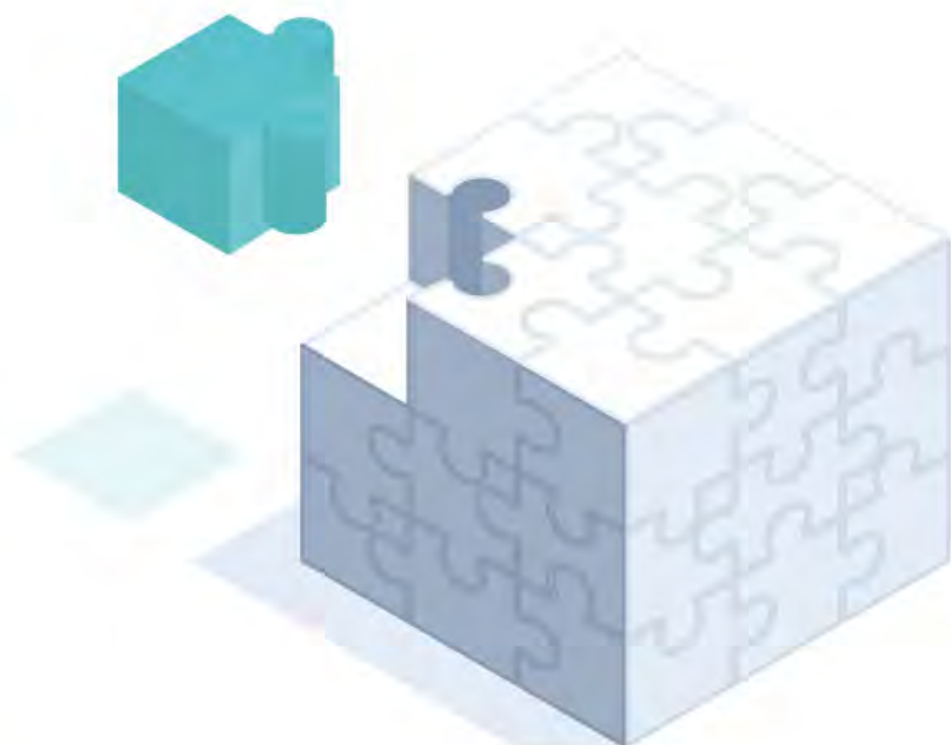


Java EE

Aproveite toda a plataforma para
construir aplicações



Casa do
Código

—  —
SÉRIE CAELUM

ALBERTO SOUZA

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

Agradecimentos

Este é o segundo livro que escrevo pela Casa do Código e a satisfação é imensa. Ser um dos autores da editora que desburocratizou a criação de livros no país é uma enorme honra. Por conta disso, meu primeiro agradecimento vai novamente para Paulo e Adriano, por me darem espaço para escrever.

Vou aproveitar para também deixar os agradecimentos aos meus pais, dona Sebastiana e seu Alberto! Sempre me incentivaram a estudar e, deixaram bem claro para mim, que a única coisa que ninguém pode tirar de você é o conhecimento adquirido. Acho que nunca vou conseguir demonstrar o quanto isso foi relevante.

Por último, quero agradecer a você, que tem sede de conhecimento e quer estar sempre aprendendo. São de pessoas assim que o mundo precisa e eu espero, sinceramente, que cada gota de suor direcionado para seu aprendizado se transforme em um passo a mais dado na sua carreira.

Autor

Alberto Souza é Bacharel em Ciência da Computação pela Universidade Salvador e desenvolvedor desde 2005, tendo participado de muitos projetos web e experimentado diversas linguagens. Participa de projetos open source como o Stella e VRaptor. Possui a certificação SCJP e trabalha como desenvolvedor e instrutor pela Caelum. Seu Twitter é @alberto_souza e você também pode encontrá-lo no GitHub, no endereço <http://www.github.com/asouza>. Também é um dos fundadores do projeto SetupMyProject, uma aplicação web que vem ajudando diversos desenvolvedores a criarem projetos JAVA de maneira bastante simples.

Além de já ter escrito outros livros, *Play Framework na prática: Gaste tempo no que é precioso* e o *Spring MVC: Domine o principal framework web Java*, ele também possui dois blogs sobre tecnologia. Um específico sobre o Spring (<http://domineospring.wordpress.com>) e outro que ele desvenda os detalhes por trás de várias tecnologias que são usadas nos projetos (<http://setupmyproject.wordpress.com>) .

Público-alvo

Este livro é para pessoas que já conheçam um pouco sobre a estrutura de um projeto web utilizando a linguagem JAVA. O mínimo de JSF vai ajudar para que você consiga acompanhar a construção da aplicação WEB de uma maneira mais fluida, mas fique tranquilo, pois tudo do JSF será explicado para você. Quase todo capítulo vai cobrir detalhes que vão além do uso normal e vai deixá-lo mais crítico em relação ao uso da tecnologia.

Considere também que o livro é sobre o JAVA EE como plataforma, então existem várias outras especificações que serão cobertas e que lhe deixará mais embasado para tomar suas futuras decisões em relação à plataforma.

Sumário

1	Introdução	1
1.1	Por que o Java EE?	2
1.2	Spring x Java EE	4
1.3	Comece a aventura	4
1.4	Público-alvo	5
1.5	Curiosidade: por que JSF como framework?	5
1.6	Vamos lá?	6
2	Começando o projeto	7
2.1	Configuração e criação do projeto	7
2.2	Acessando o primeiro endereço	25
2.3	Um pouco por dentro do framework	29
2.4	Conclusão	32
3	Cadastro de produtos	33
3.1	Formulário de cadastro	33
3.2	Lógica de cadastro	35
3.3	Gravando os dados no banco de dados	38
3.4	Configurando a JPA	42
3.5	Configuração do DataSource no WildFly	46
3.6	A necessidade de uma transação	54
3.7	Conclusão	56

4	Melhorando o cadastro e um pouco mais de JSF	59
4.1	Associando vários autores	60
4.2	Limpando o formulário	67
4.3	Listando os livros	68
4.4	Forward x Redirect	70
4.5	Exibindo a mensagem de sucesso	73
4.6	Isolando o código de infraestrutura	76
4.7	Conclusão	81
5	Validação e conversão de dados	83
5.1	Validação básica	83
5.2	Exibindo as mensagens de erro de maneira amigável	87
5.3	Trocando as mensagens default do JSF	90
5.4	Integração com a Bean Validation	92
5.5	Convertendo a data	96
5.6	Converter para entidades	101
5.7	Conclusão	107
6	Upload de arquivos	109
6.1	Recebendo o arquivo no Managed Bean	109
6.2	Salvando o caminho do arquivo	114
6.3	Gravando os arquivos fora do servidor web	117
6.4	Conclusão	125
7	Carrinho de compras	127
7.1	Exibindo os livros na página inicial	127
7.2	Navegando para o detalhe do produto	131
7.3	Lidando com LazyInitializationException	139
7.4	Será que o melhor é o DAO ser um EJB stateful?	143
7.5	Formatando a data de publicação	145
7.6	Carrinho de compras e o escopo de sessão	146
7.7	Conclusão	156

8	Fechamento da compra e processamento assíncrono	157
8.1	Implementando a tela de finalização	157
8.2	Gravando as informações do usuário	162
8.3	Validação seletiva com a BeanValidation	167
8.4	Salvando as informações do checkout	173
8.5	Integrando com outra aplicação	178
8.6	Executando operações demoradas assincronamente	183
8.7	JAX-RS para todas as outras requisições HTTP	187
8.8	Configurando o JAX-RS	193
8.9	Curiosidade: usando um ExecutorService gerenciado	194
8.10	Conclusão	196
9	Melhorando a performance com cache	197
9.1	Cacheando o retorno das consultas na JPA	197
9.2	Provider de cache e suas configurações	204
9.3	Invalidação do cache por alteração	205
9.4	Cacheando trechos da página	206
9.5	Conclusão	209
10	Respondendo mais de um formato	211
10.1	Expondo os dados em outros formatos	211
10.2	Content negotiation	219
10.3	Simulando um cliente para o nosso serviço	220
10.4	Conclusão	223
11	Mais de processamento assíncrono com JMS	225
11.1	Enviando o e-mail de finalização	225
11.2	Um pouco mais sobre processamento assíncrono	232
11.3	Utilizando o JMS para mensageria	235
11.4	Registrando tratadores de mensagens com MDBs	239
11.5	Implementação do JMS utilizada pelo WildFly	245
11.6	Cautela no uso do código assíncrono	245
11.7	Conclusão	246

12	Protegendo a aplicação	247
12.1	Definindo as regras de acesso com o JAAS	248
12.2	Configurando o formulário de login	250
12.3	Configurando o LoginModule do JAAS	254
12.4	Exibindo o usuário logado e escondendo trechos da página	260
12.5	Conclusão	267
13	Organização do layout em templates	269
13.1	Templates	271
13.2	Conclusão	276
14	Internacionalização	277
14.1	Isolando os textos em arquivos de mensagens	278
14.2	Accept-Language header	281
14.3	Passando parâmetros nas mensagens	282
14.4	Deixe que o usuário defina a língua	283
14.5	Conclusão	288
15	Enviando e recebendo informações via WebSocket	289
15.1	Como notificar os usuários?	289
15.2	API de WebSockets e o navegador	290
15.3	Enviando mensagens a partir do servidor	293
15.4	Outros detalhes da especificação de WebSockets	302
15.5	Conclusão	303
16	Últimas considerações técnicas	305
16.1	Deploy	305
16.2	Profiles	309
16.3	Testes	309
16.4	Projetos paralelos que podem nos ajudar	309
16.5	Conclusão	310
17	Hora de praticar	311
17.1	Estudamos muitos assuntos!	312
17.2	Mantenha contato	312

CAPÍTULO 1

Introdução

Por muito tempo, a especificação Java EE (*Java Enterprise Edition*) ficou conhecida como sendo algo extremamente complexo de ser entendido. Caso você tenha trabalhado com as primeiras versões ou conhece alguém que trabalhou, deve se lembrar da complicação que era configurar cada uma das tecnologias, e também o quão trabalhoso era fazê-las funcionarem em conjunto.

Pois bem, essas lembranças ficaram em um passado distante. O surgimento de outros frameworks no mercado, notoriamente o Spring, fizeram com que as empresas responsáveis por evoluir a especificação se movimentassem.

Desde a versão 5, o Java EE vem evoluindo a passos largos tanto pelo lado da configuração das tecnologias envolvidas quanto pelo prisma da integração entre elas. Com a entrada do CDI (*Context Dependency Injection*) na versão 6, as integrações pararam de girar em torno dos EJBs (*Enterprise JavaBean*) e tudo ficou mais simples de ser trabalhado. A versão 7, que é a atual, deixou

tudo ainda mais fácil, e trouxe novas melhorias no CDI, além de também deixar muito mais simples a integração com sistemas de mensageria. E esses são apenas dois dos exemplos.

Com toda essa desburocratização, as tecnologias envolvidas na especificação e, por consequência, os servidores que as implementam – como o Wildfly (antigo Jboss) –, alçaram novos voos. Hoje, aquela história que Java EE era só para empresas gigantes não é mais válida. Qualquer tipo de aplicação, dentro de qualquer empresa, pode tirar proveito da última versão da especificação.

A partir de agora, tudo o que eu comentar é relativo à versão 7 do Java EE. Se em algum momento eu precisar referenciar outra versão, você será devidamente avisado.

1.1 POR QUE O JAVA EE?

Quase todas as aplicações possuem certas funcionalidades que exigem um certo conjunto de tecnologias, que vivem se repetindo entre elas. Alguns exemplos mais comuns são:

- Mandar e-mail;
- Gravar dados no banco de dados;
- Fazer a segurança da aplicação;
- Expor a aplicação na web.

Podemos também pensar em tecnologias necessárias para outras funcionalidades, vistas mais comumente em aplicações maiores:

- Integração via mensageria;
- Integração via REST;
- WebSockets para alertar usuários sobre algo;
- Realização de tarefas assíncronas.

Você provavelmente já caiu (ou vai cair) em uma aplicação que demande a integração de muitas das coisas que acabamos de listar. E é nesse tipo de cenário que possuir um servidor que já traz todas essas implementações decididas pode ajudar bastante.

Dê uma olhada no conjunto de especificações que compõem o Java EE 7.

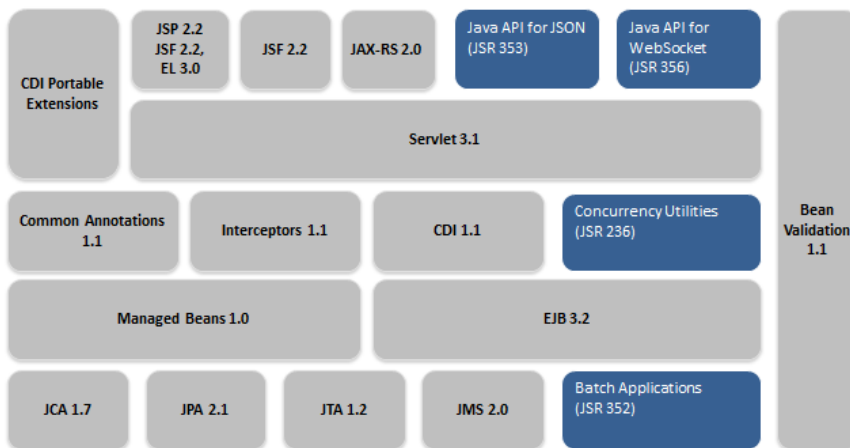


Fig. 1.1: Especificações do Java EE

Perceba que são muitas coisas, e tudo isso é entregue pronto para você. Não vamos ser mentirosos e dizer que tudo vai funcionar lindamente sem nenhuma gota de suor, mas, hoje em dia, o trabalho para configurar e tirar proveito de cada uma delas está muito menor. Você ficará surpreso com a facilidade com que quase todas nossas funcionalidades serão implementadas!

Baseado nessa evolução, escolher um servidor Java EE para sua próxima aplicação é uma consideração que se faz necessária, mesmo sabendo que a competição continua acirrada, já que os competidores, como o Spring, não pararam de evoluir também.

Um outro ponto altamente relevante é o quanto de ajuda você pode obter através da internet. Nesse ponto, os projetos relacionados ao Java EE, como o JSF (*JavaServer Faces*), também se beneficiam bastante.

1.2 SPRING X JAVA EE

Essa é uma outra discussão que quase sempre acaba aparecendo. Há muito tempo, nas primeiras versões do Java EE, não tinha comparação, o ecossistema do Spring era realmente uma alternativa melhor. Nem era muito simples de configurar como é hoje, mas, mesmo assim, era muito mais fácil do que qualquer coisa necessária para o Java EE.

Nos dias de hoje, as duas frentes estão muito fortes. Ambas possuem implementações que se destacam frente à concorrente e, com isso, a decisão de seguir por um lado ou pelo outro deve ser tomada com base no que o seu projeto precisa.

