades.Livro;

**public** **class** LivroDaoImpl **implements** LivroDao, Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** GenericoDao<Livro, Integer> genericoDao;

@Inject

LivroDaoImpl(EntityManager entityManager){

genericoDao = **new** GenericoDaoImpl<Livro, Integer>(Livro.**class**, entityManager);

}

@Override

**public** Livro pesquisarPorId(Integer id) {

**return** genericoDao.pesquisarPorID(id);

}

@Override

**public** List<Livro> listarLivros() {

**return** genericoDao.listarTodos();

}

@Override

**public** List<Livro> listarLivrosDisponiveisParaEmprestimo() {

List<Livro> todosLivros = listarLivros();

List<Livro> livrosDisponiveis = **new** ArrayList<Livro>();

**for**(Livro livro : todosLivros){

List<Emprestimo> emprestimos = livro.getEmprestimos();

**boolean** todosEmprestimosTemDataDevolucao = **true**;

**for**(Emprestimo emprestimo : emprestimos){

**if**(emprestimo.getDataDevolucao() == **null**)

todosEmprestimosTemDataDevolucao = **false**;

}

**if**(todosEmprestimosTemDataDevolucao)

livrosDisponiveis.add(livro);

}

**return** livrosDisponiveis;

}

@Override

**public** List<Livro> listarLivrosEmprestados() {

List<Livro> todosLivros = listarLivros();

List<Livro> livrosEmprestados = **new** ArrayList<Livro>();

**for**(Livro livro : todosLivros){

List<Emprestimo> emprestimos = livro.getEmprestimos();

**for**(Emprestimo emprestimo : emprestimos){

**if**(emprestimo.getDataDevolucao() == **null**)

livrosEmprestados.add(livro);

}

}

**return** livrosEmprestados;

}

//restante dos métodos omitidos

}

A **Listagem 28** exibe o livro DAO. Nessa listagem temos quatro métodos: pesquisarPorId, listarLivros, listarLivrosDisponiveisParaEmprestimo e listarLivrosEmprestados. O pesquisarPorId irá delegar para o DAO genérico a tarefa de pesquisar um livro pelo id e o listarLivros irá delegar para o DAO genérico a tarefa de recuperar uma lista com todos os livros. Os outros dois métodos estão relacionados ao empréstimo e devolução de livros. Na tela de empréstimos eu só devo conseguir visualizar os livros disponíveis para empréstimo, e na tela de devolução só devo conseguir visualizar os livros que estão emprestados. Essa é justamente a funções desses dois métodos, um retorna os livros disponíveis e o outro retorna os livros que estão emprestados.

Um livro pode estar relacionado a muitos empréstimos e cada um desses empréstimos tem uma data de devolução, que é realmente a data efetiva em que a devolução do livro foi feita. Enquanto um livro não é devolvido a data de devolução permanece nula. Então ambos os métodos listarLivrosDisponiveisParaEmprestimo e listarLivrosEmprestados recuperam uma lista com todos os livros e iteram nessa lista. A cada passada dentro do loop for são recuperados os empréstimos de um livro específico e verificado se os empréstimos possuem data de devolução. Se todos empréstimos vinculados aquele livro específico possuírem data de devolução significa que o livro está disponível e pode ser emprestado, mas se por ventura algum dos empréstimos tiver a data de devolução nula significa que ele já se encontra emprestado.

# Criando as páginas JSF e os Managed Beans

Agora que já temos as entidades e as classes DAO prontas iremos iniciar o desenvolvimento das páginas JSF e dos managed beans.

Nossas páginas irão utilizar imagens e um arquivo .css para formatação. Dentro da pasta WebContent do projeto deverá ser criada uma pasta chamada resources que será usada para armazenar essas imagens e o css(esses arquivos podem ser baixados do projeto de exemplo que está disponível no Github). O conteúdo do arquivo css não será exibido em nenhuma listagem por não ser esse o foco do artigo.

Templates

Uma boa prática que devemos adotar durante o desenvolvimento de um sistema é evitar a duplicidade de código e isso vale para nossas páginas. Uma maneira de evitarmos código duplicado em nossas páginas é a criação de arquivos de template com a biblioteca de tags Facelets. Um arquivo de template deve agrupar os códigos que são comuns a várias páginas. No caso da nossa aplicação, os templates definirão o cabeçalho e rodapé, pois são coisas comuns a todas as páginas. Fazendo uma analogia, imagine um template como se fosse uma folha de papel com buracos, e esses buracos são locais onde podemos encaixar diversos conteúdos diferentes. Cada cliente do template(template client), no caso a página que usa o template, poderia inserir um conteúdo diferente em cada um desses “buracos”.

Criaremos dentro de WEB-INF dois arquivos de template, um chamado \_template\_simples.xhtml e outro chamado \_template.xhtml. Mantê-los dentro de WEB-INF é uma forma de evitar que sejam acessados diretamente pelo navegador através de uma URL.

**Listagem 29.** \_template\_simples.xhtml

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!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:f=*"http://java.sun.com/jsf/core"*

xmlns:h=*"http://java.sun.com/jsf/html"*

xmlns:ui=*"http://java.sun.com/jsf/facelets"*>

<h:head>

<title>Easy - Sistema Gerenciador de Biblioteca</title>

<h:outputStylesheet library=*"css"* name=*"style.css"*/>

</h:head>

<h:body>

<div id=*"cabecalho"*>

<h:graphicImage library=*"imagens"* name=*"logo-easy.png"* id=*"logoCompany"*/>

</div>

<div id=*"usuarioLogado"*>

<h:form rendered=*"#{usuarioLogadoBean.logado}"*>

Logado como: #{usuarioLogadoBean.usuario.usuario}

<h:commandButton value=*"[Sair]"* action=*"#{loginBean.efetuarLogout}"* />

</h:form>

</div>

<div id=*"conteudo"*>

<ui:insert name=*"corpo"* />

</div>

<div id=*"rodape"*>

Copyright 2014.

Todos os Direitos reservados a Easy Java Magazine

</div>

</h:body>

</html>

A **Listagem 29** mostra o conteúdo da página \_template\_simples.xhtml. Para podermos usar as tags da tag library Facelets precisamos declarar o namespace xmlns:ui="http://java.sun.com/jsf/facelets. Nesse código estamos usando a tag ui:insert(que é uma tag da biblioteca Facelets e indicamos isso colocando o mesmo prefixo ui que usamos na declaração do namespace), e ela é usada para demarcar as partes do template que podem ser substituídas pelas páginas clientes que irão usar o template. O corpo dessa nossa página possui 4 divs, a primeira representa o cabeçalho e contém uma imagem, a segunda mostra o usuário logado e tem um botão para fazer logout, a terceira é reservada para comportar o conteúdo principal da página(é a que contém a tag ui:insert) e a quarta e última div representa o rodapé. Na div com id usuarioLogado o managed bean usuarioLogadoBean(veremos seu código adiante) é usado para exibir o nome do usuário logado caso algum já tenha se autenticado.

A outra página de template - template.xhtml - que veremos a seguir tem basicamente a mesma estrutura, contendo apenas algumas tags a mais dentro da div do conteúdo. Portanto como todas as outras páginas serão clientes ou da página \_template\_simples.xhtml ou da página \_template.xhtml, terão por consequência essa estrutura também e irão apenas substituir o conteúdo principal.

**Listagem 30.** \_template.xhtml

<!-- restanto do código ... -->

<div id=*"conteudo"*>

<br/>

<h:form>

<h:messages globalOnly=*"true"* showDetail=*"true"* />

</h:form>

<br/>

<ui:insert name=*"corpo"* />

<h:form>

<h:commandLink value=*"Voltar ao Menu"* action=*"menu\_principal?faces-redirect=true"* />

</h:form>

</div>

<!-- restanto do código ... -->

Como foi mencionado anteriormente o conteúdo do arquivo \_template.xhtml é quase igual ao do arquivo \_template\_simples.xhtml, portanto na **Listagem 30** é mostrada somente a parte diferente entre os dois para evitar listagens com código duplicado. O que muda entre os dois arquivos é o que tem dentro da div conteudo. Dentro dessa div no arquivo \_template.xhtml usamos a tag h:messages para mostrar ao usuário as mensagens que forem adicionadas através do método addMessage da classe FacesContext. Logo abaixo é usada a tag ui:insert que já foi explicada. Por último é criado um link através da tag h:commandLink que irá redirecionar para o menu principal, que é apenas uma página que tem links para todas as demais. A tag h:commandLink precisa estar dentro da tag h:form para funcionar.

Autenticação

Antes de criarmos a página de login e o managed bean associado a ela para que o usuário possa se autenticar, precisaremos criar antes outros managed beans que irão apoiar esse processo de autenticação.

**Listagem 31.** Managed Bean UsuarioLogadoBean

**package** br.com.easyjavamagazine.mb;

**import** java.io.Serializable;

**import** javax.enterprise.context.SessionScoped;

**import** javax.inject.Named;

**import** br.com.easyjavamagazine.entidades.Usuario;

@Named

@SessionScoped

**public** **class** UsuarioLogadoBean **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** Usuario usuario;

**public** **void** logar(Usuario usuario){

**this**.usuario = usuario;

}

**public** **void** deslogar(){

**this**.usuario = **null**;

}

**public** **boolean** isLogado(){

**return** usuario != **null**;

}

**public** Usuario getUsuario(){

**return** usuario;

}

}

O código do primeiro managed bean que estamos vendo nesse artigo se encontra na **Listagem 31**, o UsuarioLogadoBean. Um managed bean pode ser definido como um objeto que é gerenciado pelo container. Se estivéssemos usando JSF sem CDI, poderíamos usar a anotação @ManagedBean(javax.faces.bean.ManagedBean) para registrar a classe como um managed bean do JSF. Porém a recomendação é que se use a anotação @Named no lugar de @ManagedBean, e nesse caso estaremos usando um managed bean gerenciado pelo container CDI e não pelo JSF. É importante ressaltar que a anotação @Named não torna a classe um bean, ela apenas possibilita referenciarmos esse bean dentro de uma EL presente em alguma página JSF. O container CDI trata qualquer classe que satisfaça as condições abaixo como um managed bean:

* Que não seja uma classe interna não estática(non-static inner class).
* Que seja uma classe concreta ou “anotada” com @Decorator.
* Que não esteja “anotada” com qualquer anotação que a defina como um EJB ou declarada como um EJB no arquivo ejb-jar.xml.
* Que não implemente javax.enterprise.inject.spi.Extension.
* Que tenha um construtor apropriado que pode ser um construtor default sem parâmetros ou um construtor marcado com @Inject.

A anotação @Named nos permite dar uma nome para nosso bean, pois ela aceita uma String como argumento, e esse é o nome que deve ser usado dentro de qualquer EL que queira usar esse bean. Por exemplo poderíamos colocar em cima da classe @Named(“usuarioBean”) e na EL usaríamos esse nome: #{usuarioBean.logado}. Também existe a opção de não passarmos nenhuma String para a anotação escrevendo apenas @Named e nesse caso será usado o nome padrão que é o nome não qualificado da classe com a primeira letra em minúscula. Para a classe UsuarioLogadoBean o nome padrão seria usuarioLogadoBean por exemplo.

Na classe UsuarioLogadoBean foram colocadas as anotações @Named – que já foi explicada – e @SessionScoped que indica que só existirá uma instância desse bean por sessão. Esse bean irá armazenar o objeto usuário que representa o usuário logado no sistema. É uma classe bem simples que contém apenas quatro métodos: o logar que seta o usuário logado na variável usuario, o deslogar que seta a variável usuario como nula, o isLogado que retorna um boolean para indicar se existe algum usuário logado e o getUsuario que retorna o usuário logado.

**Listagem 32.** Managed Bean FuncionarioEUsuarioVerificadorBean

**package** br.com.easyjavamagazine.mb;

**import** javax.enterprise.context.RequestScoped;

**import** javax.inject.Inject;

**import** javax.inject.Named;

**import** br.com.easyjavamagazine.dao.FuncionarioBibliotecaDao;

**import** br.com.easyjavamagazine.dao.UsuarioDao;

@Named

@RequestScoped

**public** **class** FuncionarioEUsuarioVerificadorBean {

@Inject

**private** FuncionarioBibliotecaDao funcionarioBibliotecaDao;

@Inject

**private** UsuarioDao usuarioDao;

**public** **boolean** existeFuncionarioEUsuarioCadastrado(){

**return** (funcionarioBibliotecaDao.existeFuncionarioCadastrado() && usuarioDao.existeUsuarioCadastrado());

}

}

O código do managed bean FuncionarioEUsuarioVerificadorBean é exibido na **Listagem 32**. A função desse bean é basicamente verificar se já foi cadastrado algum funcionário da biblioteca e algum usuário e isso é feito por meio do único método existente na classe chamado existeFuncionarioEUsuarioCadastrado. Dentro do método a verificação é feita através dos DAOs que foram injetados por meio da anotação @Inject.

**Listagem 33.** Página login.xhtml

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!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://java.sun.com/jsf/html"*

xmlns:f=*"http://java.sun.com/jsf/core"*

xmlns:ui=*"http://java.sun.com/jsf/facelets"*>

<ui:composition template=*"/WEB-INF/\_template\_simples.xhtml"*>

<ui:define name=*"corpo"*>

<h2>Login no sistema</h2>

<h:form>

<h:panelGrid columns=*"2"* styleClass=*"campos"*>

<h:outputLabel value=*"Login:"* for=*"login"* />

<h:inputText id=*"login"* value=*"#{loginBean.usuario.usuario}"*/>

<h:outputLabel value=*"Senha:"* for=*"senha"* />

<h:inputSecret id=*"senha"* value=*"#{loginBean.usuario.senha}"*/>

<h:commandButton value=*"Efetuar Login"* action=*"#{loginBean.efetuarLogin}"*/>

<h:commandButton value=*"Criar usuário administrador"* action=*"funcionario\_biblioteca?faces-redirect=true"*

rendered=*"#{loginBean.opcaoCadastroUsuarioHabilitada}"* />

</h:panelGrid>

</h:form>

</ui:define>

</ui:composition>

</html>

A **Listagem 33** exibe o código da página login.xhtml. Dentro dessa página informamos no atributo template da tag ui:composition qual o arquivo de template que será utilizado, ou seja, estamos declarando que a página login.xhtml é cliente do template \_template\_simples.xhtml. Também usamos a tag ui:define para definir o conteúdo que será substituído no template que precisa ter uma tag ui:insert correspondente. Nesse caso todo o conteúdo que estiver entre as tags <ui:define name="corpo"> e </ui:define> irão entrar no lugar da linha <ui:insert name="corpo" /> que foi colocada anteriormente quando criamos o template.

Quando usamos JSF podemos usar EL para associar o valor de um componente(por exemplo um h:inputText) a uma propriedade de um managed bean. Na página login.xhtml é feito esse tipo de associação, através da EL #{loginBean.usuario.usuario}, entre o valor do componente de texto onde será digitado o usuário(h:inputText com id login) e a propriedade usuario do objeto usuario que por sua vez é uma propriedade do managed bean LoginBean. Assim quando mudarmos o valor do h:inputText com id login a propriedade usuario do objeto usuario que está dentro de LoginBean será alterada também e vice versa. O mesmo conceito é aplicado para vincular o h:inputSecret com id senha com a propriedade senha através da EL #{loginBean.usuario.senha}".

Também podemos usar EL para referenciar métodos de managed beans dentro das tags JSF. Na página login.xhtml também é usado esse conceito, pois ao clicarmos no primeiro h:commandButton será invocado o método efetuarLogin do managed bean LoginBean como está configurado na EL #{loginBean.efetuarLogin}" colocada no atributo action do componente.

Na página de login ainda temos um outro h:commandButton que só será mostrado caso a propriedade opcaoCadastroUsuarioHabilitada do LoginBean seja true, e ela será true somente enquanto não for cadastrado nenhum funcionário da biblioteca e um respectivo usuário para ele. Para ficar mais claro o motivo desse botão existir vamos imaginar que acabamos de fazer o deploy desse sistema de biblioteca em produção, e portanto o banco de dados está vazio e ainda não foi cadastrado nada nele. Ao acessarmos a tela de login teremos que informar o nome de usuário e a senha, mas o que fazer agora se ainda não foi cadastrado nenhum usuário e senha? Tudo bem, poderíamos fazer um insert direto no banco, mas a solução que foi adotada é mais elegante. Resolvemos esse problema exibindo o botão "Criar usuário administrador" que conforme explicado anteriormente só irá aparecer se não existir nem funcionário nem usuário cadastrado, e quando clicado irá redirecionar para uma tela onde será possível cadastrar essas informações. Sempre o usuário precisará estar vinculado a um funcionário, por isso ambos são cadastrados na mesma tela. O primeiro usuário cadastrado dessa maneira será administrador e terá permissão para cadastrar os demais usuários que também poderão ser administradores ou não.

**Listagem 34.** Managed Bean LoginBean

**package** br.com.easyjavamagazine.mb;

**import** java.io.Serializable;

**import** javax.enterprise.context.RequestScoped;

**import** javax.inject.Inject;

**import** javax.inject.Named;

**import** br.com.easyjavamagazine.dao.UsuarioDao;

**import** br.com.easyjavamagazine.entidades.Usuario;

@Named

@RequestScoped

**public** **class** LoginBean **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

@Inject

**private** UsuarioLogadoBean usuarioLogado;

@Inject

**private** FuncionarioEUsuarioVerificadorBean verificadorBean;

**private** **boolean** opcaoCadastroUsuarioHabilitada;

@Inject

**private** UsuarioDao usuarioDao;

**private** Usuario usuario = **new** Usuario();

**public** String efetuarLogin(){

Usuario usuarioEncontrado = usuarioDao.pesquisarUsuario(**this**.usuario);

**if**(usuarioEncontrado != **null**){

usuarioLogado.logar(usuarioEncontrado);

**return** "menu\_principal?faces-redirect=true";

}**else**{

**this**.usuario = **new** Usuario();

**return** "login?faces-redirect=true";

}

}

**public** String efetuarLogout(){

usuarioLogado.deslogar();

**this**.usuario = **new** Usuario();

**return** "login";

}

**public** **boolean** isOpcaoCadastroUsuarioHabilitada() {

opcaoCadastroUsuarioHabilitada = !verificadorBean.existeFuncionarioEUsuarioCadastrado();

**return** opcaoCadastroUsuarioHabilitada;

}

**public** Usuario getUsuario(){

**return** **this**.usuario;

}

}

Na **Listagem 34** temos o código do LoginBean. Vamos as explicações dos métodos.