Minha função com este livro é fornecer uma visão pragmática sobre o Java EE. Não esconderei nada e você mesmo será capaz de avaliar qual conjunto de tecnologias é melhor para a sua empresa ou para o próximo projeto. Não é à toa que eu também já escrevi o livro *Spring MVC: Domine o principal framework web Java*, pela Casa do Código. Melhor do que nos apaixonarmos pela tecnologia é sabermos tirar proveito delas em função do cenário que se apresenta.

1.3 COMECE A AVENTURA

Durante o livro, vamos construir uma aplicação muito parecida com o próprio site da Casa do Código. Desenvolveremos funcionalidades que nos levarão ao uso de diversas das especificações do Java EE, como:

- *Bean validation* para validarmos os nossos dados;
- Processamento assíncrono via JMS;
- JSF 2.2 e algumas das suas novidades, como a integração com o HTML 5;
- WebSockets para notificarmos usuários de promoções relâmpagos;
- CDI para isolarmos a criação de componentes e ajudarmos na composição de regras de negócios;

- JAX-RS para possibilitarmos integrações via REST;
- JPA para acesso a banco de dados, além de sua API de cache;
- Um pouco sobre EJBs e um caso de uso bem interessante;
- JAAS para adicionarmos autenticação e autorização;
- Java Mail para mandarmos e-mails de finalização de compra.
- JTA para controlarmos as transações.
- Um pouco sobre a Concurrency Utilities API para termos Threads gerenciadas pelo container.
- O suficiente sobre a JNDI para conseguirmos buscar os objetos necessários dentro do servidor.

1.4 PÚBLICO-ALVO

O público seria as pessoas que já conhecem um pouco sobre a estrutura de um projeto baseado no Design Pattern MVC, através do JSF. O mínimo de JSF ajudará você a conseguir acompanhar a construção da aplicação WEB.

Mas fique tranquilo, pois tudo além do básico do JSF será explicado para você. Quase todo capítulo vai cobrir detalhes que vão além do uso normal e vai deixá-lo mais crítico em relação ao uso da tecnologia.

1.5 CURIOSIDADE: POR QUE JSF COMO FRAMEWORK?

A nossa aplicação web vai ser baseada na Casa do Código e, talvez, alguns de vocês pensem: por que será usado o JSF? Ele não é mais comumente usado em sistemas web que, na verdade, funcionam quase como sistemas desktop? Sim, isso é verdade. Em contrapartida, o JSF 2.2 evoluiu em relação ao controle do HTML gerado, dando-nos a possibilidade de construir páginas mais condizentes com as necessidades atuais.

Além disso, durante o livro, será usada uma abordagem que deixará o JSF mais próximo dos frameworks **Action Based** com que somos acostumados,

mesmo ele sendo um framework muito conhecido por ser baseado em componentes, o famoso **Component Based**.

Complementando, ele é uma das especificações oficiais do Java EE e, como no livro vamos discutir a especificação, nada mais justo que usarmos a tecnologia indicada por ela.

NÃO FIQUE PERDIDO!

Todo o código-fonte do livro está disponível no GitHub, basta acessar o endereço <https://github.com/asouza/casadocodigojavaee>.

Fique à vontade para navegar pelos arquivos do projeto. Os *commits* foram divididos de acordo com os capítulos do livro, justamente para que você possa acessar o código compatível com o capítulo que esteja lendo.

Além disso, foi criado um grupo de discussão, no qual você pode postar todas as suas dúvidas referentes à leitura. Para acessá-lo, basta seguir este link: <https://groups.google.com/forum/#!forum/livro-javaee>.

1.6 VAMOS LÁ?

Fique junto comigo e leia cada capítulo com calma; eu sempre lhe direi se é hora de dar uma pausa ou de já pular para o próximo. Espero que você tenha uma grande jornada!

CAPÍTULO 2

Começando o projeto

2.1 CONFIGURAÇÃO E CRIAÇÃO DO PROJETO

O objetivo do livro é construir uma aplicação semelhante à da própria Casa do Código. Por ser um site de *e-commerce*, a aplicação nos fornece a chance de implementar diversas funcionalidades. Além disso, é um domínio que não vai nos causar muitas dúvidas.

No final do livro, a nossa aplicação terá a seguinte cara:



Fig. 2.1: Site da Casa do Código

A primeira coisa que precisamos fazer é justamente criar e configurar o mínimo necessário para ter nossa aplicação, rodando corretamente com o JSF. Como vamos usar o Maven para gerenciar as nossas dependências, somos obrigados a criar o projeto seguindo uma estrutura determinada por essa ferramenta.

O layout das pastas, como de costume, vai seguir o seguinte formato:

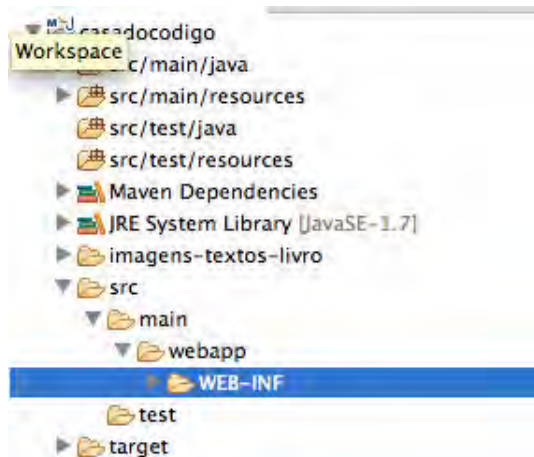


Fig. 2.2: Estrutura das pastas

Provavelmente, você já deve ter feito esta atividade algumas vezes, então, para mudar um pouco a sua rotina, vou utilizar um modo um pouco diferente de criar o projeto. Usaremos um projeto da Red Hat, chamado **Jboss Forge**, que pode ser baixado em <http://forge.jboss.org/download>.

O Jboss Forge fornece vários comandos prontos para a criação de projetos. Após baixado, extraia o arquivo em uma pasta de sua preferência. Acesse o terminal do seu computador e digite o seguinte comando:

```
$caminhoParaSuaInstalacao/bin/forg
```

Isso abrirá uma aplicação no próprio console.

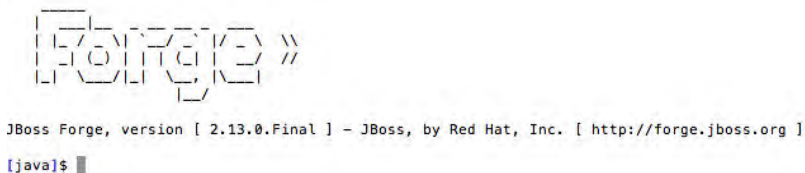


Fig. 2.3: Console do Forge

Agora, basta que digitemos o comando responsável pela criação do projeto.

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

Pronto! Isso é suficiente para você criar o seu projeto baseado no Maven. A estrutura de pastas já está pronta, só precisamos realizar as devidas configurações.

Dependências necessárias

Precisamos adicionar todas as dependências necessárias para que seja possível começarmos a trabalhar no projeto. Essa já é uma das partes bem interessantes de trabalhar em um projeto fortemente baseado nas especificações do Java EE.

Como quase todas as dependências importantes já vêm prontas dentro do servidor que vamos usar, nós quase não precisamos alterar o `pom.xml`. Para não ficarmos perdendo tempo realizando essas configurações, vamos mais uma vez utilizar o console do **Forge**.

Entre na pasta criada para o seu projeto e rode o Forge mais uma vez. De dentro do console, escreva o seguinte comando:

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

Depois da execução, deve aparecer uma mensagem de sucesso, parecida com essa:

```
**SUCCESS** JavaServer Faces has been installed.
```

Sem esforço nenhum, acabamos de criar tudo o que é necessário para ter nosso projeto com JSF rodando.

Como não queremos usar configurações das quais não temos conhecimento, vamos passear pelas alterações realizadas pelo comando. Um primeiro arquivo que podemos olhar é o `pom.xml`, responsável por gerenciar nossas dependências.

```
<dependencyManagement>  
  <dependencies>
```

```
<dependency>
  <groupId>org.jboss.spec</groupId>
  <artifactId>jboss-javaee-6.0</artifactId>
  <version>3.0.2.Final</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>
<dependency>
  <groupId>javax.faces</groupId>
  <artifactId>javax.faces-api</artifactId>
  <version>2.2</version>
  <scope>provided</scope>
</dependency>
</dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>javax.faces</groupId>
    <artifactId>javax.faces-api</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

Aqui, as dependências foram geradas com a tag `dependencyManagement`. Esta geralmente é usada quando estamos nos preocupando em criar projetos que podem precisar das mesmas configurações de um certo projeto “pai”. Não é o nosso caso, mas também não interfere em nada no nosso trabalho, inclusive você pode até apagar o conteúdo dela. A única coisa em que temos de prestar atenção é que, quando for necessário adicionar novas `libs` manualmente, vamos sempre usar a tag `<dependency>` dentro da `<dependencies>`. Só o que está dentro de `<dependencies>` é realmente importado para o projeto.

Ainda em relação às dependências, perceba que ele adicionou a especificação do JSF com o escopo **provided**. Como vamos rodar nosso código dentro de um servidor de aplicação, no caso o Wildfly, todas essas bibliotecas já estarão disponíveis. Só as deixamos no `pom.xml`, porque vamos precisar usar as classes dentro do nosso projeto. Também, para que a IDE (*Integrated*

Development Environment) compile, ela precisará ter acesso aos `jars`, pelo menos em tempo de compilação.

Além desse arquivo, também foi gerado um outro, chamado `faces-config.xml`, criado na pasta `WEB-INF`. Mais para a frente, vamos usá-lo para realizar configurações relativas ao JSF; por enquanto, segure a ansiedade.

Habilitando o CDI

Como já comentamos, o CDI é a especificação que faz a cola entre as diversas especificações. Como não poderia ser diferente, vamos usá-la intensamente durante a construção de nosso projeto. Por conta disso, é interessante já deixar esse passo configurado.

De novo, vamos usar o Forge para gerar as configurações necessárias. Dentro do console, execute o seguinte comando:

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

Mais uma vez, o `pom.xml` foi alterado, agora para adicionar as dependências necessárias para o CDI, e mais uma vez elas foram configuradas como `provided`, já que estarão disponíveis dentro do nosso servidor de aplicações.

```
<dependencyManagement>
...
<dependency>
  <groupId>org.jboss.spec</groupId>
  <artifactId>jboss-javaee-6.0</artifactId>
  <version>3.0.2.Final</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>
<dependency>
  <groupId>javax.enterprise</groupId>
  <artifactId>cdi-api</artifactId>
  <version>1.1</version>
  <scope>provided</scope>
</dependency>
...
```



```
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>javax.enterprise</groupId>
    <artifactId>cdi-api</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

Além disso, dentro da pasta `WEB-INF`, foi criado mais um arquivo, dessa vez o `beans.xml`.

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

Este é o arquivo necessário para habilitar o funcionamento do CDI no seu projeto. Preste atenção no atributo `bean-discovery-mode`, ele está configurado com o valor `all`. Isso quer dizer que toda classe criada dentro do nosso projeto será carregada e controlada pela implementação do CDI que será usada. Você será lembrado disso quando começarmos realmente a produzir o código.

Perceba que o uso do Forge nos poupou um pouco de trabalho de ter de ficar criando os arquivos e colocando-os nos seus devidos lugares. É uma ferramenta muito boa, e você pode automatizar diversos *setups* dos seus projetos.

Importando o projeto para a IDE

Como estamos usando o Wildfly e escrevendo um projeto baseado no Java EE, vou sugerir que você tenha mais uma experiência diferente e use o **JBoss Developer Studio**. Ele é uma IDE baseada no Eclipse, e customizada pela Red Hat para atender quem deseja trabalhar mais intensamente com o

Java EE. Caso opte por ela, o download pode ser realizado em <http://www.jboss.org/products/devstudio/download/>.

Diferente do Eclipse tradicional, o Developer Studio traz um *wizard* de instalação. Basta seguir os passos sugeridos por ele.

Veja algumas telas que são importantes:

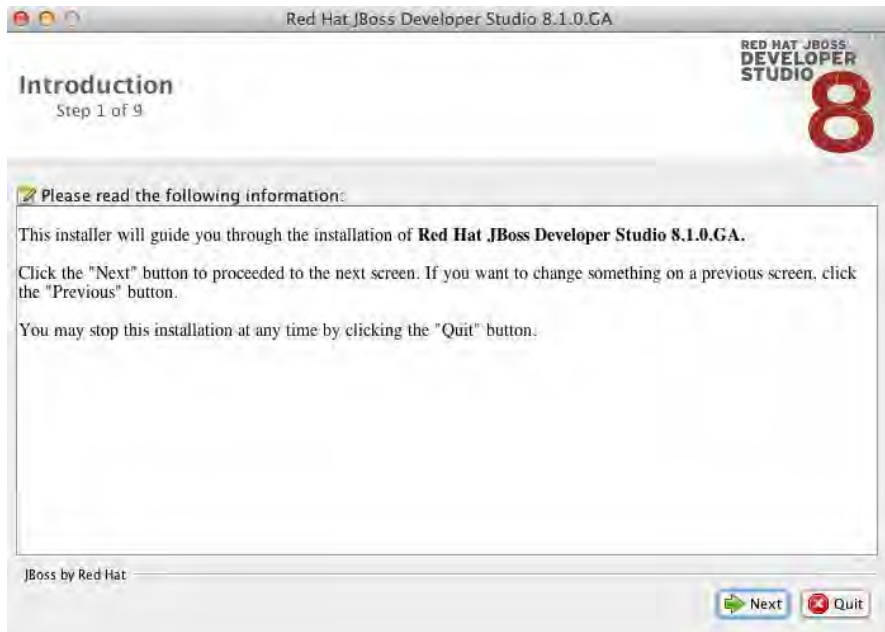


Fig. 2.4: Primeiro passo da instalação

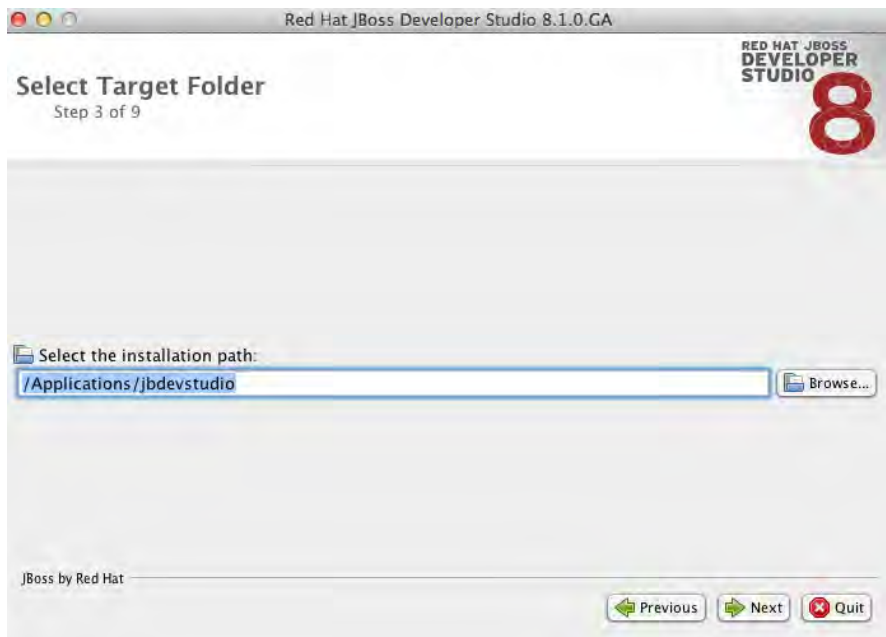


Fig. 2.5: Escolha sua pasta de instalação



Fig. 2.6: Outros plugins da Red Hat não são necessários

Vamos importar o projeto como um `Maven Project`; assim, para cada nova dependência necessária, basta alterarmos o `pom.xml`, e a própria IDE se encarregará de atualizar o nosso `classpath`.