O método efetuarLogin primeiramente passa ao método pesquisarUsuario um objeto usuario que armazena o usuário e senha digitados na tela de login. Em seguida é checado se a invocação do método pesquisarUsuario retornou algum objeto do tipo Usuario(o que significa que localizou um registro no banco correspondente ao usuário e senha digitados na tela) ou se retornou null(o que significa que não foi localizado nenhum registro no banco). Caso tenha sido retornado um objeto do tipo Usuario ele passa a ser o usuário logado no sistema depois que o método logar é invocado e logo em seguida é feito um redirect para a página menu\_principal.xhtml. Caso tenha sido retornado null é feito um redirect para a página de login.

O método efetuarLogout faz logout do usuário que estava logado e redireciona para a tela de login.

O método isOpcaoCadastroUsuarioHabilitada retorna true somente se ainda não tiverem sido cadastrados nenhum funcionário da biblioteca e nenhum usuário.

Criando a página principal do sistema

**Listagem 35.** Página menu\_principal.xhtml

</html>

<?xml version=*"1.0"* encoding=*"ISO-8859-1"* ?>

<!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:f=*"http://java.sun.com/jsf/core"*

xmlns:h=*"http://java.sun.com/jsf/html"*

xmlns:ui=*"http://java.sun.com/jsf/facelets"*>

<ui:composition template=*"/WEB-INF/\_template\_simples.xhtml"*>

<ui:define name=*"corpo"*>

<h:form>

<h:commandLink value=*"Realizar Empréstimo de Livro"* action=*"emprestimo?faces-redirect=true"* />

<br/>

<h:commandLink value=*"Realizar Devolução de Livro"* action=*"devolucao?faces-redirect=true"* />

<br/>

<h:commandLink value=*"Cadastro de Funcionário e Usuário"* action=*"funcionario\_biblioteca?faces-redirect=true"*

rendered=*"#{usuarioLogadoBean.usuario.admin}"* />

<!-- restante dos links omitidos -->

</h:form>

</ui:define>

</ui:composition>

</html>

Quando o usuário se autenticar com sucesso no sistema será redirecionado para a página menu\_principal.xhtml. O código dessa página pode ser visualizado na **Listagem 35**. Ela é apenas uma página simples com links para as outras páginas, e os links são gerados através da tag h:commandLink. Uma coisa que é importante observar é que na última tag h:commandLink foi utilizado o atributo rendered para definir que o link para cadastrar funcionários e usuários só deve ser renderizado se o usuário logado for administrador(a EL usada para atingir esse resultado foi #{usuarioLogadoBean.usuario.admin}). Qualquer usuário que não seja administrador não terá acesso para cadastrar outros usuários.

Criando classes para navegação

**Listagem 36.** Classe Navegador

**package** br.com.easyjavamagazine.util;

**import** javax.faces.application.NavigationHandler;

**import** javax.faces.context.FacesContext;

**import** javax.inject.Inject;

**public** **class** Navegador {

@Inject

**private** NavigationHandler navigationHandler;

@Inject

**private** FacesContext facesContext;

**public** **void** forwardTo(String url) {

navigationHandler.handleNavigation(facesContext, **null**, url);

facesContext.renderResponse();

}

}

Em nosso projeto em alguns momento precisaremos fazer um forward para alguma página específica. Para quem não está familiarizado com o termo forward, pense nele como uma forma de mandar o cliente para outra página. A **Listagem 36** mostra o código da classe Navegador que irá realizar o forward quando precisarmos, através do método forwardTo que recebe a url de destino como argumento. Esse método usa duas classes da API do JSF(NavigationHandler e FacesContext) para poder direcionar o cliente para a url que desejamos. Para que possam ser injetados um NavigationHandler e um FacesContext é necessário criarmos classes que produzam esses tipos de objetos.

**Listagem 37.** Fábrica de NavigationHandler

**package** br.com.easyjavamagazine.fabricas;

**import** javax.enterprise.context.RequestScoped;

**import** javax.enterprise.inject.Produces;

**import** javax.faces.application.Application;

**import** javax.faces.application.NavigationHandler;

**import** javax.faces.context.FacesContext;

**import** javax.inject.Inject;

**public** **class** FabricaDeNavigationHandler {

@Inject

FacesContext facesContext;

@Produces @RequestScoped

**public** NavigationHandler criarNavigationHandler() {

**if** (facesContext != **null**) {

Application application = facesContext.getApplication();

**if** (application != **null**) {

**return** application.getNavigationHandler();

}

}

**return** **null**;

}

}

A **Listagem 37** mostra a classe que produz objetos do tipo NavigationHandler. A classe tem um único método produtor chamado criarNavigationHandler que a partir de um objeto FacesContext obtem um objeto Application e através dele retorna um objeto NavigationHandler.

**Listagem 38.** Fábrica de FacesContext

**package** br.com.easyjavamagazine.fabricas;

**import** javax.enterprise.context.RequestScoped;

**import** javax.enterprise.inject.Produces;

**import** javax.faces.context.FacesContext;

**public** **class** FabricaDeFacesContext {

**public** FabricaDeFacesContext() {

}

@Produces @RequestScoped

**public** FacesContext criarFacesContext() {

**return** FacesContext.*getCurrentInstance*();

}

}

A **Listagem 38** mostra a classe que produz objetos do tipo FacesContext. A classe apresenta um único método produtor chamado criarFacesContext que apenas retorna o objeto FacesContext obtido pela invocação FacesContext.getCurrentInstance().

Criando o cadastro de Funcionários e Usuários

**Listagem 39.** Página funcionario\_biblioteca.xhtml

<?xml version=*"1.0"* encoding=*"ISO-8859-1"* ?>

<!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:f=*"http://java.sun.com/jsf/core"*

xmlns:h=*"http://java.sun.com/jsf/html"*

xmlns:ui=*"http://java.sun.com/jsf/facelets"*>

<ui:composition template=*"/WEB-INF/\_template.xhtml"*>

<ui:define name=*"corpo"*>

<h:form>

<h2>

<h:outputText value=*"Novo Funcionário"* rendered=*"#{empty funcionarioBibliotecaBean.funcionarioBiblioteca.id}"* />

<h:outputText value=*"Editar Funcionário"* rendered=*"#{not empty funcionarioBibliotecaBean.funcionarioBiblioteca.id}"* />

</h2>

<fieldset>

<legend>Dados do Funcionário</legend>

<h:inputHidden value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.id}"* />

<h:inputHidden value=*"#{funcionarioBibliotecaBean.telefoneFixo.id}"* />

<h:inputHidden value=*"#{funcionarioBibliotecaBean.telefoneCelular.id}"* />

<h:inputHidden value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.id}"* />

<h:inputHidden value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.usuario.id}"* />

<h:outputLabel value=*"Nome do Funcionario:"* for=*"nome"* />

<h:inputText id=*"nome"* value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.nome}"* />

<h:outputLabel value=*"E-mail:"* for=*"email"* />

<h:inputText id=*"email"* value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.email}"* />

<h:outputLabel value=*"CPF:"* for=*"cpf"* />

<h:inputText id=*"cpf"* value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.cpf}"* />

<h:outputLabel value=*"Sexo:"* for=*"sexo"* />

<h:selectOneRadio id=*"sexo"* value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.sexo}"*>

<f:selectItem itemValue=*"M"* itemLabel=*"Masculino"* />

<f:selectItem itemValue=*"F"* itemLabel=*"Feminino"* />

</h:selectOneRadio>

<h:outputLabel value=*"Telefone Fixo:"* for=*"telefoneFixo"* />

<h:inputText id=*"telefoneFixo"* value=*"#{funcionarioBibliotecaBean.telefoneFixo.numero}"* />

<h:outputLabel value=*"Telefone Celular:"* for=*"telefoneCelular"* />

<h:inputText id=*"telefoneCelular"* value=*"#{funcionarioBibliotecaBean.telefoneCelular.numero}"* />

<h:outputLabel value=*"Endereço:"* for=*"endereco"* />

<h:inputText id=*"endereco"* value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.endereco}"* />

<h:outputLabel value=*"Complemento:"* for=*"complemento"* />

<h:inputText id=*"complemento"* value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.complemento}"* />

<h:outputLabel value=*"Cep:"* for=*"cep"* />

<h:inputText id=*"cep"* value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.cep}"* />

<h:outputLabel value=*"Bairro:"* for=*"bairro"* />

<h:inputText id=*"bairro"* value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.bairro}"* />

<h:outputLabel value=*"Cidade:"* for=*"cidade"* />

<h:inputText id=*"cidade"* value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.cidade}"* />

<h:outputLabel value=*"Estado:"* for=*"estado"* />

<h:inputText id=*"estado"* value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.estado}"* />