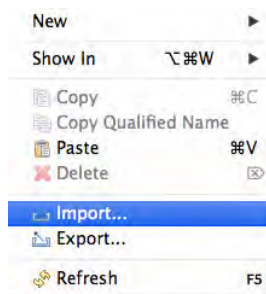


Fig. 2.7: Selecione a opção import

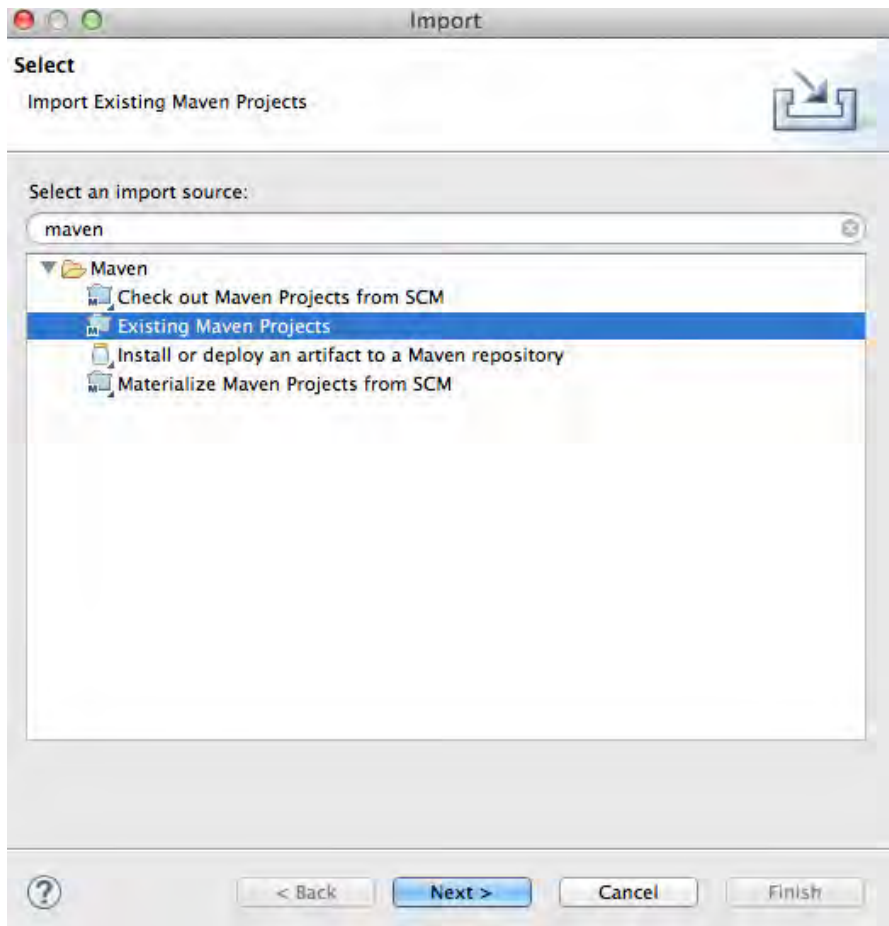


Fig. 2.8: Escolha o Maven Project

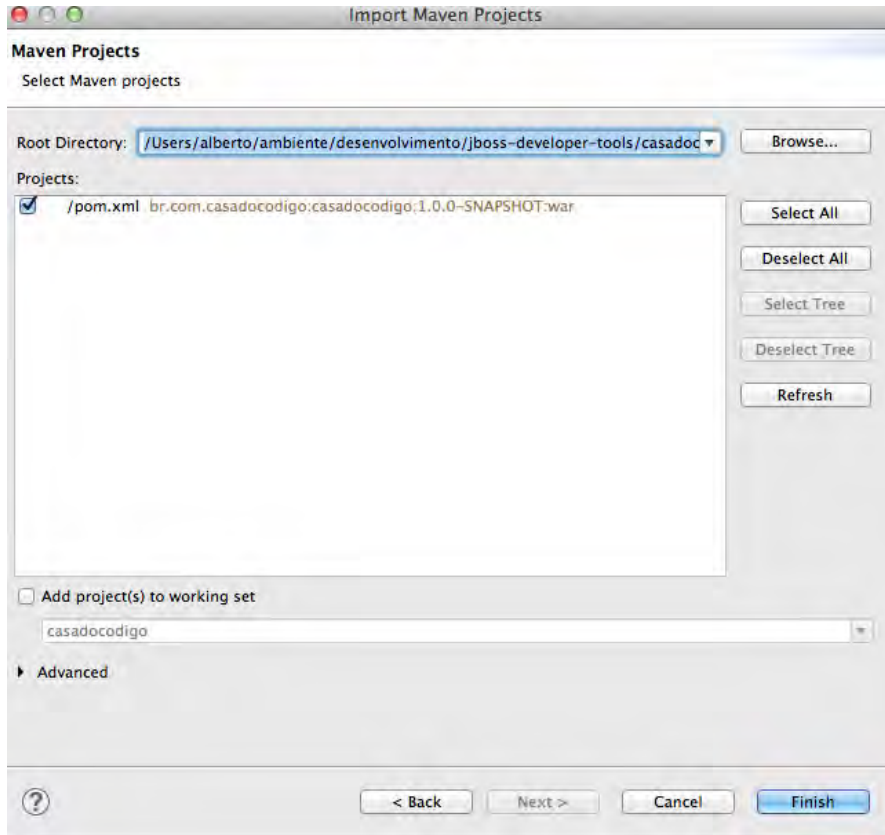


Fig. 2.9: Aponte para o endereço onde seu projeto foi criado

Pronto, agora temos nosso projeto importado. O último passo é adicioná-lo a um servidor, para que possamos realizar nossos testes e ver como a aplicação vai andando.

Para isso, vamos seguir o caminho padrão de criação de servidores dentro do Eclipse. Apenas para nos localizarmos, os próximos passos deverão ser realizados na aba `Servers` do seu Eclipse.

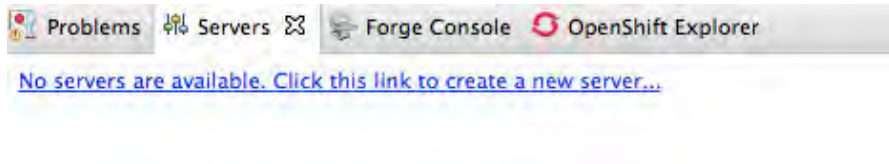


Fig. 2.10: Clique no link de criação de servidores

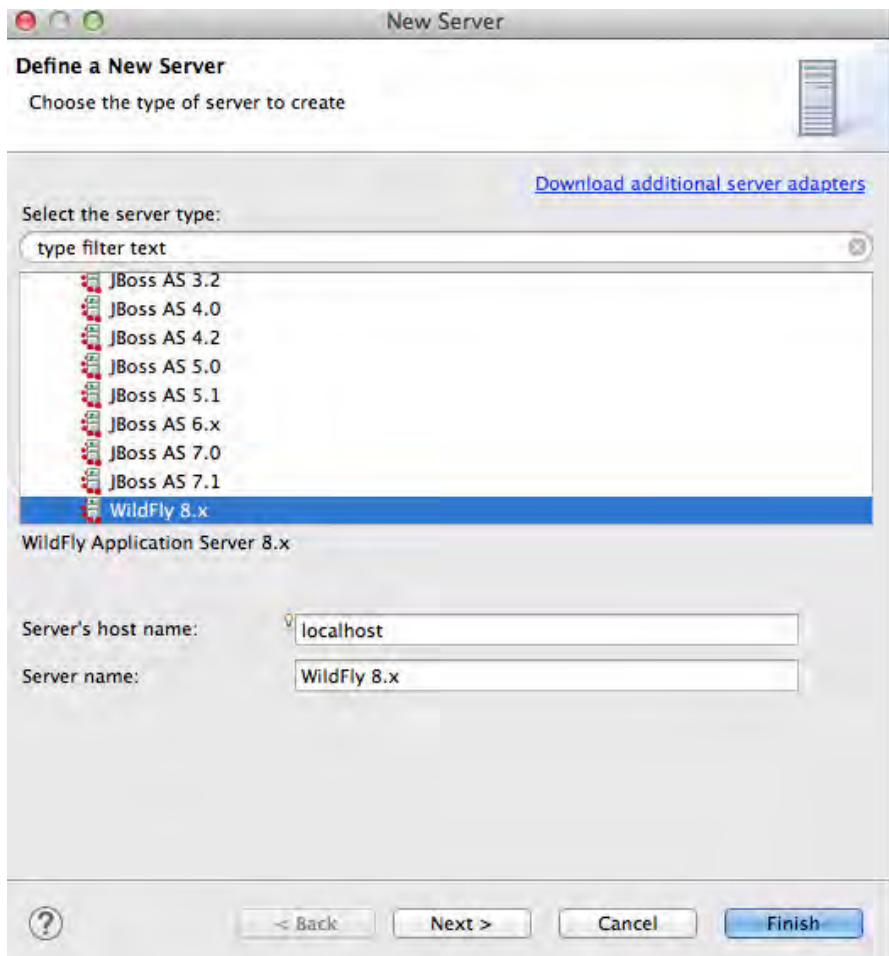


Fig. 2.11: Selecione o WildFly, e clique no menu Jboss Community

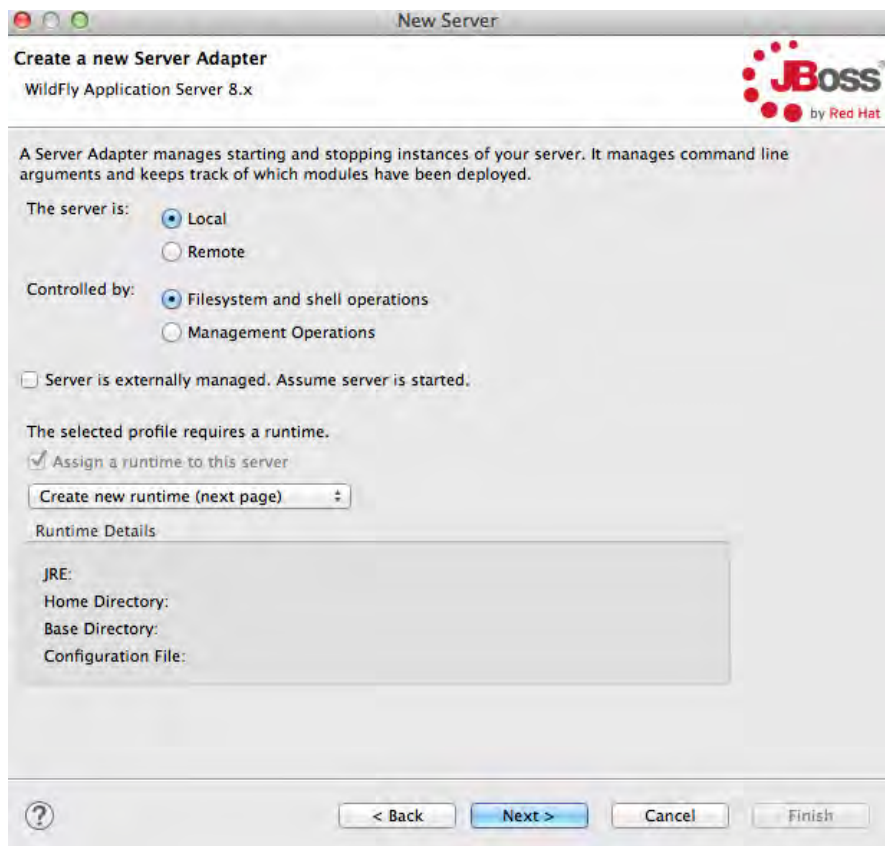


Fig. 2.12: Deixe tudo no valor default

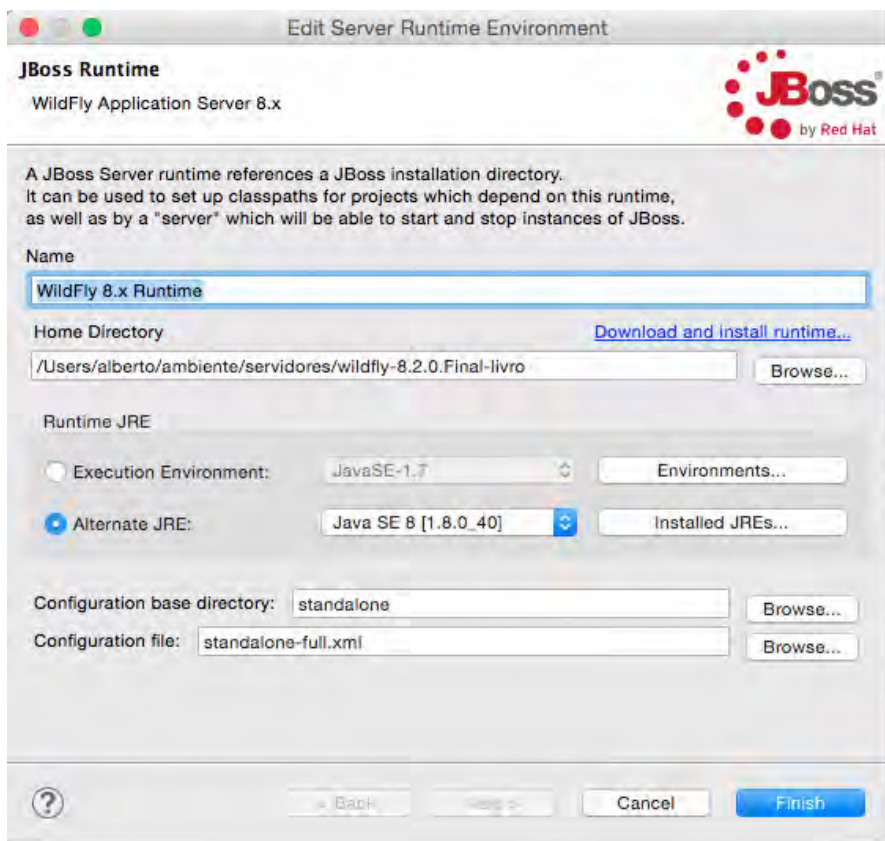


Fig. 2.13: Aponte para o caminho de instalação e use o standalone-full.xml

Para que esse passo seja possível, você deve realizar o download do Wildfly, que pode ser encontrado em <http://wildfly.org/downloads/>. Pegue a versão **8.2.0.Final**.

VERSÕES DO WILDFLY

A partir da versão 8.x, o WildFly começou a implementar a especificação do Java EE 7. Durante o livro, vamos usar a própria versão 8, mas fique a vontade para testar novas versões que também implementam a especificação do 7. Tudo deve funcionar normalmente.

Além disso, o wizard pede que nós utilizemos o arquivo de configuração chamado `standalone.xml`. O problema é que ele não carrega todos os módulos da especificação que serão necessários para o desenvolvimento da aplicação. Por exemplo, mesmo não sendo importante agora, mais à frente no livro será usado um recurso chamado de mensageria, que não está configurado neste arquivo.

Para evitar ter de começar alterando um arquivo e depois ter de utilizar outro, já troque essa opção pelo `standalone-full.xml`. Por fim, adicione o projeto no servidor.

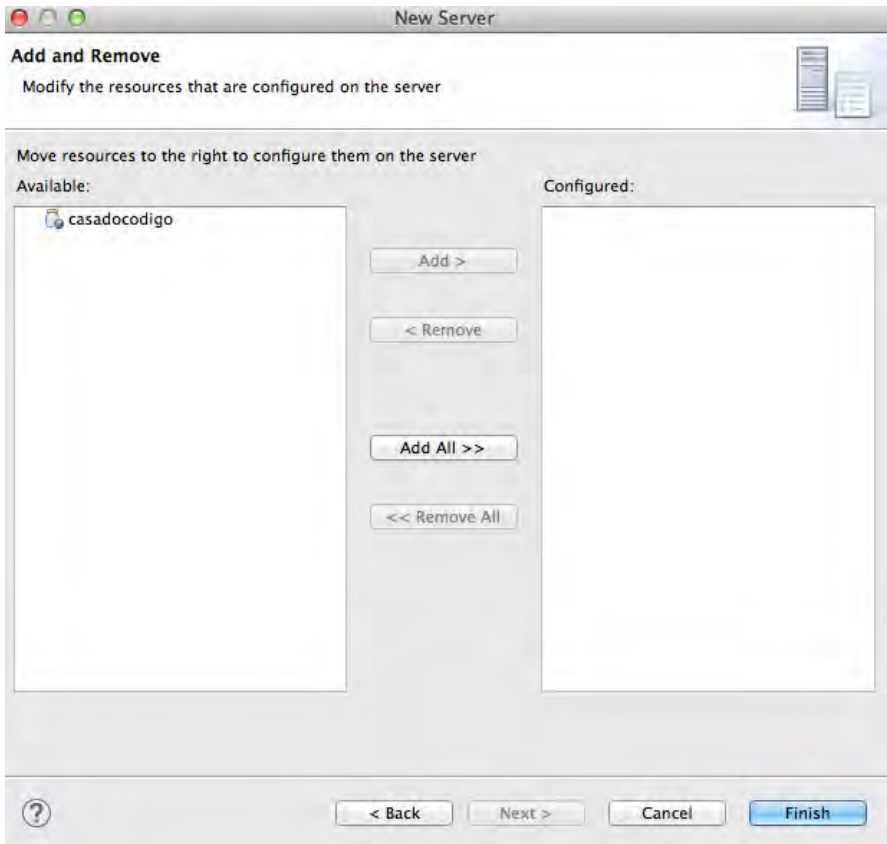


Fig. 2.14: Adicione o projeto no servidor

Associe um servidor ao projeto

Vamos precisar das APIs do Java EE que estão disponíveis dentro do servidor. Até agora, temos usado o Forge para configurar e adicionar as dependências necessárias dentro do nosso `pom.xml`, para que possamos compilar nosso código. Isso é útil apenas quando outros arquivos precisam ser gerados.

Durante vários momentos, vamos precisar apenas ter acesso às interfaces da especificação. Para garantirmos que só vamos usar o Forge quando realmente for necessário, vamos adicionar as `libs` disponíveis no Wildfly dentro do nosso projeto.

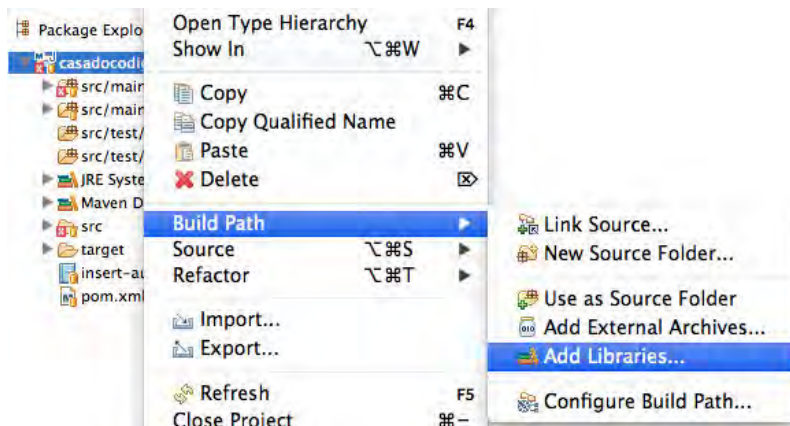


Fig. 2.15: Menu de alteração do Build Path

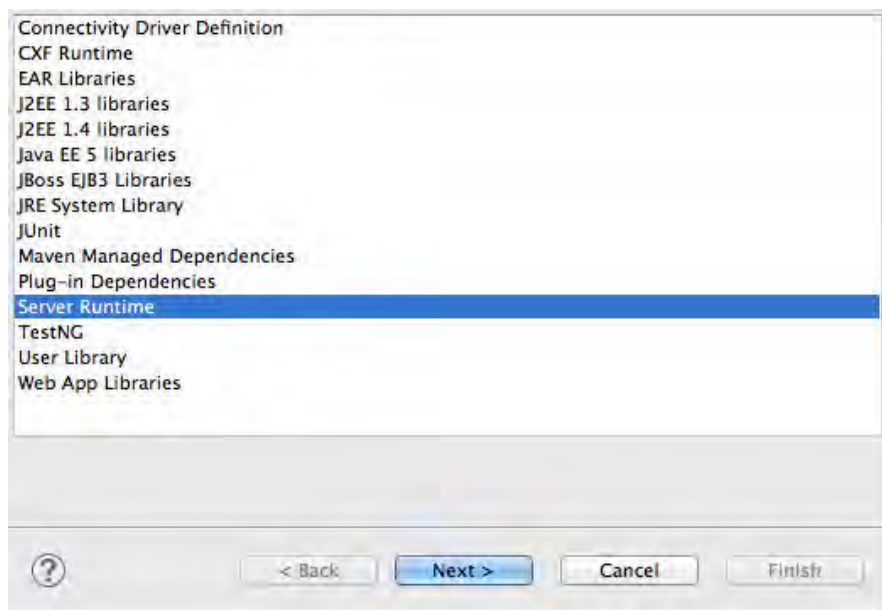


Fig. 2.16: Server Runtime libraries

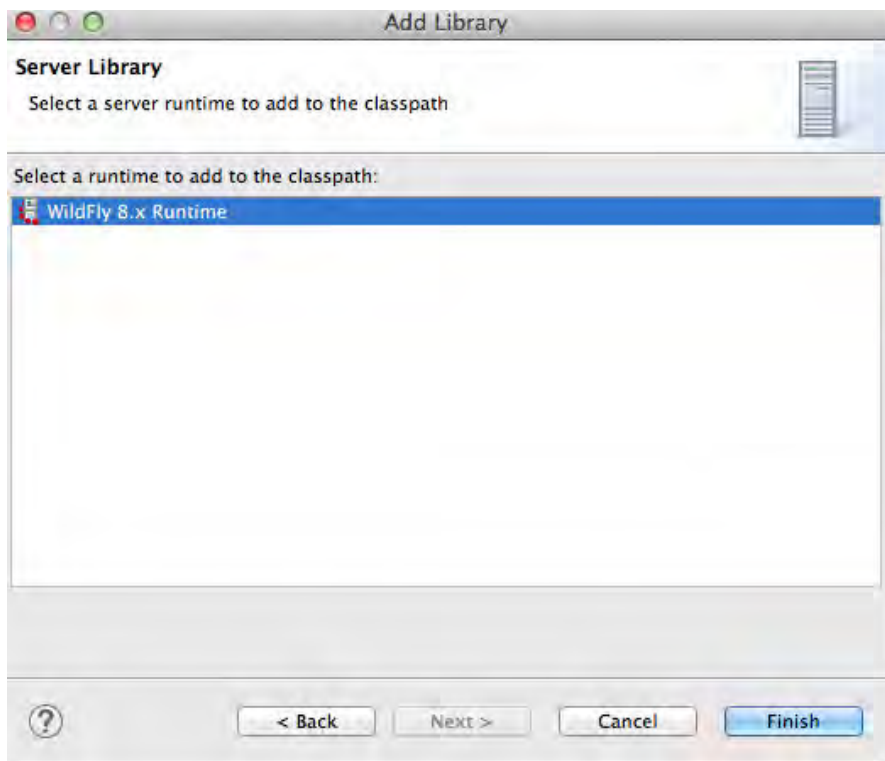


Fig. 2.17: Selecione o WildFly

2.2 ACESSANDO O PRIMEIRO ENDEREÇO

Com tudo configurado, chegou a hora de realmente começarmos a produzir alguma coisa! O nosso primeiro desafio é levar o usuário para a tela de cadastro de livros da Casa do Código. O endereço que queremos que ele digite é o `/produtos/form.xhtml`.

Neste momento, caso o usuário digite <http://localhost:8080/casadocodigo/produtos/form.xhtml>, ele receberá um erro do servidor indicando que esse endereço não pode ser atendido por ninguém.

Então, por mais óbvio que isso pareça, precisamos criar uma página que fique acessível pelo navegador. Podemos criar uma pasta chamada `produtos` dentro de `WebContent` e, lá dentro, criamos o arquivo chamado

form.xhtml.

```
<html xmlns:h="http://java.sun.com/jsf/html">
  <h:body>
    <h:form>
      <div>
        <h:outputLabel>Titulo</h:outputLabel>
        <h:inputText/>
      </div>
      <div>
        <h:outputLabel>Descrição</h:outputLabel>
        <h:inputTextarea cols="20" rows="10"/>
      </div>
      <div>
        <h:outputLabel>Número de páginas</h:outputLabel>
        <h:inputText/>
      </div>
      <div>
        <h:outputLabel>Preço</h:outputLabel>
        <h:inputText/>
      </div>
      <h:commandButton value=" Gravar"/>
    </h:form>
  </h:body>
</html>
```

Aqui, já usamos algumas tags do JSF que foram importadas por meio do *namespace* <http://java.sun.com/jsf/html>. Agora, caso o mesmo endereço seja acessado, recebemos uma resposta estranha.

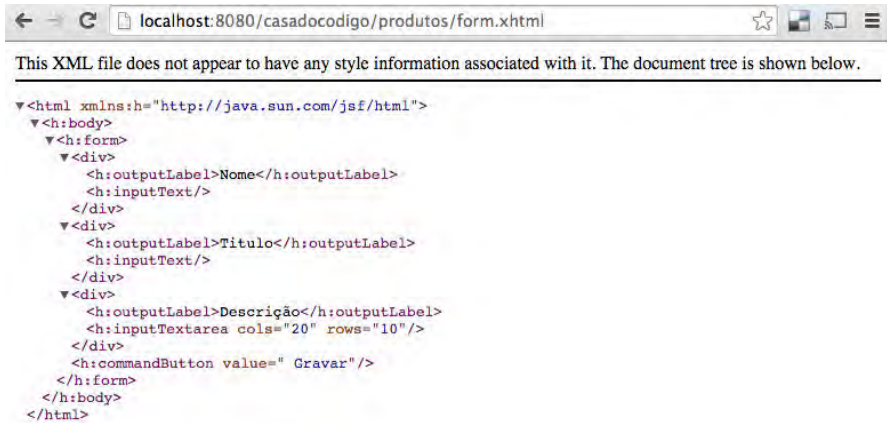


Fig. 2.18: Navegador interpretou um XML comum

Perceba que as tags que usamos do JSF nem foram interpretadas. Como pedimos por um endereço que termina com `xhtml`, o Wildfly entendeu que realmente está sendo solicitado um simples documento XML, e não uma página que precisa ser processada para a geração de HTML.

Vamos fazer uma simples alteração. Em vez de terminar com `xhtml`, vamos fazer com que o endereço termine com `.jsf`. E agora, como um passe de mágica, foi exibida a página corretamente. A pergunta intrigante é: *como isso aconteceu?*

Já está tudo configurado!

O Wildfly já possui a implementação do JSF; no caso, está sendo usada a Mojarra sem nós precisarmos fazer nada, nem adicionar um `web.xml`. O `servlet` do JSF já foi configurado.

Um outro bom questionamento é: *só porque estou dentro de um servidor de aplicação, quer dizer que vou usar o JSF como framework MVC?* Perceba que essa seria uma suposição um tanto quanto presunçosa dos servidores. Na verdade, meio que sem saber, informamos ao servidor que nosso projeto gostaria de habilitar o JSF.

Lembra do arquivo `faces-config.xml`? Quando um servidor de aplicação o encontra na pasta `WEB-INF`, automaticamente assume que a aplicação quer usar JSF, e o seu `servlet` é configurado.

Voltando à extensão `.jsf`, ela é apenas uma das opções de `url-pattern` criadas por padrão.

- `/faces` – <http://localhost:8080/casadocodigo/faces/produtos/form.xhtml>
- `*.jsf` – <http://localhost:8080/casadocodigo/produtos/form.jsf>
- `*.faces` – <http://localhost:8080/casadocodigo/produtos/form.faces>

Perceba que sempre temos de passar o caminho para o arquivo, o que muda são as maneiras como passamos.

O engraçado é que, por mais que essas sejam as convenções, nenhuma delas é usada pela maioria das empresas no mercado. É muito comum encontrar o `servlet` do JSF configurado para o `url-pattern` `*.xhtml`. Para ficarmos próximos do que você vai encontrar no seu próximo emprego, vamos realizar essa alteração.

Primeiro, é necessário que o arquivo `web.xml` seja gerado. Para isso, vamos utilizar o próprio Eclipse. Clique com o botão direito em cima do nome do projeto, e escolha a opção `Properties`.