</fieldset>

<fieldset>

<legend>Dados de Login</legend>

<h:outputLabel value=*"Usuário:"* for=*"usuario"* />

<h:inputText id=*"usuario"* value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.usuario.usuario}"* />

<h:outputLabel value=*"Senha:"* for=*"senha"* />

<h:inputSecret id=*"senha"* value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.usuario.senha}"* />

<h:outputLabel value=*"O usuário é administrador?"* for=*"admin"* />

<h:selectOneRadio id=*"admin"* value=*"#{funcionarioBibliotecaBean.funcionarioBiblioteca.usuario.admin}"*

disabled=*"#{funcionarioBibliotecaBean.campoAdminDesabilitado}"*>

<f:selectItem itemValue=*"true"* itemLabel=*"Sim"* />

<f:selectItem itemValue=*"false"* itemLabel=*"Não"* />

</h:selectOneRadio>

<h:commandButton value=*"Salvar"* action=*"#{funcionarioBibliotecaBean.salvar}"* />

</fieldset>

</h:form>

<h:form>

<h2>Listagem de Funcionários da Biblioteca</h2>

<h:dataTable value=*"#{funcionarioBibliotecaBean.funcionariosBiblioteca}"* var=*"funcionarioBiblioteca"* styleClass=*"dados"* rowClasses=*"impar, par"*>

<h:column>

<f:facet name=*"header"*>Nome do Funcionario</f:facet>

#{funcionarioBiblioteca.nome}

</h:column>

<h:column>

<f:facet name=*"header"*>Usuário</f:facet>

#{funcionarioBiblioteca.usuario.usuario}, administrador: #{funcionarioBiblioteca.usuario.admin ? "Sim" : "Não"}

</h:column>

<h:column>

<f:facet name=*"header"*>E-mail</f:facet>

#{funcionarioBiblioteca.email}

</h:column>

<h:column>

<f:facet name=*"header"*>CPF</f:facet>

#{funcionarioBiblioteca.cpf}

</h:column>

<h:column>

<f:facet name=*"header"*>Sexo</f:facet>

#{funcionarioBiblioteca.sexo}

</h:column>

<h:column>

<f:facet name=*"header"*>Telefones</f:facet>

#{funcionarioBiblioteca.telefones[0].numero}, #{funcionarioBiblioteca.telefones[1].numero}

</h:column>

<h:column>

<f:facet name=*"header"*>Endereço</f:facet>

#{funcionarioBiblioteca.endereco.endereco}, #{funcionarioBiblioteca.endereco.complemento}, CEP: #{funcionarioBiblioteca.endereco.cep}

</h:column>

<h:column>

<f:facet name=*"header"*>Bairro</f:facet>

#{funcionarioBiblioteca.endereco.bairro}

</h:column>

<h:column>

<f:facet name=*"header"*>Cidade</f:facet>

#{funcionarioBiblioteca.endereco.cidade}

</h:column>

<h:column>

<f:facet name=*"header"*>Estado</f:facet>

#{funcionarioBiblioteca.endereco.estado}

</h:column>

<h:column>

<f:facet name=*"header"*>Ações</f:facet>

<h:commandLink action=*"#{funcionarioBibliotecaBean.remover(funcionarioBiblioteca)}"* value=*"Remover"* />

&nbsp;

<h:commandLink action=*"#{funcionarioBibliotecaBean.alterar(funcionarioBiblioteca)}"* value=*"Alterar"* />

</h:column>

</h:dataTable>

<f:metadata>

<f:event type=*"preRenderView"* listener=*"#{funcionarioBibliotecaBean.redirecionarParaLogin}"* />

</f:metadata>

</h:form>

</ui:define>

</ui:composition>

</html>

Na **Listagem 39** podemos ver o código da página funcionário\_biblioteca.xhtml. Essa página é composta por duas grandes seções. A primeira seção cosiste em um formulário com componentes para cadastrar um funcionário da biblioteca e suas informações pessoais além de permitir que se cadastre também um usuário e uma senha que serão vinculados ao funcionário. A segunda seção cosiste em uma tabela que lista as informações de todos funcionários cadastrados além de apresentar opções para exclusão e edição. Na parte do formulário foi usado o conceito de vincular o valor dos componentes as propriedades do FuncionarioBibliotecaBean. Já a tabela é construida com o componente h:dataTable e também associamos o valor desse componente a uma propriedade do managed bean. No atributo value da tag h:dataTable colocamos a EL #{funcionarioBibliotecaBean.funcionariosBiblioteca}, o que fará com que o componente invoque o método getFuncionariosBiblioteca do bean que retorna uma lista de objetos do tipo FuncionarioBiblioteca que será usada para popular a tabela.

É importante ressaltar que os valores dos componentes são associados com as propriedades do managed bean, e essas propriedades tem relação com os getters e setters e não com as variáveis membro da classe. Em outras palavras o bean tem uma propriedade só pelo fato de ter um getter e um setter e o JSF irá usar esses métodos acessores, mas como implementá-los fica por conta do desenvolvedor.

Algo interessante que deve ser observado é que foi utilizado o atributo disabled de um dos componentes (h:selectOneRadio com id admin) para podermos habilitá-lo ou desabilitá-lo de acordo com a propriedade campoAdminDesabilitado do managed bean FuncionarioBibliotecaBean. Esse componente é utilizado para indicar se o usuário é administrador, mas como temos uma regra de negócio que dita que o primeiro usuário a ser cadastrado deve ser administrador esse componente é desabilitado na hora do primeiro cadastro.

Outro ponto interessante nessa página é o uso da tag f:event que especifica um listener para um evento. Também deve ser informado o tipo do evento, que nesse caso é preRenderView. Esse tipo de evento será disparado antes da página JSF ser exibida e o método que funciona como listener(redirecionarParaLogin) será chamado. O método redirecionarParaLogin tem a função de direcionar o usuário para a tela de login quando necessário.

**Listagem 40.** Managed Bean FuncionarioBibliotecaBean

**package** br.com.easyjavamagazine.mb;

**import** java.util.List;

**import** javax.enterprise.context.RequestScoped;

**import** javax.inject.Inject;

**import** javax.inject.Named;

**import** br.com.easyjavamagazine.dao.FuncionarioBibliotecaDao;

**import** br.com.easyjavamagazine.entidades.Endereco;

**import** br.com.easyjavamagazine.entidades.FuncionarioBiblioteca;

**import** br.com.easyjavamagazine.entidades.Telefone;

**import** br.com.easyjavamagazine.entidades.Usuario;

**import** br.com.easyjavamagazine.interceptadores.Transacional;

**import** br.com.easyjavamagazine.util.Navegador;

@Named

@RequestScoped

**public** **class** FuncionarioBibliotecaBean {

@Inject

**private** FuncionarioBibliotecaDao funcionarioBibliotecaDao;

@Inject

**private** FuncionarioEUsuarioVerificadorBean verificadorBean;

**private** FuncionarioBiblioteca funcionarioBiblioteca = **new** FuncionarioBiblioteca();

**private** Telefone telefoneFixo = **new** Telefone();

**private** Telefone telefoneCelular = **new** Telefone();

**private** List<FuncionarioBiblioteca> funcionariosBiblioteca;

@Inject

**private** Navegador navegador;

**private** **boolean** redirecionarParaLogin;

**private** **boolean** campoAdminDesabilitado;

FuncionarioBibliotecaBean(){

funcionarioBiblioteca.setEndereco(**new** Endereco());

Usuario usuario = **new** Usuario();

usuario.setFuncionarioBiblioteca(funcionarioBiblioteca);

funcionarioBiblioteca.setUsuario(usuario);

}

@Transacional

**public** **void** salvar(){

**if**(isCampoAdminDesabilitado()){

redirecionarParaLogin = **true**;

}

**if**(funcionarioBiblioteca.getId() == **null**){

vincularTelefones();

funcionarioBibliotecaDao.inserir(funcionarioBiblioteca);

}**else**{

funcionarioBiblioteca.getTelefones().clear();

vincularTelefones();

funcionarioBibliotecaDao.atualizar(funcionarioBiblioteca);

}

telefoneFixo = **new** Telefone();

telefoneCelular = **new** Telefone();

funcionarioBiblioteca = **new** FuncionarioBiblioteca();

funcionariosBiblioteca = funcionarioBibliotecaDao.listarFuncionariosBiblioteca();

}

**public** **void** vincularTelefones(){

telefoneFixo.setFixo(**true**);

funcionarioBiblioteca.getTelefones().add(telefoneFixo);

funcionarioBiblioteca.getTelefones().add(telefoneCelular);

}

**public** **void** alterar(FuncionarioBiblioteca funcionarioBiblioteca){

**this**.funcionarioBiblioteca = funcionarioBiblioteca;

**for**(Telefone telefone : funcionarioBiblioteca.getTelefones()){

**if**(telefone.isFixo()){

**this**.telefoneFixo = telefone;

}**else**{

**this**.telefoneCelular = telefone;

}

}

}

@Transacional

**public** **void** remover(FuncionarioBiblioteca funcionarioBiblioteca){

funcionarioBibliotecaDao.remover(funcionarioBiblioteca);

funcionariosBiblioteca = funcionarioBibliotecaDao.listarFuncionariosBiblioteca();