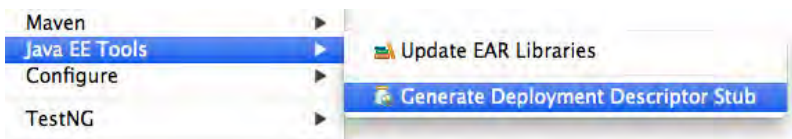


Fig. 2.19: Gerador de `web.xml`

No arquivo gerado, adicione a nova configuração do `Servlet`.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
  <display-name>casadocodigo</display-name>
```



```
<servlet>
    <servlet-name>jsf</servlet-name>
    <servlet-class>
        javax.faces.webapp.FacesServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>jsf</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>

</web-app>
```

Pronto, agora você pode acessar o endereço previamente combinado, que tudo funcionará normalmente.

2.3 UM POUCO POR DENTRO DO FRAMEWORK

Um ponto que muitas vezes diferencia “usuários” do framework de “entendores” é justamente ter o conhecimento do que acontece por dentro da ferramenta. Por exemplo, como o JSF registra o `Servlet` dele automaticamente? Foi explicado que ele analisa a existência do arquivo `faces-config`, mas onde isso é feito? Onde está o código de carregamento da `FacesServlet`?

Essas são perguntas que, por mais que não tenham a ver com o seu código de negócio em si, podem ser importantes para entender o fluxo do framework.

O `web.xml` é desnecessário!

Desde a versão 3 da especificação, não é mais necessário ter o arquivo `web.xml` para realizar a configuração de `servlets` e filtros. Uma das opções é declarar um arquivo chamado `web-fragment` dentro da pasta `META-INF` e empacotar como um `jar`. O projeto que adicionar essa `lib` já vai ganhar de brinde as configurações contidas dentro do `fragment`.

Outra opção usada pelo JSF é a de registrar um `listener` que é chamado quando o servidor acaba de subir. Dentro do `jar` da implementação do JSF, existe um arquivo com o nome

`javax.servlet.ServletContainerInitializer`, localizado na pasta `META-INF/services`. Quando o servidor sobe, ele analisa esse arquivo e carrega qualquer classe que esteja configurada dentro dele. No caso do JSF, a classe é a `FacesInitializer`.

Agora vamos dar uma olhada, apenas por curiosidade, na implementação dessa classe.

```
public void onStartup(Set<Class<?>> classes,
    ServletContext servletContext)
    throws ServletException {

    if (shouldCheckMappings(classes, servletContext)) {
        InitFacesContext initFacesContext = new
            InitFacesContext(servletContext);
        if (null == initFacesContext) {
            throw new
ServletException("Unable to initialize Mojarra");
        }

        Map<String,? extends ServletRegistration> existing =
            servletContext.getServletRegistrations();
        for (ServletRegistration registration :
            existing.values()) {
            if (FACES_SERVLET_CLASS.
                equals(registration.getClassName())) {
                // FacesServlet has already been defined,
                // so we're not going to add
                // additional mappings;
                return;
            }
        }
        ServletRegistration reg =
            servletContext.addServlet("FacesServlet",
                "javax.faces.webapp.FacesServlet");
        reg.addMapping("/faces/*", "*.jsf", "*.faces");
        servletContext.setAttribute(RIConstants.
            FACES_INITIALIZER_MAPPINGS_ADDED,
            Boolean.TRUE);
    }
}
```

```
        ...

    }

}

private boolean shouldCheckMappings
    (Set<Class<?>> classes, ServletContext context) {

    if (classes != null && !classes.isEmpty()) {
        return true;
    }

    try {
        return
            (context.getResource("/WEB-INF/faces-config.xml")
             != null);
    } catch (MalformedURLException mue) {

    }

    return false;
}
```

Perceba que o método `shouldCheckMappings` verifica a existência do arquivo `faces-config.xml`.

O trecho que segue é o responsável por decidir se o `servlet` deve ser registrado automaticamente.

```
for (ServletRegistration registration : existing.values()) {
    if (FACES_SERVLET_CLASS.
        equals(registration.getClassName())) {
        // FacesServlet has already been defined, so we're
        // not going to add additional mappings;
        return;
    }
}

ServletRegistration reg =
    servletContext.addServlet("FacesServlet",
```

```
        "javax.faces.webapp.FacesServlet");  
reg.addMapping("/faces/*", "*.jsf", "*.faces");
```

Ele procura por um mapeamento do `servlet` e, caso não tenha, faz o registro programático. Por sinal, perceba que os `url-patterns` criados são justamente os que comentamos anteriormente.

POSSÍVEL PROBLEMA COM OS COOKIES

O JSF faz uso do escopo de *session* para implementar detalhes internos, como manter a árvore de componentes (vamos voltar a ela mais à frente). Caso você tenha outras aplicações no seu computador que também fazem uso da sessão, é bem provável que você acabe com vários cookies associados à chave `jsessionid` no seu navegador.

Nesse cenário, muitas vezes o JSF se perde, não conseguindo associar corretamente a requisição com a árvore de componentes. Para não passar por problemas desnecessários, minha sugestão é que você limpe os cookies associados ao `localhost` no seu navegador.

2.4 CONCLUSÃO

Este capítulo foi um pouco longo e, para ser bem sincero, até simples. O problema é que foi necessária muita explicação para um simples *hello world*. A parte boa dessa história é que você passou por uma das etapas mais difíceis no aprendizado de qualquer projeto: o *setup* inicial.

Agora, tudo tende a ser mais fluido! Por isso, eu indico que você continue a leitura. Nos próximos capítulos, já começaremos a acessar o banco de dados para cadastrar e listar os produtos da loja.

CAPÍTULO 3

Cadastro de produtos

Passada a fase de configuração, chegou a hora de começarmos a implementar algumas das funcionalidades da loja. Uma das *features* mais básicas, porém importante, é o cadastro de livros, já que sem livros não existem compras. O legal é que já começamos um pouco disso no capítulo anterior, então vamos apenas dar continuação.

3.1 FORMULÁRIO DE CADASTRO

Vamos reaproveitar o formulário que usamos como *hello world* do nosso projeto para começar a criar a tela de cadastro.

```
<html xmlns:h="http://java.sun.com/jsf/html"
      xmlns:jsf="http://xmlns.jcp.org/jsf">
<h:body>
```

```
<h:form>
  <div>
    <h:outputLabel>Título</h:outputLabel>
    <h:inputText/>
  </div>
  <div>
    <h:outputLabel>Descrição</h:outputLabel>
    <h:inputTextarea cols="20" rows="10"/>
  </div>
  <div>
    <h:outputLabel>Número de páginas</h:outputLabel>
    <h:inputText/>
  </div>
  <div>
    <h:outputLabel>Preço</h:outputLabel>
    <h:inputText/>
  </div>
  <h:commandButton value=" Gravar"/>
</h:form>
</h:body>
</html>
```

Não tem nada de mais, apenas um HTML normal com algumas tags do JSF. Só para constar, nosso modelo de livro não precisa ser muito complexo. Ele terá apenas um título, descrição, número total de páginas e preço, pelo menos por enquanto.

Um detalhe importante que ficou faltando: para onde vamos enviar as informações cadastradas? Ainda não criamos nada que trate essa lógica. Como o JSF tenta trazer para o mundo web uma programação parecida com o mundo desktop, temos de adicionar um método que deve ser chamado após o formulário ser submetido.

Para começarmos a implementação dessa lógica, vamos alterar o nosso `commandButton` para que ele referencie o método que deve ser invocado.

```
.
.
.
<h:commandButton value="Gravar" action="#{adminBooksBean.save}"/>
```

3.2 LÓGICA DE CADASTRO

Agora, se você tentar cadastrar um novo livro, vai receber a seguinte *exception*:

```
javax.servlet.ServletException:
    javax.el.PropertyNotFoundException:
    /produtos/form.xhtml @17,69 action="#{adminBooksBean.save}":
    Target Unreachable, identifier 'adminBooksBean' resolved to null
```

Ela indica que a *expression language* não conseguiu resolver a referência para a variável `adminBooksBean`. Essa será justamente a primeira classe que vamos criar, para que possamos começar a implementar o cadastro no nosso sistema.

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

```
...
```

```
@Model
```

```
public class AdminBooksBean {
    public void save(){
        System.out.println("Precisamos salvar o livro!!");
    }
}
```

Agora, quando apertamos o botão, a mensagem já é impressa no console do nosso servidor. Um ponto importante aqui é o uso da *annotation* `@Model`. Ela é uma *annotation* do CDI que indica que os objetos da classe por ela anotada sempre serão criados no escopo de `request`, e ficarão disponíveis para serem acessados pela *Expression Language*.

Por padrão, o nome da variável cujo objeto ficará acessível é o mesmo nome da classe, com a primeira letra em minúsculo. Esse é um caso bem comum para uma aplicação web, na qual, em geral, queremos criar novos objetos a partir de cada requisição.

Voltando para nossa funcionalidade, precisamos pelo menos imprimir as informações de cadastro que foram preenchidas pelo usuário. Para fazer isso, é necessário associar cada um dos campos do nosso formulário a uma propriedade.

```
<h:form>
  <div>
    <h:outputLabel>Titulo</h:outputLabel>
    <h:inputText value="#{adminBooksBean.product.title}" />
  </div>
  <div>
    <h:outputLabel>Descrição</h:outputLabel>
    <h:inputTextarea cols="20" rows="10"
      value="#{adminBooksBean.product.description}" />
  </div>

  <div>
    <h:outputLabel>Número de páginas</h:outputLabel>
    <h:inputText
      value="#{adminBooksBean.product.numberOfPages}" />
  </div>

  <div>
    <h:outputLabel>Preço</h:outputLabel>
    <h:inputText value="#{adminBooksBean.product.price}" />
  </div>
  <h:commandButton value="Gravar"
    action="#{adminBooksBean.save}" />
</h:form>
```

Analise, por exemplo, a propriedade `value` do `input` referente ao título do livro. Estamos dizendo que a classe `AdminBooksBean` possui um método chamado `getProduct`, e que este retorna um objeto que possui um *getter* e um *setter* para a propriedade `title`. Dessa forma, o JSF consegue navegar pelas propriedades dos `beans`, também conhecidos no mercado como **Managed Beans**, e popular os referidos campos.

Agora, para fazer esse código funcionar, precisamos implementar as partes que faltam no nosso código. Inicialmente, vamos criar a classe que representa o nosso livro.

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

```
//imports
```



```
public class Book {  
  
    private Integer id;  
    private String title;  
    private String description;  
    private int numberOfPages;  
    private BigDecimal price;  
  
    //getters e setters  
  
    //adicione um toString interessante.  
}
```

Agora, precisamos realizar as alterações necessárias na classe `AdminBooksBean`.

```
public class AdminBooksBean {  
  
    private Book product = new Book();  
  
    public void save(){  
        System.out.println(product);  
    }  
  
    public Book getProduct() {  
        return product;  
    }  
  
}
```

Caso você realize um novo teste na aplicação, deverá ver as informações que foram passadas pelo usuário.

Reflexão sobre como a requisição é feita

Esse é um bom momento para pensarmos um pouco em como as coisas acontecem dentro de uma aplicação JSF. Na web tradicional, somos acostumados a dar nomes para os *inputs*, e pegar os valores associados a estes nomes

do lado do servidor. Só que pare e olhe para o nosso formulário: onde estão os nomes? Toda a associação é realizada por meio do atributo `value`, disponível nas tags do próprio JSF.

Quando utilizamos o JSF, para cada tela carregada, é construída uma árvore de componentes representando cada um dos elementos que escrevemos na página. Tanto é que, se você olhar o código-fonte que aparece no seu navegador, encontrará algo assim:

```
<input type="hidden" name="javax.faces.ViewState"
      id="j_id1:javax.faces.ViewState:0"
      value=
      autocomplete="off" />
```

Esse é o `id` que foi dado para a árvore criada. Agora, todas as vezes em que realizarmos um `post` por meio do nosso formulário, o `id` será mantido, pelo menos enquanto não fizermos nenhum *redirect*.

Experimente acessar novamente o endereço do formulário em vez de submeter os dados do `form`. Você verá que o `id` vai mudar. Para ir mantendo o estado dos valores dos inputs do lado do servidor, o JSF guarda a árvore na sessão do usuário.

Agora, você pode se perguntar: *para que serve esse `id`, já que ele mantém o estado da árvore na sessão?* Esse é um ótimo questionamento! O problema é que, se você abrir várias abas no seu navegador e, como a sessão continuará sendo a mesma, o framework vai precisar distinguir sobre qual árvore o `request` está falando.

É por conta desse mecanismo que os valores que você preencheu no formulário para realizar os testes continuaram nos inputs, mesmo depois de você ter feito a requisição. Em uma aplicação web normal, esses valores teriam desaparecido. É também por conta disso que você não precisa colocar nomes nos inputs, nem actions nos formulários, tudo é associado no momento em que ele constrói a árvore de componentes.

3.3 GRAVANDO OS DADOS NO BANCO DE DADOS

Por mais que nosso formulário já esteja um tanto quanto funcional, ainda não estamos efetivamente gravando as informações passadas. Para melhorar essa

parte, vamos começar a efetuar as alterações necessárias a partir do nosso `controller`.

A primeira coisa é começar a usar um DAO (*Data Access Object*) responsável pelo acesso aos dados referente à classe `Book`.

LEMBRETE SOBRE O DAO

O Data Access Object é apenas uma classe cujo objetivo é isolar o acesso aos dados de uma determinada parte do sistema. No nosso exemplo, vamos precisar gravar, listar e carregar um livro. Não queremos misturar essa parte de acesso à infraestrutura com nossas lógicas.

O código do método `save` deve ficar parecido com o que segue:

```
public void save(){
    bookDAO.save(product);
}
```

E de onde vem essa instância da classe `BookDAO`? Uma das opções é que nós mesmos criemos esse objeto na mão.

```
@Model
public class AdminBooksBean {

    private Book product = new Book();
    private BookDAO bookDAO = new BookDAO();

    public void save(){
        bookDAO.save(product);
    }
}
```

O problema é que, provavelmente, o construtor do `BookDAO` vai precisar receber algum objeto que represente a conexão com o banco de dados e, nesse caso, vamos ter de começar a controlar essas dependências na mão. Essa é uma ótima parte para usarmos mais do CDI no nosso projeto.

Como a nossa classe já é gerenciada pelo CDI, podemos pedir para que ele instancie um novo objeto do tipo `BookDAO` para nós. O nosso único trabalho é indicar que precisamos receber essa instância **injetada**.