}

**public** **void** redirecionarParaLogin(){

**if**(redirecionarParaLogin)

navegador.forwardTo("login");

}

@Transacional

**public** List<FuncionarioBiblioteca> getFuncionariosBiblioteca(){

**if**(funcionariosBiblioteca == **null**){

funcionariosBiblioteca = funcionarioBibliotecaDao.listarFuncionariosBiblioteca();

}

**return** funcionariosBiblioteca;

}

**public** FuncionarioBiblioteca getFuncionarioBiblioteca() {

**return** funcionarioBiblioteca;

}

**public** **void** setFuncionarioBiblioteca(FuncionarioBiblioteca funcionarioBiblioteca) {

**this**.funcionarioBiblioteca = funcionarioBiblioteca;

}

**public** Telefone getTelefoneFixo() {

**return** telefoneFixo;

}

**public** **void** setTelefoneFixo(Telefone telefoneFixo) {

**this**.telefoneFixo = telefoneFixo;

}

**public** Telefone getTelefoneCelular() {

**return** telefoneCelular;

}

**public** **void** setTelefoneCelular(Telefone telefoneCelular) {

**this**.telefoneCelular = telefoneCelular;

}

**public** **boolean** isCampoAdminDesabilitado() {

campoAdminDesabilitado = !verificadorBean.existeFuncionarioEUsuarioCadastrado();

**return** campoAdminDesabilitado;

}

}

Na **Listagem 40** temos o código do managed bean FuncionarioBibliotecaBean. Dentro do construtor da classe são estabelecidas as relações entre os objetos e o funcionário da biblioteca é associado a um endereço e a um usuário. Porém no momento que o managed bean acaba de ser instanciado(pelo container), tanto o objeto que representa o funcionário da biblioteca como os demais estão “vazios” pois também acabaram de ser instanciados(através do operador new). Os dados que irão preencher as propriedades desses objetos virão dos valores dos componentes da tela.

Nessa classe temos uma série de métodos importantes que serão explicados.

Iremos começar as explicações pelo método isCampoAdminDesabilitado. Para entendermos o motivo desse método existir temos que entender em que situações o managed bean é usado. O FuncionarioBibliotecaBean será usado em duas situações:

* Quando o usuário do sistema estiver logado e for administrador, poderá clicar no link para cadastramento de funcionários e usuários e consequentemente irá utilizar o FuncionarioBibliotecaBean. Nessa situação o método isCampoAdminDesabilitado deve retornar false e o usuário logado poderá escolher entre cadastrar um novo usuário administrador ou cadastrar um usuário comum(não administrador).
* Quando o usuário do sistema estiver na tela de login e não houver nenhum funcionário nem usuário cadastrados irá aparecer para ele o botão "Criar usuário administrador". Nessa situação, quando ele for cadastrar o primeiro funcionário e o primeiro usuário(ambos relacionados), esse usuário será administrador e por isso o método isCampoAdminDesabilitado deve retornar true e consequentemente o componente da tela que define se o usuário é admin ou não será desabilitado. Aproveitando o gancho, após o cadastramento do primeiro funcionário e usuário o cliente do sistema será direcionado para a página de login novamente para poder se autenticar(isso só acontecerá na primeira vez, ao realizar o segundo cadastro por exemplo não haverá essa mudança para a tela de login). Parte dessa lógica para fazer o forward para a página login.xhtml está no método salvar, então a seguir iremos explicá-lo.

O método salvar verifica se está sendo salvo o primeiro funcionário e seu respectivo login(a verificação é baseada no retorno do método isCampoAdminDesabilitado que foi explicado anteriormente). Caso seja o primeiro usuário, a variável redirecionarParaLogin é setada para true e por esse motivo o método redirecionarParaLogin que atua como listener(conforme explicado na listagem anterior) irá realizar o forward(só faz forward se a variável for true). Ainda no método salvar tem um condicional que verifica se o id do funcionário da biblioteca é nulo. Se o id for nulo indica que está sendo inserido um funcionário novo e não editado um funcionário existente(pois caso o funcionário já existisse na base ele teria um id). Se é um funcionário novo passamos o objeto para o método inserir do DAO, se é edição de funcionário existente passamos o objeto para o método atualizar do DAO. Antes de passarmos o funcionário para ser salvo ou atualizado no banco de dados, vinculamos a ele os telefones através do método vincularTelefones. No final do método “resetamos” os objetos usando o operador new, pois dessa forma como os componentes da página estão vinculados a propriedades relacionadas a esses objetos, essa é uma maneira de limpar os dados dos componentes e assim “resetamos” o formulário de cadastro. Por último atualizamos a lista de funcionários do managed bean com os dados que estão no banco, para que ela reflita o insert ou update que foi feito e para que isso também seja refletido no conteúdo do componente h:dataTable da página funcionário\_biblioteca.xhtml. Note que o método salvar usa a anotação @Transacional que criamos para que seja executado dentro de uma transação.

O método alterar não irá atualizar o objeto no banco de dados, pois como já vimos o método salvar que irá persistir as alterações na base. O papel do método alterar é viabilizar que os dados do objeto a ser alterado sejam mostrados no formulário, permitindo sua edição. Para que esse método possa fazer seu trabalho ele recebe o objeto funcionário que deve ser alterado como argumento, atribui esse objeto a variável de instância funcionarioBiblioteca. Também atribui os telefones recuperados a partir do objeto que deve ser alterado as variáveis de instância telefoneFixo e telefoneCelular. Isso tudo é feito porque os componentes de formulário da página irão usar os getters dessas variáveis de instância para mostrar os dados na tela para edição.

O método remover recebe como argumento o funcionário da biblioteca a ser removido. O funcionário a ser removido é repassado para o método remover do DAO e logo em seguida a lista de funcionários do bean é atualizada com o conteúdo presente no banco de dados.

Criando a funcionalidade para empréstimo de livros

**Listagem 41.** Página emprestimo.xhtml

<?xml version=*"1.0"* encoding=*"ISO-8859-1"* ?>

<!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:f=*"http://java.sun.com/jsf/core"*

xmlns:h=*"http://java.sun.com/jsf/html"*

xmlns:ui=*"http://java.sun.com/jsf/facelets"*>

<ui:composition template=*"/WEB-INF/\_template.xhtml"*>

<ui:define name=*"corpo"*>

<h:form>

<h2>

<h:outputText value=*"Empréstimo"* />

</h2>

<fieldset>

<legend>Dados do Empréstimo</legend>

<h:outputLabel value=*"Escolher livro a emprestar:"* for=*"livro"* />

<h:selectOneMenu id=*"livro"* value=*"#{emprestimoBean.idLivro}"*>

<f:selectItems value=*"#{emprestimoBean.livrosDisponiveis}"* var=*"livro"* itemValue=*"#{livro.id}"* itemLabel=*"#{livro.nome}"* />

</h:selectOneMenu>

<h:outputLabel value=*"Leitor:"* for=*"leitor"* />

<h:selectOneMenu id=*"leitor"* value=*"#{emprestimoBean.idLeitor}"*>

<f:selectItems value=*"#{leitorBean.leitores}"* var=*"leitor"* itemValue=*"#{leitor.id}"* itemLabel=*"#{leitor.nome}"* />

</h:selectOneMenu>

<h:outputLabel value=*"Data do empréstimo:"* for=*"dataEmprestimo"* />

<h:inputText id=*"dataEmprestimo"* value=*"#{emprestimoBean.dataEmprestimoFormatada}"* disabled=*"true"* />

<h:outputLabel value=*"Data prevista para devolução:"* for=*"dataPrevista"* />

<h:inputText id=*"dataPrevista"* value=*"#{emprestimoBean.dataPrevistaFormatada}"* disabled=*"true"* />

<h:commandButton value=*"Realizar Empréstimo"* action=*"#{emprestimoBean.realizarEmprestimo}"* />

</fieldset>

</h:form>

</ui:define>

</ui:composition>

</html>

O nosso próximo passo será implementarmos a funcionalidade para empréstimo de livros. Na **Listagem 41** é apresentado o código da página emprestimo.xhtml que nada mais é do que um formulário que irá registrar o empréstimo salvando suas informações no banco de dados. No formulário apresentado nessa página é possível escolher o livro que será emprestado e para qual leitor será emprestado.

Para listarmos todos livros disponíveis para empréstimo usamos os componentes h:selectOneMenu(que irá gerar um elemento HTML selet) e f:selectItems(que irá gerar elementos HTML option). Através do componente h:selectOneMenu é possível associar o valor do item selecionado com uma propriedade do managed bean, que nesse caso é a propriedade idLivro. Também é importante entender para que servem os atributos do componente f:selectItems:

* value – colocamos nesse atributo a EL #{emprestimoBean.livrosDisponiveis}, para indicar que os livros que devem ser listados devem ser provindos da propriedade livrosDisponiveis do managed bean(será invocado o método getLivrosDisponiveis no bean).
* itemValue – define o que será setado na propriedade idLivro do bean. Como podemos notar na EL – itemValue="#{livro.id}" – será setado o id do livro selecionado.
* itemLabel – define as legendas que serão mostradas ao usuário.

Para que sejam listados todos os leitores também usamos as mesmas tags h:selectOneMenu e f:selectItems. O managed bean também terá uma propriedade idLeitor para receber o id do leitor selecionado.

Também serão exibidas a data do empréstimo e a data prevista para devolução do livro. Usamos o componente h:inputText para exibir cada uma das datas, porém configuramos seu atributo disabled para true para que o usuário do sistema não possa alterar essas datas que serão geradas automaticamente.

O último componente do formulário é um h:commandButton que irá acionar o método realizarEmprestimo do managed bean quando clicado.

**Listagem 42.** Managed Bean LeitorBean

**package** br.com.easyjavamagazine.mb;

**import** java.util.List;

**import** javax.enterprise.context.RequestScoped;

**import** javax.inject.Inject;

**import** javax.inject.Named;

**import** br.com.easyjavamagazine.dao.LeitorDao;

**import** br.com.easyjavamagazine.entidades.Leitor;

**import** br.com.easyjavamagazine.interceptadores.Transacional;

@Named

@RequestScoped

**public** **class** LeitorBean {

//restante da implementação omitida

@Inject

**private** LeitorDao leitorDao;

**private** List<Leitor> leitores;

@Transacional

**public** List<Leitor> getLeitores(){

**if**(leitores == **null**){

leitores = leitorDao.listarLeitores();

}

**return** leitores;

}

//restante da implementação omitida

}

Como podemos ver na **Listagem 41** é utilizada a tag f:selectItems para que sejam exibidos os nomes dos leitores, e no atributo value dessa tag é colocada a EL #{leitorBean.leitores}. Então para que essa página funcione o managed bean LeitorBean precisa ser criado e seu código pode ser visto na **Listagem 42**. Deixamos essa classe com o mínimo de código necessário e permaneceu apenas o método getLeitores que irá recuperar a lista com todos leitores do banco de dados através de uma classe DAO.

**Listagem 43.** Managed Bean EmprestimoBean

**package** br.com.easyjavamagazine.mb;

**import** java.time.LocalDate;

**import** java.time.format.DateTimeFormatter;

**import** java.util.List;

**import** javax.enterprise.context.RequestScoped;

**import** javax.inject.Inject;

**import** javax.inject.Named;

**import** br.com.easyjavamagazine.dao.EmprestimoDao;

**import** br.com.easyjavamagazine.dao.LeitorDao;

**import** br.com.easyjavamagazine.dao.LivroDao;

**import** br.com.easyjavamagazine.entidades.Emprestimo;

**import** br.com.easyjavamagazine.entidades.FuncionarioBiblioteca;

**import** br.com.easyjavamagazine.entidades.Leitor;

**import** br.com.easyjavamagazine.entidades.Livro;

**import** br.com.easyjavamagazine.interceptadores.Transacional;

**import** br.com.easyjavamagazine.util.MensagemUtil;

@Named

@RequestScoped

**public** **class** EmprestimoBean {

@Inject

**private** EmprestimoDao emprestimoDao;

**private** Emprestimo emprestimo = **new** Emprestimo();

@Inject

**private** LivroDao livroDao;

**private** Integer idLivro;

**private** List<Livro> livrosDisponiveis;

@Inject

**private** LeitorDao leitorDao;

**private** Integer idLeitor;

@Inject

**private** UsuarioLogadoBean usuarioLogadoBean;

String dataEmprestimoFormatada;

String dataPrevistaFormatada;

EmprestimoBean(){

emprestimo.setDataEmprestimo(LocalDate.*now*());

emprestimo.setDataPrevista(LocalDate.*now*().plusDays(7));

DateTimeFormatter formatador = DateTimeFormatter.*ofPattern*("dd/MM/yyyy");

dataEmprestimoFormatada = emprestimo.getDataEmprestimo().format(formatador);

dataPrevistaFormatada = emprestimo.getDataPrevista().format(formatador);

}

@Transacional

**public** **void** realizarEmprestimo(){

Livro livro = livroDao.pesquisarPorId(idLivro);

Leitor leitor = leitorDao.pesquisarPorId(idLeitor);

FuncionarioBiblioteca funcionarioBiblioteca = usuarioLogadoBean.getUsuario().getFuncionarioBiblioteca();

emprestimo.setLivro(livro);

livro.getEmprestimos().add(emprestimo);

emprestimo.setLeitor(leitor);

emprestimo.setFuncionarioBiblioteca(funcionarioBiblioteca);

emprestimoDao.inserir(emprestimo);

livrosDisponiveis = livroDao.listarLivrosDisponiveisParaEmprestimo();

MensagemUtil.*addMensagemInformativa*("Sucesso - ", "Empréstimo realizado com sucesso!");

}

**public** Emprestimo getEmprestimo() {

**return** emprestimo;

}

**public** **void** setEmprestimo(Emprestimo emprestimo) {

**this**.emprestimo = emprestimo;

}

**public** Integer getIdLivro() {

**return** idLivro;

}

**public** **void** setIdLivro(Integer idLivro) {

**this**.idLivro = idLivro;

}

@Transacional

**public** List<Livro> getLivrosDisponiveis() {

**if**(livrosDisponiveis == **null**){

livrosDisponiveis = livroDao.listarLivrosDisponiveisParaEmprestimo();

}

**return** livrosDisponiveis;

}

**public** Integer getIdLeitor() {

**return** idLeitor;

}

**public** **void** setIdLeitor(Integer idLeitor) {

**this**.idLeitor = idLeitor;

}

**public** String getDataEmprestimoFormatada() {

**return** dataEmprestimoFormatada;

}

**public** String getDataPrevistaFormatada() {

**return** dataPrevistaFormatada;

}

}

Na **Listagem 43** podemos visualizar o código do managed bean EmprestimoBean. No construtor dessa classe são invocados métodos setters no objeto que representa o empréstimo para que sejam definidas a data do empréstimo e a data prevista para devolução do livro. A data do empréstimo é a data atual e a data prevista para devolução é sete dias após a data atual. Observe que estamos usando a nova API de data e hora do Java 8 que simplifica bastante nosso trabalho. Fica bem mais simples somar sete dias a data atual, pois a API provê uma interface fluente que possibilita o encadeamento de chamadas de métodos o que geralmente torna o código mais fácil de ler. Para que essas duas datas sejam mostradas ao usuário no formato dd/MM/yyyy usamos a classe DateTimeFormatter – disponível a partir do Java 8 – para formata-las. As datas formatadas como String são armazenadas nas variáveis dataEmprestimoFormatada e dataPrevistaFormatada, e os componentes da página empréstimo.xhtml irão invocar os seus respectivos métodos getters para recuperar essas datas e exibí-las para o usuário.

O principal método desse bean é o realizarEmprestimo que contém a lógica referente ao empréstimo de livro. Colocamos a anotação @Transacional para que ele seja interceptado pelo nosso TransacionalInterceptor, e seja executado dentro de uma transação. Antes de persistirmos o empréstimo, precisamos relaciona-lo com o livro que está sendo emprestado, com o leitor que está tomando o livro emprestado e com o funcionário da biblioteca que está operando o sistema no momento do empréstimo. Conforme explicado na listagem anterior a classe possui duas propriedades chamadas idLivro e idLeitor que armazenam os ids dos objetos, e usamos esses ids para recuperar tanto o livro quando o leitor do banco de dados e assim podemos associa-los ao empréstimo. Como existe um relacionamento entre o usuário logado e o funcionário ao qual o usuário pertence, recuperamos o funcionário da biblioteca através do usuário logado e também o associamos ao empréstimo. Feitas as associações entre os objetos o objeto que representa o empréstimo é persistido e a lista de livros disponíveis é atualizada. Ao final também é adicionada uma mensagem indicando que a operação foi realizada com sucesso que será exibida na página.

Criando a funcionalidade para devolução de livros

**Listagem 44.** Página devolucao.xhtml

<?xml version=*"1.0"* encoding=*"ISO-8859-1"* ?>

<!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:f=*"http://java.sun.com/jsf/core"*

xmlns:h=*"http://java.sun.com/jsf/html"*

xmlns:ui=*"http://java.sun.com/jsf/facelets"*>

<ui:composition template=*"/WEB-INF/\_template.xhtml"*>

<ui:define name=*"corpo"*>

<h:form id=*"formDevolucao"*>

<h2>

<h:outputText value=*"Devolução"* />

</h2>

<fieldset>

<legend>Dados da Devolução</legend>

<h:outputLabel value=*"Escolher livro a devolver:"* for=*"livro"* />

<h:selectOneMenu id=*"livro"* value=*"#{devolucaoBean.idLivro}"*>

<f:selectItem itemLabel=*""* itemValue=*"#{null}"* />

<f:selectItems value=*"#{devolucaoBean.livrosEmprestados}"* var=*"livro"* itemValue=*"#{livro.id}"* itemLabel=*"#{livro.nome}"* />

<f:ajax listener=*"#{devolucaoBean.alteradoLivroSelecionado}"* render=*"formDevolucao"* />

</h:selectOneMenu>

<h:outputLabel value=*"Emprestado para:"* for=*"emprestadoPara"* />

<h:inputText id=*"emprestadoPara"* value=*"#{devolucaoBean.emprestimo.leitor.nome}"* disabled=*"true"* />