```
@Model
public class AdminBooksBean {

    private Book product = new Book();
    @Inject
    private BookDAO bookDAO;

    public void save(){
        bookDAO.save(product);
    }

    public Book getProduct() {
        return product;
    }
}
```

A annotation `@Inject` é justamente a responsável por indicar os pontos de injeção dentro da sua classe. Esse é um bom momento para lembrá-lo de que, em um projeto usando CDI, a única configuração necessária para habilitar o scan de classes é a criação do arquivo `beans.xml`, dentro da pasta `WEB-INF`.

PARA QUE SERVE UM ARQUIVO EM BRANCO?

É um ótimo questionamento, mas lembre: o arquivo está lá para habilitar seu projeto para ser scaneado pelo Weld, que é a implementação do CDI usada dentro do WildFly. Caso não fosse necessário criá-lo, o Weld teria de ir olhando dentro de cada classe de cada `jar` do seu projeto para descobrir quais classes gostariam de ser gerenciadas ou não. Dessa forma, ele primeiro busca pela existência do arquivo e, apenas em caso positivo, dá prosseguimento pela busca por classes.

Nesse exato momento, nosso código não compila, já que não criamos ainda a classe `BookDAO`, assim como não existe o método `save`. Chegou a hora de resolvermos essa parte do nosso código.

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

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

import br.com.casadocodigo.loja.models.Book;

public class BookDAO {

    @PersistenceContext
    private EntityManager manager;

    public void save(Book product) {
        manager.persist(product);
    }
}
```

Aqui estamos usando a JPA para realizar a parte de acesso efetivo ao nosso banco de dados. Na próxima seção, trataremos sobre como configurá-la.

Voltando ao nosso `BookDAO`, perceba que ela é uma classe bem normal. Não foi necessária nenhuma configuração extra, e essa é uma das belezas do CDI.

Até podemos usar annotations, como foi o caso na classe `AdminBooksBean`, mas várias vezes não precisamos adicionar nenhuma delas. Neste caso específico, você pode estar se perguntando: *qual é o escopo da criação desse objeto?*

Quando não colocamos nenhuma anotação em cima da classe, estamos dizendo que, na verdade, quem decide o escopo do objeto é o local onde ele vai ser injetado. Observe o exemplo:

```
@Model
public class AdminBooksBean {
```

```
...

@Inject
private BookDAO bookDAO = new BookDAO();

...
}
```

Como já vimos, a annotation `@Model` indica que os objetos da classe anotada devem viver pelo tempo de uma requisição web e, além disso, também devem ficar expostos na *expression language*. Como a classe `BookDAO` não definiu o escopo explicitamente, o tempo de vida do seu objeto, nesse caso, vai ser o da duração de um *request*, já que a classe que solicitou sua injeção definiu este escopo.

Caso tivéssemos pedido essa injeção em uma classe de escopo *session*, o objeto do tipo `BookDAO` teria assumido o mesmo. Essa estratégia dentro do CDI é chamada de escopo `@Dependent`. Não adicionar nenhuma anotação de tempo de vida é o mesmo que fazer o seguinte:

```
@Dependent
public class BookDAO {
    ...
}
```

Várias vezes, em uma aplicação web, o `@Dependent` vai ser o perfeito para você. Como quase tudo começa a partir do seu *Managed Bean*, que é criado uma vez por requisição, todos os outros objetos automaticamente vão seguir o mesmo tempo de vida.

3.4 CONFIGURANDO A JPA

Estamos quase conseguindo gravar o produto, só falta agora configurarmos a JPA (*Java Persistence API*) para que nosso acesso ao banco de dados fique completo. Para fazer isso, precisamos seguir alguns passos.

Mapeando a entidade

Chegou a hora de fazermos as configurações necessárias para que os objetos da classe `Book` possam ser salvos. Um passo importante é ensinar a implementação da JPA – em nosso caso, o Hibernate – que a classe `Book` vai representar uma tabela no banco de dados.

```
import javax.persistence.Entity;
...

@Entity
public class Book {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String title;
    private String description;
    private int numberOfPages;
    private BigDecimal price;

    ...
}
```

Aqui usamos algumas annotations específicas da JPA para realizar esses ensinamentos:

- `@Entity` – indica que a classe vai virar uma tabela;
- `@Id` – indica que o atributo em questão é a chave-primária;
- `@GeneratedValue` – indica a maneira como vai ser gerada a chave-primária;
- `@Lob` – indica que o atributo em questão vai ser salvo como *Clob* ou *Blob* no banco de dados.

Não acho que vale a pena entrarmos em todos os mapeamentos possíveis da JPA. O nosso foco durante o livro é construir uma aplicação web tirando proveito de todas as especificações que nos sejam úteis. Nesse contexto, a JPA é apenas mais uma delas.

@PersistenceContext e o Persistence.xml

Uma annotation que passou batida no nosso debate anterior foi a `@PersistenceContext`. Ela é utilizada para indicar, dentro de um *container* Java EE, a necessidade de injeção de um `EntityManager` que seja controlado e criado pelo próprio servidor. O legal dessa annotation é que sua semântica é muito forte, então, várias outras tecnologias, como o Spring, fazem seu uso para indicar a injeção de um `EntityManager`.

Um outro ponto importante são as informações de acesso ao banco de dados. A especificação JPA indica que devemos utilizar um arquivo chamado `persistence.xml`, e que nele devem estar contidas nossas informações de acesso ao banco de dados. Este arquivo deve ser criado na pasta `META-INF`, dentro do *build path* do seu projeto. Em projetos que usam Maven, como é o nosso caso, o caminho será `src/main/resources/META-INF/persistence.xml`.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="2.1"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

  <persistence-unit name="casadocodigo-dev"
    transaction-type="JTA">
    <description>Dev persistence unit</description>
    <provider>
      org.hibernate.ejb.HibernatePersistence
    </provider>
    <jta-data-source>
      java:jboss/datasources/casadocodigoDS
    </jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto"
        value="update"/>
      <property name="hibernate.show_sql"
        value="true" />
      <property name="hibernate.format_sql"
```



```
        value="true" />
    <property name="hibernate.dialect"
        value="org.hibernate.dialect
            .MySQL5InnoDBDialect"/>
    </properties>
</persistence-unit>
</persistence>
```

Perceba que fizemos algumas configurações específicas do Hibernate, como a propriedade `hibernate.hbm2ddl.auto`. Ela é usada para que não precisemos ficar criando tabelas novas no banco todas as vezes em que mapeamos uma nova entidade.

Uma outra tag interessante é a `<provider>`, necessária para informar qual implementação da JPA será utilizada no projeto. No nosso caso, como estamos dentro do WildFly, será usado o Hibernate.

Um ponto que você pode estar se perguntando é onde estão as configurações de acesso ao banco de dados em si. Sempre que vamos nos conectar, somos obrigados a informar coisas como:

- URL de acesso ao banco;
- Login;
- Senha;
- Driver necessário.

E você está coberto de razão, isso tudo é necessário. A única diferença é que vamos tirar proveito de toda a infraestrutura provida pelo nosso servidor de aplicação. Em vez de realizarmos essas configurações e deixarmos a cargo do Hibernate gerenciar as nossas conexões, vamos pedir para que o WildFly crie e gereencie-as. Dessa forma, a única tarefa do Hibernate será a de pedir uma nova conexão sempre que necessário.

Esse é justamente o objetivo da seguinte linha de configuração:

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

Simplesmente passamos um nome que está associado a um `DataSource`, que será configurado dentro do próprio servidor. Segure um pouco a curiosidade agora, na próxima seção discutiremos mais sobre esse nome. Ainda precisamos parar e pensar sobre a sigla **JTA** (*Java Transaction API*), que apareceu no nome dessa tag e também na propriedade `transaction-type`. Tudo isso será desvendado ainda neste capítulo.

3.5 CONFIGURAÇÃO DO DATASOURCE NO WILDFLY

Precisamos completar nossa configuração de acesso ao banco de dados, e o ponto que ficou faltando foi justamente a configuração relativa do `DataSource`. Aqui, vale a pena ressaltar que essa configuração vai variar entre os servidores de aplicação que você decidir usar.

Arquivo de configuração do WildFly

O arquivo responsável pelas configurações do nosso WildFly, nesse momento, é o `standalone-full.xml`. Ele fica na pasta `caminhoInstalacaoWildfly/standalone/configuration/standalone-full.xml`.

O nome `standalone` vem do fato de estarmos rodando apenas uma instância do servidor. Ele também suporta uma outra maneira de execução, na qual subimos várias instâncias do servidor e fazemos com que elas se comportem como se fossem apenas uma, modo conhecido como **Cluster Mode**.

Caso você esteja curioso e tenha navegado até a pasta `configuration`, verá que lá dentro existem outros arquivos de configurações. Esta é uma das grandes vantagens do WildFly: sua modularização. Podemos definir quais módulos são necessários para a nossa aplicação. Por exemplo, no caso do `standalone-full.xml`, recebemos pronto tudo de que normalmente podemos precisar.

- JPA;
- E-mail;
- JSF;

- Transações (JTA);
- Weld (CDI);
- Servlets;
- WebServices;
- EJB.

Perceba que já são várias opções. Há alguns itens dessa lista que já estudamos, alguns que você já deve saber – como a parte de Servlets –, e ainda outros que vamos trabalhar durante o livro. Além destes, ainda vamos usar uma outra especificação, que é a de mensageria, mas isso é assunto para ser visto mais para a frente.

Para exemplificar melhor, analise o trecho do arquivo que contém essas configurações:

```
<extensions>
  <extension module="org.jboss.as.clustering.infinispan"/>
  <extension module="org.jboss.as.connector"/>
  <extension module="org.jboss.as.deployment-scanner"/>
  <extension module="org.jboss.as.ear"/>
  <extension module="org.jboss.as.ejb3"/>
  <extension module="org.jboss.as.jaxrs"/>
  <extension module="org.jboss.as.jdr"/>
  <extension module="org.jboss.as.jmx"/>
  <extension module="org.jboss.as.jpa"/>
  <extension module="org.jboss.as.jsf"/>
  ...
</extensions>
```

Voltando ao nosso trabalho de configuração do `DataSource`, é justamente do arquivo `standalone-full.xml` que vamos precisar alterar para realizar uma parte desse trabalho.

Adicionando mais um DataSource

Caso você continue olhando o arquivo, verá que existe uma tag chamada `datasources`, que é justamente o local onde residem as configurações específicas de acesso ao banco de dados. Apenas por curiosidade, essa tag está dentro de uma outra chamada `subsystem`. Cada `subsystem` contém as configurações relativas às extensões mapeadas no início do arquivo.

Perceba que, dentro da tag `datasources`, já existe a configuração de um `DataSource`, justamente o que já vem pronto dentro do WildFly. Infelizmente, essa configuração usa o H2 como banco de dados, o que é não tão comum no mercado. Ele é mais utilizado para fins de testes.

Pensando nisso, vamos configurar um outro `DataSource` para que possamos usar o MySQL.

```
<datasources>
  <datasource>
    ...
  </datasource>

  <datasource
    jndi-name="java:jboss/datasources/casadocodigoDS"
    pool-name="casadocodigoDS">
      <connection-url>
        jdbc:mysql://localhost:3306/casadocodigo
      </connection-url>
      <driver>mysql</driver>
      <security>
        <user-name>root</user-name>
        <!-- Caso precise de senha-->
        <password>sua-senha</password>
      </security>
    </datasource>
  </datasources>
```

Perceba que a configuração é quase autoexplicativa. Algumas informações são exatamente as mesmas com que já estamos acostumados. Veja:

- `<connection-url>` – URL de conexão para sua base de dados;

- `<security>` – contém as informações de login e senha para o banco;

Entretanto, existem algumas tags e atributos que merecem um pouco mais de atenção. O mais simples é o atributo `pool-name`. Ele representa o nome do *pool de conexões* que será criado para armazenar todas as conexões que forem criadas pelo nosso `datasource`. Caso você queira especificar mais informações sobre esse *pool*, pode usar a tag `<pool>`. Veja um exemplo a seguir, a configuração não é obrigatória.

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

Outro atributo, ainda na própria tag `datasource`, é o `jndi-name`. Nele, passamos o nome que deve ser referenciado pela aplicação, para que ela possa recuperar a referência para o objeto que representa o `datasource`.

A JNDI é uma outra especificação do mundo Java que é muito utilizada. Quase todos os objetos que são criados pelo próprio container ficam disponíveis para serem acessados via algum nome que pode ser registrado. Quem faz todo o trabalho para buscar o objeto por esse nome é a implementação da especificação JNDI (*Java Naming and Directory Interface*).

Ainda em relação à JNDI, é importante prestar atenção no início da String de definição do valor, começando com `java:jboss/`. Esse começo é o que informa o nível de disponibilidade dos objetos criados dentro do container, e é o que chamamos de *namespace* da JNDI.

No caso do WildFly, para expor objetos que devem ser recuperados remotamente, podemos trabalhar com os seguintes namespaces:

- `java:jboss/`
- `java:/`

Além dos atributos que discutimos – que estão presentes na própria tag `datasource` –, existe mais um detalhe bem relevante: a tag `driver`. Perceba que apenas colocamos o nome `mysql`. Mas como o container vai saber qual classe deve ser usada para conectar neste banco? Do mesmo jeito que usamos banco MySQL, poderíamos ter usado o PostgreSQL. Caso você preste bastante atenção, verá que o mesmo ocorre na definição do driver do `datasource` anterior, com o H2.

```
<datasource jndi-name="java:jboss/datasources/ExampleDS"
  pool-name="ExampleDS" enabled="true"
  use-java-context="true">
  <connection-url>...</connection-url>
  <driver>h2</driver>
  ...
</datasource>
```

Ele também define apenas um nome do driver.

Agora, vem um outro momento importante. Perceba que, logo abaixo das tags `datasource`, existe a declaração de uma tag chamada `drivers`. É justamente nela que uma parte deste misterioso nome começa a ser resolvida.

```
<drivers>
  <driver name="h2" module="com.h2database.h2">
    <xa-datasource-class>org.h2.jdbcx.JdbcDataSource
    </xa-datasource-class>
  </driver>
</drivers>
```

O nome informado na configuração do `datasource` é apenas uma chave para o WildFly poder encontrar as configurações definidas na tag `driver`. Por exemplo, para o nosso exemplo com `mysql`, podemos adicionar outra tag.