<h:outputLabel value=*"Emprestado por:"* for=*"emprestadoPor"* />

<h:inputText id=*"emprestadoPor"* value=*"#{devolucaoBean.emprestimo.funcionarioBiblioteca.nome}"* disabled=*"true"* />

<h:outputLabel value=*"Data do empréstimo:"* for=*"dataEmprestimo"* />

<h:inputText id=*"dataEmprestimo"* value=*"#{devolucaoBean.dataEmprestimoFormatada}"* disabled=*"true"* />

<h:outputLabel value=*"Data prevista para devolução:"* for=*"dataPrevista"* />

<h:inputText id=*"dataPrevista"* value=*"#{devolucaoBean.dataPrevistaFormatada}"* disabled=*"true"* />

<h:outputLabel value=*"Multa por atraso:"* for=*"multa"* />

<h:inputText id=*"multa"* value=*"#{devolucaoBean.multa}"* disabled=*"true"* />

<h:commandButton value=*"Realizar Devolução"* action=*"#{devolucaoBean.realizarDevolucao}"* />

</fieldset>

</h:form>

</ui:define>

</ui:composition>

</html>

A **Listagem 44** exibe o código da página devolucao.xhtml, responsável por viabilizar a devolução do livro em conjunto com o DevolucaoBean. A página, através dos componentes h:selectOneMenu e f:selectItems – explicados anteriormente – exibe uma lista de livros que se encontram emprestados e são passíveis de devolução. Usamos também o componente f:selectItem para inserir uma opção vazia no início da listagem de livros emprestados apresentada na tela.

Uma tag que também foi usada, e que é fundamental ao funcionamento dessa tela, é a f:ajax. No atributo listener foi colocada a EL #{devolucaoBean.alteradoLivroSelecionado}, portanto toda vez que escolhermos um item na lista de livros emprestados o método alteradoLivroSelecionado do nosso bean será invocado. Após o método ser invocado o formulário da página será atualizado(renderizado novamente), pois informamos seu id no abributo render(render="formDevolucao"). Com esse mecanismo ao selecionarmos qualquer livro na lista os demais campos do formulários serão preenchidos automaticamente e o operador do sistema poderá visualizar os dados relacionados ao empréstimo daquele livro como o nome do leitor que pegou o livro emprestado, o nome do funcionário da biblioteca que realizou o empréstimo, a data do empréstimo e a data prevista para devolução e se existe multa a ser paga por atraso. A edição dos valores dos campos foi desabilitada através do atributo disabled="true".

**Listagem 45.** Managed Bean DevolucaoBean

**package** br.com.easyjavamagazine.mb;

**import** java.math.BigDecimal;

**import** java.time.LocalDate;

**import** java.time.format.DateTimeFormatter;

**import** java.util.List;

**import** javax.enterprise.context.RequestScoped;

**import** javax.inject.Inject;

**import** javax.inject.Named;

**import** br.com.easyjavamagazine.dao.EmprestimoDao;

**import** br.com.easyjavamagazine.dao.LivroDao;

**import** br.com.easyjavamagazine.entidades.Emprestimo;

**import** br.com.easyjavamagazine.entidades.Livro;

**import** br.com.easyjavamagazine.interceptadores.Transacional;

**import** br.com.easyjavamagazine.util.FormatadorDeNumeros;

**import** br.com.easyjavamagazine.util.MensagemUtil;

@Named

@RequestScoped

**public** **class** DevolucaoBean {

@Inject

**private** LivroDao livroDao;

**private** Integer idLivro;

**private** List<Livro> livrosEmprestados;

@Inject

**private** EmprestimoDao emprestimoDao;

**private** Emprestimo emprestimo = **new** Emprestimo();

**private** String dataEmprestimoFormatada;

**private** String dataPrevistaFormatada;

**private** String multa = FormatadorDeNumeros.*formatarBigDecimalComoMoeda*(**new** BigDecimal(0.0d));

**public** **void** alteradoLivroSelecionado(){

**if**(idLivro != **null**){

setarEmprestimo();

DateTimeFormatter formatador = DateTimeFormatter.*ofPattern*("dd/MM/yyyy");

dataEmprestimoFormatada = emprestimo.getDataEmprestimo().format(formatador);

dataPrevistaFormatada = emprestimo.getDataPrevista().format(formatador);

LocalDate dataAtual = LocalDate.*now*();

**if**(dataAtual.isAfter(emprestimo.getDataPrevista())){

multa = FormatadorDeNumeros.*formatarBigDecimalComoMoeda*(**new** BigDecimal(15.0d));

}

}**else**{

limparCampos();

}

}

@Transacional

**public** **void** realizarDevolucao(){

setarEmprestimo();

emprestimo.setDataDevolucao(LocalDate.*now*());

emprestimoDao.atualizar(emprestimo);

livrosEmprestados = livroDao.listarLivrosEmprestados();

limparCampos();

MensagemUtil.*addMensagemInformativa*("Sucesso - ", "Devolução realizada com sucesso!");

}

**private** **void** setarEmprestimo(){

Livro livroSelecionado = getLivroSelecionado();

Emprestimo emprestimoDoLivroSelecionado = getEmprestimoDoLivroSelecionado(livroSelecionado);

emprestimo = emprestimoDoLivroSelecionado;

}

**private** Livro getLivroSelecionado(){

**for**(Livro livro : getLivrosEmprestados()){

**if**(livro.getId() == idLivro){

**return** livro;

}

}

**return** **null**;

}

**private** Emprestimo getEmprestimoDoLivroSelecionado(Livro livroSelecionado){

**for**(Emprestimo emprestimo : livroSelecionado.getEmprestimos()){

**if**(emprestimo.getDataDevolucao() == **null**){

**return** emprestimo;

}

}

**return** **null**;

}

**private** **void** limparCampos(){

emprestimo = **new** Emprestimo();

dataEmprestimoFormatada = "";

dataPrevistaFormatada = "";

multa = FormatadorDeNumeros.*formatarBigDecimalComoMoeda*(**new** BigDecimal(0.0d));

}

**public** Integer getIdLivro() {

**return** idLivro;

}

**public** **void** setIdLivro(Integer idLivro) {

**this**.idLivro = idLivro;

}

@Transacional

**public** List<Livro> getLivrosEmprestados() {

**if**(livrosEmprestados == **null**){

livrosEmprestados = livroDao.listarLivrosEmprestados();

}

**return** livrosEmprestados;

}

**public** Emprestimo getEmprestimo() {

**return** emprestimo;

}

**public** **void** setEmprestimo(Emprestimo emprestimo) {

**this**.emprestimo = emprestimo;

}

**public** String getDataEmprestimoFormatada() {

**return** dataEmprestimoFormatada;

}

**public** **void** setDataEmprestimoFormatada(String dataEmprestimoFormatada) {

**this**.dataEmprestimoFormatada = dataEmprestimoFormatada;

}

**public** String getDataPrevistaFormatada() {

**return** dataPrevistaFormatada;

}

**public** **void** setDataPrevistaFormatada(String dataPrevistaFormatada) {

**this**.dataPrevistaFormatada = dataPrevistaFormatada;

}

**public** String getMulta() {

**return** multa;

}

**public** **void** setMulta(String multa) {

**this**.multa = multa;

}

}

Na **Listagem 45** podemos visualizar o código do managed bean DevolucaoBean. Essa classe possui alguns métodos importantes que serão explicados.

O método alteradoLivroSelecionado será invocado quando o usuário do sistema escolher algum livro na lista de livros emprestados que aparece na página devolucao.xhtml. Dentro do método é verificado se o conteúdo da variável idLivro está preenchido, o que significa que foi selecionado algum livro, ou caso o conteúdo da variável seja nulo significa que foi selecionada a opção em branco da lista. Se tiver sido selecionado algum livro o método setarEmprestimo é chamado para atribuir o objeto empréstimo relacionado aquele livro específico a variável de instância emprestimo. Se existir mais de um empréstimo vinculado ao livro, o que vale é aquele que ainda não tem data de devolução, pois essa data será atualizada quando a devolução for concretizada. Ainda dentro do método são recuperadas a data do emprésimo e a data prevista para devolução, formatadas e atribuídas as variáveis de instâncias dataEmprestimoFormatada e dataPrevistaFormatada para que possam ser exibidas na página. Em seguida é avaliado se a data atual é maior que a data prevista para empréstimo, e nesse caso é gerada uma multa de 15 reais. Para mantermos o sistema simples a multa nem será persistida, só será mostrada na tela para que seja cobrado algum valor pelo atraso. A lógica explicada se aplica se o operador do sistema selcionar algum livro na lista, mas caso ele selecione a opção em branco os valores dos demais campos do formulário também devem ficar em branco e nesse caso o método limparCampos é chamado.

O método principal dessa classe é o realizarDevolucao. O método basicamente recupera o empréstimo associado ao livro que foi selecionado para ser devolvido, atualiza a data de devolução desse empréstimo com a data atual, persiste as alterações do empréstimo no banco de dados através de uma classe DAO, atualiza a lista dos livros emprestados para que o livro que acabou de ser devolvido não apareça mais nela, limpa os campos chamando o método limparCampos e adiciona uma mensagem para que o usuário veja dizendo que a devolução foi realizada com sucesso. Repare que o método é marcado com a anotação @Transacional.

**Listagem 46.** Classe para formatação de números

**package** br.com.easyjavamagazine.util;

**import** java.math.BigDecimal;

**import** java.text.DecimalFormat;

**import** java.text.DecimalFormatSymbols;

**import** java.util.Locale;

**public** **class** FormatadorDeNumeros {

**public** **static** String formatarBigDecimalComoMoeda(BigDecimal numero){

Locale localeBrasil = **new** Locale("pt","BR");

DecimalFormatSymbols moedaReal = **new** DecimalFormatSymbols(localeBrasil);

DecimalFormat df = **new** DecimalFormat("¤ ###,###,##0.00", moedaReal);

**return** df.format(numero);

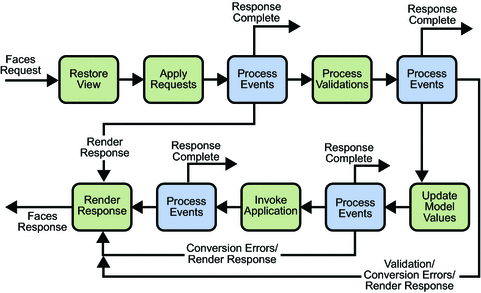
}

}

O managed bean DevolucaoBean mostrado anteriormente faz uso em alguns lugares(nos locais onde precisa formatar o valor da multa) do método formatarBigDecimalComoMoeda presente na classe FormatadorDeNumeros. O método recebe um objeto do tipo BigDecimal e retorna uma String que representa o valor desse BigDecimal no formato da moeda do Brasil, acrescentando o R$. Por exemplo ao passarmos para o método new BigDecimal(15.0d) receberemos de volta R$ 15,00. A **Listagem 46** mostra o código da classe e do método em questão.

Evitando acessos indevidos com PhaseListener

A aplicação está quase finalizada porém nesse momento ela tem um problema que precisa ser corrigido. Da forma que está agora, a aplicação permite que o usuário acesse qualquer página através da URL mesmo sem estar logado. E mesmo que o usuário tenha se logado, se ele não for administrador não deveria poder acessar a página de cadastro de funcionários e usuários, mas por enquanto ele consegue. Para resolver esse problema dos acessos indevidos vamos criar uma classe que implemente a interface PhaseListener da API do JSF, mas antes precisamos saber quais são as fases do ciclo de vida do processamento de uma requisição feita a uma página JSF e entender cada uma dessas fases.



**Figura 9.** Ciclo de vida de uma requisição JSF

A **Figura 9** mostra as 6 fases do ciclo de vida, são elas:

* **Restore view**: é nessa fase que a árvore de componentes UI, chamada de view, é criada caso seja a requisição inicial a página JSF. Se a view correspondente a página JSF já existe, ela é restaurada.
* **Apply request values**: nessa fase os valores vindos por parâmetros na requisição são atribuídos a cada componente da árvore.
* **Process validations**: nessa fase são processados todos validators associados aos componentes da árvore.
* **Update model values**: se não ocorreu nenhum erro de validação nem conversão nas etapas anteriores, isso significa que os valores são válidos, e são passados dos componentes para as propriedades do managed bean.
* **Invoke application**: depois do modelo ter sido atualizado na fase anterior, nessa fase é que são tratados os eventos disparados pelo usuário(por exemplo se ele clicou em um botão salvar no formulário) e executadas as regras de negócios.
* **Render response**: nessa fase será renderizada a resposta para o usuário.

**Listagem 47.** PhaseListener Autorizador

**package** br.com.easyjavamagazine.listeners;

**import** javax.faces.context.FacesContext;

**import** javax.faces.event.PhaseEvent;

**import** javax.faces.event.PhaseId;

**import** javax.faces.event.PhaseListener;

**import** javax.inject.Inject;

**import** br.com.easyjavamagazine.mb.FuncionarioEUsuarioVerificadorBean;

**import** br.com.easyjavamagazine.mb.UsuarioLogadoBean;

**import** br.com.easyjavamagazine.util.Navegador;

**public** **class** Autorizador **implements** PhaseListener {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

@Inject

**private** UsuarioLogadoBean usuarioLogado;

@Inject

FacesContext context;

@Inject

**private** Navegador navegador;

@Inject

**private** FuncionarioEUsuarioVerificadorBean verificadorBean;

@Override

**public** **void** afterPhase(PhaseEvent event) {

**if**(usuarioLogado.isLogado()){

**if**("/funcionario\_biblioteca.xhtml".equals(context.getViewRoot().getViewId()) &&

!usuarioLogado.getUsuario().isAdmin()){

navegador.forwardTo("menu\_principal");

}

}**else**{

**if**("/login.xhtml".equals(context.getViewRoot().getViewId())){

**return**;

}

**if**("/funcionario\_biblioteca.xhtml".equals(context.getViewRoot().getViewId()) &&

!verificadorBean.existeFuncionarioEUsuarioCadastrado()){

**return**;

}

navegador.forwardTo("login");

}

}

@Override

**public** **void** beforePhase(PhaseEvent event) {

// não é necessário fazer nada antes da fase Restore View

}

@Override

**public** PhaseId getPhaseId() {

**return** PhaseId.***RESTORE\_VIEW***;

}

}

Uma classe que implemente a interface PhaseListener será notificada antes e depois do processamento de cada uma dessas fases do ciclo de vida de uma requisição JSF, ou até mesmo podemos escolher a fase que nos interessa. Para nós só interessa sermos notificados antes e depois do processamento da fase Restore view. A **Listagem 47** mostra a implementação de PhaseListener usada no sistema de biblioteca, a classe Autorizador.

Sobrescrevemos os métodos beforePhase e afterPhase, porém nossa implementação do código ficou no afterPhase e deixamos o beforePhase vazio. Isso porque antes da fase Restore view a chamada ao método context.getViewRoot() retorna null. Também foi sobrescrito o método getPhaseId onde podemos informar qual fase do ciclo de vida nos interessa para sermos notificados.

Dentro do método afterPhase está toda a lógica que previne os acessos indevidos. Primeiramente verificamos se o usuário já se logou no sistema. Caso ele esteja logado e seja administrador poderá acessar qualquer página, mas caso não seja administrador não poderá acessar o cadastro de funcionários e usuário e se ele tentar será direcionado a página menu\_principal.xhtml. A outra situação é se o usuário ainda não tiver se autenticado, e nesse caso não poderá acessar nenhuma página do sistema, com exceção da página de login obviamente e da página de cadastro de funcionários e usuários desde que ainda não exista nenhum funcionário e nenhum usuário cadastrados. Se ele tentar acessar qualquer página diferente dessas duas será direcionado para a página de login.

**Listagem 48.** Arquivo faces-config.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<faces-config

xmlns=*"http://xmlns.jcp.org/xml/ns/javaee"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig\_2\_2.xsd"*

version=*"2.2"*>

<lifecycle>

<phase-listener>

br.com.easyjavamagazine.listeners.Autorizador

</phase-listener>

</lifecycle>

</faces-config>

Para que os métodos da classe Autorizador sejam realmente chamados antes e depois da fase Restore view, devemos registrá-la no arquivo faces-config.xml conforme **Listagem 48**.

# Conclusão

O artigo cobriu o desenvolvimento de uma aplicação utilizando JSF, JPA e CDI. O objetivo foi apresentar os principais recursos de cada um dessas especificações e também mostrar como as implementações dessas especificações podem trabalhar juntas para resolver diversos problemas e facilitar a vida do desenvolvedor.

É claro que não é uma regra usar as três coisas juntas, e podemos optar por usar somente uma ou duas delas dependendo do projeto. Podemos até combinar o uso de qualquer uma delas com outras tantas tecnologias Java, pois no mundo Java o que não falta são opções. Cabe ao desenvolver adquirir experiência e conhecimento para saber encaixar cada uma das peças do quebra cabeça de forma a entregar um software de qualidade e sempre usando as tecnologias mais adequadas para resolver os problemas peculiares de cada projeto.

# Links:

Site do Java Community Process

<https://jcp.org/en/home/index>

Wikipédia – explicação sobre Java Community Process

<http://pt.wikipedia.org/wiki/Java_Community_Process>

Site do Mockito

<http://mockito.org>

Endereço do Mockito no Github

<https://github.com/mockito/mockito>

Aplicação reduzida(igual a desenvolvida no artigo) no Github

<http://xxxxxxxxxxxxxxxxxxxxxxxxxxxx>

Aplicação completa no Github

<http://xxxxxxxxxxxxxxxxxxxxxxxxxxxx>

Download JDK 8

<http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html>

Download Eclipse Luna

<https://www.eclipse.org/downloads/>

Download Jboss WildFly

<http://wildfly.org/downloads/>

Download do instalador do MySQL

<http://dev.mysql.com/downloads/installer/>

Download do driver do MySQL

<http://dev.mysql.com/downloads/connector/j/>

Documentação MySQL Command-Line Tool

<http://dev.mysql.com/doc/refman/5.0/en/mysql.html>

Post “Associações com mappedBy no Hibernate”

<http://www.devmedia.com.br/associacoes-com-mappedby-no-hibernate/29425>