```
<drivers>
  <driver name="h2" module="com.h2database.h2">
```

```
<xa-datasource-class>org.h2.jdbcx.JdbcDataSource
</xa-datasource-class>
</driver>
<driver name="mysql" module="com.mysql">
  <datasource-class>
    com.mysql.jdbc.Driver
  </datasource-class>
</driver>
</drivers>
```

Agora, vou pedir um favor, não fique chateado. Ainda precisamos realizar mais um passo para finalmente configurar nosso `datasource`. Com tudo isso pronto, falta apenas mais um detalhe: informar onde está o `jar` com as classes do driver do MySQL. E aqui o WildFly toma mais uma decisão.

Várias vezes, as empresas querem definir uma versão padrão de certas bibliotecas, por exemplo, os drivers de acesso ao banco de dados. Para este tipo de caso, em alguns servidores, você simplesmente adiciona esses `jars` em uma pasta compartilhada. Por exemplo, no TomCat, isso ficaria na pasta `lib` do servidor.

E se alguma aplicação precisar da versão atualizada do driver, como ela fará agora? Uma atualização pode impactar em várias outras aplicações.

Para permitir o compartilhamento de `jars` entre aplicações, mas também com o pensamento de possibilitar a evolução das versões, o WildFly usou um esquema de modularização. Adicionamos os `jars` necessários dentro da pasta `modules`, que fica dentro da pasta de instalação do servidor.

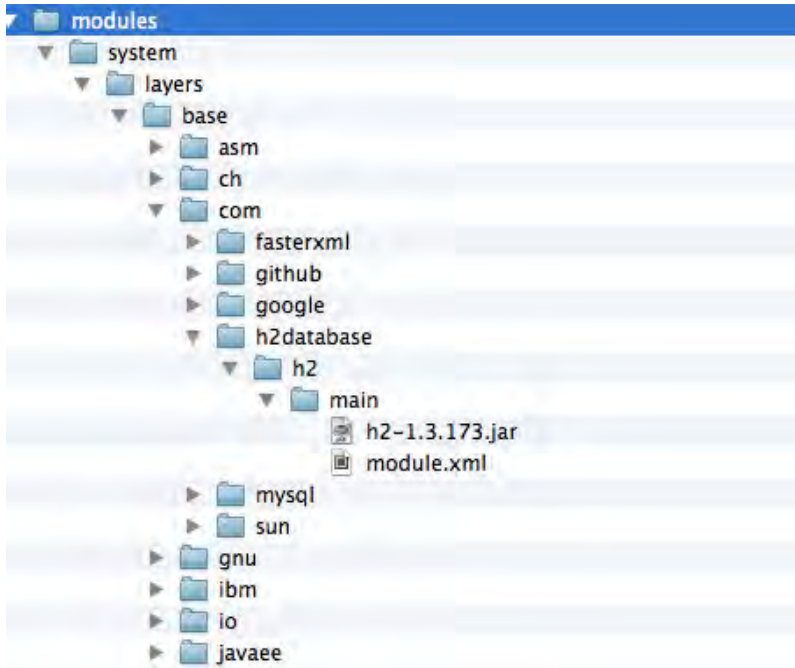


Fig. 3.1: Estrutura de módulos do WildFly

Perceba que o `com.h2database.h2` coincidentemente é o caminho até a pasta `main`, onde existem o `jar` e um arquivo de configuração referente ao banco H2. Vamos dar uma olhada nesse arquivo.

```
<module xmlns="urn:jboss:module:1.3" name="com.h2database.h2">

  <resources>
    <resource-root path="h2-1.3.173.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
    <module name="javax.transaction.api"/>
    <module name="javax.servlet.api" optional="true"/>
  </dependencies>
</module>
```

Não precisamos ficar debatendo sobre seu conteúdo. A parte mais impor-

tante para nós é o atributo `name` e a tag `resources`.

- `name` – valor a ser referenciado por alguma configuração externa, como o driver.
- `resources` – informa a localização dos `jars` necessários para que o módulo seja carregado.

Precisamos criar algo parecido para a configuração do driver do MySQL. Então, dentro da pasta `modules/system/layers/base/com`, crie a seguinte estrutura de pastas: `mysql/main`. Dentro da pasta `main`, crie o arquivo chamado `module.xml` com o seguinte conteúdo:

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

<module xmlns="urn:jboss:module:1.3" name="com.mysql">

    <resources>
        <resource-root path="mysql-connector-java-5.1.35.jar"/>
    </resources>
    <dependencies>
        <module name="javax.api"/>
    </dependencies>
</module>
```

Para fechar, só precisamos do driver. Você pode baixá-lo em <http://central.maven.org/maven2/mysql/mysql-connector-java/5.1.35/mysql-connector-java-5.1.35.jar>.

Pronto! Agora com tudo configurado, você já pode iniciar o WildFly e tentar cadastrar um novo livro.

Alterando o arquivo de configuração pelo Eclipse

O arquivo de configuração do seu WildFly pode ser alterado diretamente por meio da navegação no sistema de arquivos, como foi mostrado durante o capítulo, assim como por dentro do seu Eclipse.

Para fazer da segunda forma, que este autor acha mais simples, basta abrir a aba `Servers` e navegar diretamente até o arquivo.

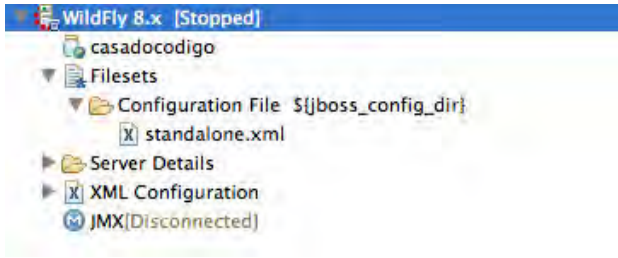


Fig. 3.2: Alterando o standalone.xml pelo Eclipse

Você pode fazer todas as alterações necessárias a partir de agora, por meio da própria IDE. Durante o capítulo, essa forma não foi mostrada diretamente justamente para termos sempre em mente que é importante entender como as coisas funcionam, para não ficarmos presos à IDE.

3.6 A NECESSIDADE DE UMA TRANSAÇÃO

Quando tentamos gravar um novo livro, nesse momento, recebemos a seguinte exception:

```
javax.servlet.ServletException:  
javax.persistence.TransactionRequiredException: JBAS011469:  
Transaction is required to perform this operation  
(either use a transaction or extended persistence context)
```

A exception é até bem clara, fala que precisamos de uma transação para conseguirmos salvar um objeto. Para a nossa sorte, lidar com transações já é suportado pela própria especificação Java EE. Assim, nosso único trabalho será fazer uma simples configuração.

...

```
import javax.transaction.Transactional;
```

```
@Model
```

```
public class AdminBooksBean {
```

```
private Book product = new Book();
@Inject
private BookDAO bookDAO;

@Transactional
public void save(){
    bookDAO.save(product);
}

public Book getProduct() {
    return product;
}
}
```

A annotation `@Transactional` foi introduzida a partir do Java EE 7, e permite marcar que métodos gerenciados pelo CDI rodem dentro de um contexto transacional. Além disso, no início do capítulo, também fizemos uma leve configuração no arquivo `persistence.xml`.

```
<persistence-unit name="casadocodigo-persistence-unit"
    transaction-type="JTA">
    <jta-data-source>
        java:jboss/datasources/casadocodigoDS
    </jta-data-source>
</persistence>
```

Indicamos que quem vai cuidar das nossas transações é a JTA, justamente a especificação que cuida da parte de transações no Java EE. Também é por conta dessa especificação que usamos a tag `<jta-data-source>`, para informar que nosso `DataSource` deve ser criado de modo que as transações sejam gerenciadas pela JTA.

Apenas por curiosidade, quando rodamos nossa aplicação em um servidor de aplicação, o valor do atributo `transaction-type` já é JTA por padrão. Quando rodamos uma aplicação que usa a JPA dentro de outro ambiente, o valor default é `RESOURCE_LOCAL`, que informa que o responsável por cuidar das transações é o próprio `EntityManager`.

Agora, finalmente, somos capazes de gravar os nossos livros!

Opinião do autor sobre a JTA

A Java Transaction API é uma especificação que vai bem além do que cuidar de simples transações de banco de dados. Suas implementações suportam que várias operações sejam envolvidas na mesma transação, por exemplo:

- Comunicação com banco de dados;
- Comunicação com servidores de mensageria;
- Comunicação com múltiplos bancos de dados na mesma transação.

A possibilidade de termos todas essas operações envolvidas em uma mesma transação, por mais que exista, é baixa. Geralmente, este tipo de cenário pode ser encontrado em alguns bancos ou bolsas de valores. Algumas vezes, chega a ficar até mais complexo, com sistemas rodando em máquinas diferentes tendo de participar da mesma transação.

Quando precisamos envolver múltiplos serviços na mesma transação, trabalhamos com o conceito de transação distribuída, que já é suportado pela própria JTA. Drivers de banco de dados, por exemplo, suportam o conceito por meio da implementação da interface `XAConnection`.

Dito isso, este autor não acha válido gastarmos muito tempo discutindo todas essas possibilidades da JTA, já que quase não temos contexto de uso para tais cenários. Ainda vamos voltar à JTA quando fomos discutir mensageria, controle mais fino sobre a transação corrente etc., que tendem a ser casos mais práticos e relevantes na maior parte da sua vida como desenvolvedor.

3.7 CONCLUSÃO

Neste capítulo, passamos por muita coisa. Foi necessária muita configuração para que nosso acesso ao banco de dados fosse possível. Só que o saldo positivo de todo esse trabalho é que aprendemos mais sobre o WildFly e suas configurações.

Por sinal, a configuração específica do servidor ainda é o “calcanhar de Aquiles” da especificação Java EE. Já temos muitos detalhes que funcionam

no automático, mas como muitas especificações têm relação com detalhes de infraestrutura, somos obrigados a realizar esses ajustes.

A parte boa disso tudo é que conseguimos gravar nossos livros no banco de dados e, olhando apenas para o código, até que não tivemos nada complexo. A parte potencialmente mais complicada deste momento, que era o controle transacional, funcionou apenas com o uso de uma anotação. E essa é a parte que mais evoluiu na especificação, escrever código Java em si tem ficado muito fácil.

No próximo capítulo, vamos evoluir o nosso cadastro, associando autores com os livros cadastrados, e listando e adicionando mensagens de confirmação. Minha sugestão é que você não pare ainda. Venha comigo e vamos completar essa funcionalidade.

CAPÍTULO 4

Melhorando o cadastro e um pouco mais de JSF

Nosso cadastro de livros já começou a funcionar, só que está faltando uma característica bem importante: os autores dos livros. No site da Casa do Código, os livros podem ser escritos tanto por uma pessoa como por várias.

Um caso bom para mostrar essa possibilidade é a página do livro *Coletânea Front-end* (<http://www.casadocodigo.com.br/products/livro-coletanea-front-end>) . Ele foi escrito por diversas pessoas, e o nome de cada um é exibido dentro do site.

Coletânea Front-end: Uma antologia da comunidade front-end brasileira

Almir Filho, Bernard De Luna, Caio Gondim, Deivid Marques, Diego Eis, Eduardo Shiota, Giovanni Keppelen, Luiz Corte Real, Jaydson Gomes, Reinaldo Ferraz e Sérgio Lopes

Fig. 4.1: Imagem ilustrando os nomes dos autores

Vamos começar a trabalhar na nossa aplicação para termos a mesma possibilidade.

4.1 ASSOCIANDO VÁRIOS AUTORES

Nossa primeira tarefa vai ser preparar a nossa tela de cadastro para que o usuário que administra a loja possa associar os autores. Para isso, vamos alterar o arquivo `livros/form.xhtml`.

```
...
<div>
    <h:outputLabel>Autores</h:outputLabel>
    <h:selectManyListbox
        value="#{adminBooksBean.selectedAuthorsIds}"
        <f:selectItems value="#{adminBooksBean.authors}"
            var="author"
            itemLabel="#{author.name}" itemValue="#{author.id}"/>
    </h:selectManyListbox>
</div>
```

Aqui foi necessário usar as tags `selectManyListbox` e `selectItems`, para que possamos construir uma tag `select` do HTML com a possibilidade de escolha de vários itens. Mesmo que estejamos usando apenas algumas tags do JSF, já temos trabalho a ser feito. Por exemplo, de onde vem a listagem de

autores que precisamos exibir na tela? Qual lista vai ser preenchida com as informações selecionadas pelo usuário?

Vamos por partes. Perceba que no atributo `value` da tag `selectManyListbox` referenciamos a propriedade `selectedAuthorsIds`. É justamente a lista que vamos manter na nossa classe `AdminBooksBean` para podermos receber os IDs selecionados.

```
@Model
public class AdminBooksBean {
    ...
    private List<Integer> selectedAuthorsIds =
        new ArrayList<>();

    // get e set para esse atributo.
}
```

Sempre que um usuário selecionar os autores e tentar cadastrar um novo livro, vamos usar os `ids` escolhidos para associar os autores ao livro.

```
@Model
public class AdminBooksBean {
    ...
    private List<Integer> selectedAuthorsIds =
        new ArrayList<>();

    @Transactional
    public void save(){
        populateBookAuthor();
        bookDAO.save(product);
    }

    private void populateBookAuthor() {
        selectedAuthorsIds.stream().map( (id) -> {
            return new Author(id);
        }).forEach(product :: add);
    }
}
```

O método `populateBookAuthor` usa um pouco da parte dos lambdas do Java 8. Lembre também de criar o método `add` na classe `Book`, para

que possamos adicionar cada um dos livros. Para que nossa tela funcione de maneira adequada, precisamos fornecer a lista de autores que deve ser exibida.

Perceba que a tag `selectItems` faz referência à propriedade `authors`, da classe `AdminBooksBean`.

```
<f:selectItems value="#{adminBooksBean.authors}" var="author"
    itemLabel="#{author.name}" itemValue="#{author.id}" />
```

Para que isso funcione, é necessário que pelo menos seja criado um *getter* no nosso bean.

```
@Model
public class AdminBooksBean {
    ...
    private List<Author> authors = new ArrayList<Author>();
    ...

    public List<Author> getAuthors() {
        return authors;
    }
}
```

Nesse momento, o nosso código nem compila, já que ainda não criamos a classe que representa o autor. Ainda bem que esse é um problema fácil de se resolver.

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

...

@Entity
public class Author {

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

```
/**
 * @deprecated Apenas para o uso dos frameworks.
 */
public Author(){

}

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

//métodos de acesso
}
```

Quando você reiniciar o servidor, dada a nossa configuração do Hibernate, o banco de dados já vai ser atualizado com a nova tabela. Para facilitar, já vamos criar alguns autores, basta executar os seguintes inserts no seu banco de dados:

```
insert into Author(name) values('Alberto Souza');
insert into Author(name) values('Mauricio Aniche');
insert into Author(name) values('Adriano Almeida');
insert into Author(name) values('Paulo Silveira');
insert into Author(name) values('Sergio Lopes');
insert into Author(name) values('Guilherme Silveira');
```

Agora, caso você acesse a tela de cadastro de novos livros, vai perceber que ela até funciona, mas não exibe nenhum autor no campo de seleção. Na verdade, o leitor mais atento talvez até tenha percebido isso, já que em nenhum momento foi carregada uma lista de autores em nosso código.

Nesse momento, sempre que usarmos a classe `AdminBooksBean` vamos ter a necessidade de carregar a lista de autores. Pensando nisso, uma primeira solução que pode passar pela nossa cabeça é a de carregar essa lista direto no construtor.

```
@Model
public class AdminBooksBean {
    @Inject
```

```

private AuthorDAO authorDAO;
private List<Author> authors = new ArrayList<Author>();
...

public AdminBooksBean(){
    this.authors = authorDAO.list();
}

...
}

```

O grande problema aqui é que o construtor vai ser executado antes da injeção de dependências ter sido realizada no nosso objeto. Para resolver isso, pensando em Orientação a Objetos (OO), bastaria que recebêssemos as dependências pelo construtor.

```

@Inject
public AdminBooksBean(AuthorDAO authorDAO, BookDAO bookDAO) {
    this.bookDAO = bookDAO;
    this.authors = authorDAO.list();
}

```

Caso você tente fazer essa alteração, no momento do *restart* do servidor, você receberá a seguinte exception:

```

WELD-001435: Normal scoped bean class
br.com.casadocodigo.loja.managedbeans.AdminBooksBean is not
proxyable because it has no no-args constructor - Managed Bean
[class br.com.casadocodigo.loja.managedbeans.AdminBooksBean] with
qualifiers [@Any @Default].

```

Aqui somos obrigados a aceitar o destino. A especificação do CDI define que toda classe cujo escopo for superior a *dependent* precisa ter um construtor sem argumentos. Podemos até trabalhar com um construtor sem argumentos e outro com, como já estamos acostumados a fazer para atender os frameworks, mas nesse caso usaremos uma solução alternativa.

Como precisamos rodar um código após a parte de injeção ter sido feita, podemos criar um método no nosso *bean* e anotá-lo com `@PostConstruct`.

```
@PostConstruct
public void loadObjects(){
    this.authors = authorDAO.list();
}
```

Essa annotation vem de uma especificação chamada **Commons Annotations for the Java Platform**. Você pode saber um pouco mais sobre ela acessando o endereço <https://jcp.org/en/jsr/detail?id=250>.

Seu objetivo é definir algumas annotations que podem ser úteis em todo tipo de aplicação Java, desde aplicações desktop até aplicações web que rodem dentro de servidores de aplicações, igual à nossa situação. Lembre-se de que esse método é chamado automaticamente pela nossa implementação do CDI, idealmente você nunca deveria chamá-lo manualmente. Até pensando nisso, a especificação permite que você diminua a visibilidade do método para `private`.

Problemas com a conversão

Com os autores devidamente carregados, chegou a hora de tentarmos realizar novos cadastros. O problema é que, na hora de concluir o cadastro, recebemos a seguinte exception:

```
javax.servlet.ServletException: java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.Integer
```

O JSF não conseguiu converter os `ids` dos autores para o tipo inteiro esperado dentro da lista. O problema é, até certo ponto, curioso, pois se adicionarmos um `System.out` dentro do método que adiciona os autores dentro do livro, serão impressos todos os `ids` que foram selecionados.

```
private void populateBookAuthor() {
    //Essa linha imprime => [3, 1, 2]=====
    System.out.println(selectedAuthorsIds+"=====");
    selectedAuthorsIds.stream().map( (id) -> {
        return new Author(id);
    }).forEach(product :: add);
}
```

A grande armadilha aqui é o uso da parte de `Generics`. Quando definimos o tipo de uma lista, somos obrigados a respeitá-lo, só que apenas em tempo de compilação. Em tempo de execução, até existem meios de recuperar o tipo que foi definido e garantir que ele só entrará na lista de valores que respeitem a restrição. A seguir, veja um exemplo usando um pouco de *reflection*.

```
Field field = AdminBooksBean.class.getDeclaredField
    ("selectedAuthorsIds");
ParameterizedType type =
    (ParameterizedType)field.getGenericType();
//imprime java.lang.Integer
System.out.println(type.getActualTypeArguments()[0]);
```

Só que nesse ponto o JSF comete um pecado. A implementação padrão que usamos, a *Mojarra*, não faz essa verificação de maneira automática e, por consequência, adiciona objetos do tipo `String` à lista que deveria ser de inteiros. E é exatamente essa falha, ou podemos chamar até de preguiça, que faz o nosso código receber uma exception do tipo `java.lang.ClassCastException`.

Para contornar esse problema, pelo menos neste momento, temos de recorrer ao uso de um detalhe de implementação do JSF. Podemos configurar para ser usado um conversor específico para os valores de um determinado componente.

```
<h:selectManyListbox
    value="#{adminBooksBean.selectedAuthorsIds}"
    converter="javax.faces.Integer">
    <f:selectItems value="#{adminBooksBean.authors}"
        var="author"
        itemLabel="#{author.name}" itemValue="#{author.id}" />
</h:selectManyListbox>
```

Repare que passamos o `id` do conversor responsável por transformar objetos do tipo `String` em objetos do tipo `Integer`. Essa informação pode ser encontrada dentro da própria classe que define o conversor.

```
package javax.faces.convert;
```

```
public class IntegerConverter implements Converter {  
  
    /**  
     * <p>The standard converter id for this converter.</p>  
     */  
    public static final String CONVERTER_ID =  
        "javax.faces.Integer";  
  
    ...  
}
```

Você também pode navegar até <http://docs.oracle.com/javasee/7/tutorial/jsf-page-core001.htm> para encontrar os `ids` necessários.

Independente da forma, estamos quebrando o encapsulamento do conversor para conseguir realizar uma simples transformação. Provavelmente, não acontecerá, mas, caso o `id` seja trocado, muitas aplicações podem sofrer.

Pronto, agora conseguimos cadastrar novos livros com seus autores. No próximo capítulo, vamos melhorar ainda mais essa parte de conversão!

4.2 LIMPANDO O FORMULÁRIO

Depois que você realizou alguns cadastros, deve ter percebido que os dados ainda permaneceram nos campos do formulário. Essa é uma característica marcante do JSF. Por padrão, ele sempre mantém o estado da árvore de componentes usadas. Em nosso caso, estamos usando inputs que são preenchidos com valores vindos dos beans, então os dados são mantidos mesmo após um *request*.

Para o nosso cadastro, esse comportamento não faz muito sentido. Depois de cadastrar um novo livro, queremos que esse formulário fique limpo. Para fazer isso, precisamos limpar os dados mantidos dentro de todos os objetos que estão ligados ao nosso formulário.

```
@Transactional  
public void save(){  
    populateBookAuthor();  
    bookDAO.save(product);  
    clearObjects();  
}
```

```
}  
  
private void clearObjects() {  
    this.product = new Book();  
    this.selectedAuthorsIds.clear();  
}
```

Perceba que simplesmente instanciamos de novo os objetos. No caso da lista, retiramos todos os objetos que estavam dentro dela, mas poderíamos ter atribuído uma nova instância do `ArrayList`.

4.3 LISTANDO OS LIVROS

Outra funcionalidade importante na parte de administração da Casa do Código é a de listar os livros cadastrados. O interessante dessa implementação é que não precisaremos usar nada de novo, e servirá como revisão de alguns dos conceitos que foram discutidos até aqui.

Mais uma vez, vamos começar pensando na página, que terá o nome de `lista.xhtml`, e ficará, como já era esperado, na pasta `livros`:

```
<html xmlns:h="http://java.sun.com/jsf/html"  
    xmlns:f="http://xmlns.jcp.org/jsf/core"  
    xmlns="http://www.w3.org/1999/xhtml"  
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">  
    <h:body>  
        <h:dataTable value="#{adminListBooksBean.books}"  
            var="book">  
            <h:column>  
                <f:facet name="header">  
                    Título  
                </f:facet>  
                #{book.title}  
            </h:column>  
            <h:column>  
                <f:facet name="header">  
                    Número de páginas  
                </f:facet>  
                #{book.numberOfPages}
```



```
        </h:column>
        <h:column>
            <f:facet name="header">Autores</f:facet>
            <ui:repeat var="author"
                value="#{book.authors}"
                #{author.name}|
            </ui:repeat>
        </h:column>
    </h:dataTable>
</h:body>
</html>
```

É uma página bem comum, apenas usamos algumas tags do JSF para criarmos uma tabela e, além disso, exibimos quais são os autores associados ao livro. Para essa tela funcionar, perceba que precisamos de uma nova classe chamada `AdminListBooksBean`.

```
package br.com.casadocodigo.loja.managedbeans.admin;
```

```
@Model
```

```
public class AdminListBooksBean {
```

```
    @Inject
```

```
    private BookDAO bookDAO;
```

```
    private List<Book> books = new ArrayList<Book>();
```

```
    @PostConstruct
```

```
    private void loadObjects(){
```

```
        this.books = bookDAO.list();
```

```
    }
```

```
    public List<Book> getBooks() {
```

```
        return books;
```

```
    }
```

```
}
```

Perceba que não tem nada de muito diferente do que já fizemos no *bean* responsável pelo cadastro de novos livros. Também usamos o

@PostConstruct para acessar o DAO injetado, e invocarmos o método que vai retornar a lista de livros. Por sinal, esse método é justamente a única coisa nova nesse trecho de código.

```
public class BookDAO {

    @PersistenceContext
    private EntityManager manager;

    public void save(Book product) {
        manager.persist(product);
    }

    public List<Book> list() {
        return manager.createQuery("select distinct(b) from
                                   Book b join fetch b.authors", Book.class)
            .getResultList();
    }
}
```

A decisão de criar uma nova classe para usar nessa nova tela é uma decisão deste autor que vos escreve. Não existe nenhuma regra dizendo que você deve ter um *bean* para cada nova tela. Entretanto, a Orientação a Objetos (OO) diz que quanto mais separarmos as responsabilidades, melhor será quando tivermos que evoluir nosso código. Sem contar que, por qual motivo você ficaria carregando a lista de livros? Ela só é usada nessa tela, e não na outra.

4.4 FORWARD X REDIRECT

Agora que já inserimos e listamos, chegou a hora de melhorar um pouco o fluxo entre essas operações. Neste momento, quando um livro é inserido, o usuário volta para a mesma tela de cadastro. Nesse tipo de cenário, o fluxo mais indicado é voltar com o usuário para a listagem, talvez mostrando uma mensagem de sucesso.

```
@Transactional
public String save(){
    populateBookAuthor();
```

```
        .save(product);
        clearObjects();
        return "/livros/lista";
    }
```

Esse código implementa justamente o fluxo sugerido. Quando acabamos de salvar um novo livro, pedimos para o JSF nos direcionar para o endereço de listagem, responsável por listar os produtos. O ponto negativo dessa solução é que o endereço que fica na barra do navegador ainda é o último acessado pelo usuário, que, nesse caso, foi um `post` para `/livros/form`. Caso o nosso cliente aperte um `F5`, o navegador vai tentar refazer a última operação, causando uma nova inserção de produto no sistema.

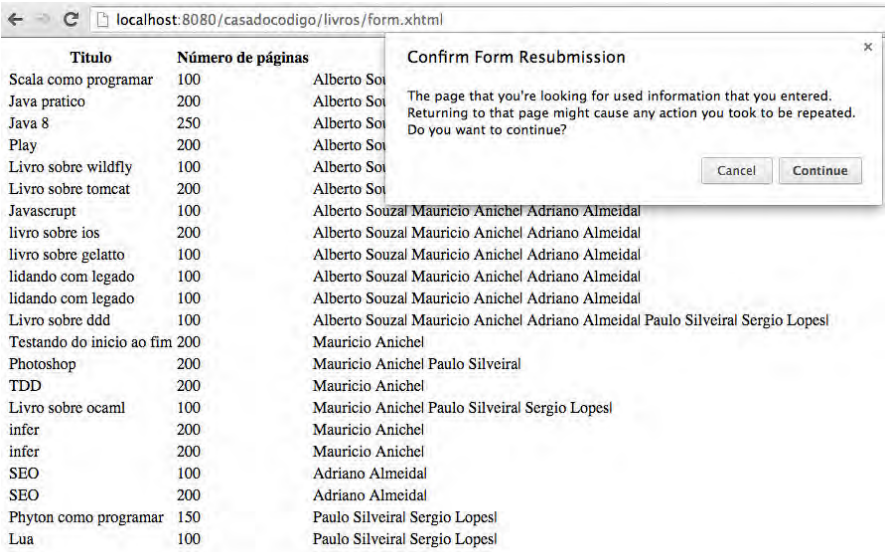


Fig. 4.2: Problema do forward depois de um post

Observe que o próprio navegador percebe que tem algo de estranho e pergunta se você tem certeza de que quer reenviar os dados. Essa técnica de redirecionamento que acontece apenas do lado do servidor é o que chamamos, no mundo Java, de **forward**. O browser nem sabe o que aconteceu, tanto que, se olharmos no console do Chrome, ele só vai identificar um *request*.

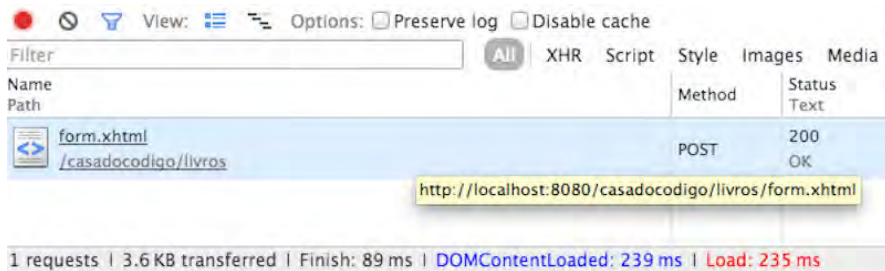


Fig. 4.3: Chrome tools indica apenas um request

É considerado uma má prática realizar um forward após o usuário ter feito um `post`, justamente por conta do problema da atualização. Para esse cenário, a melhor solução é forçar o usuário a fazer uma nova requisição para a nossa listagem e, dessa forma, permitir que ele atualize a página sem que um novo `post` seja realizado.

```
@Transactional
public String save(){
    populateBookAuthor();
    .save(product);
    return "/livros/lista?faces-redirect=true";
}
```

O parâmetro `faces-redirect=true` indica para o JSF que, em vez de simplesmente fazer um forward, é necessário que ele retorne o status 302 para o navegador, solicitando que este faça um novo *request* para o novo endereço.

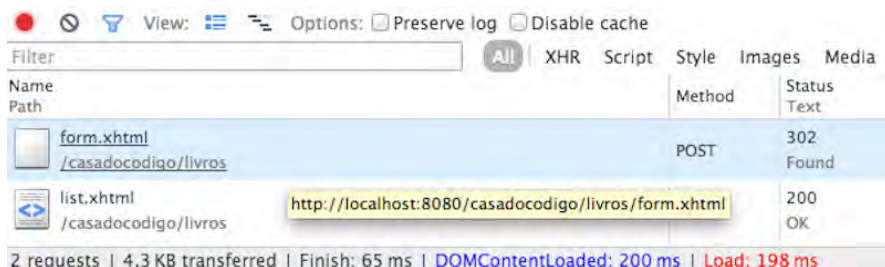


Fig. 4.4: Chrome tools indica dois requests



Fig. 4.5: Detalhes da resposta gerada pelo servidor

Perceba que, entre os cabeçalhos contidos na resposta, existe um chamado **Location**, que informa justamente qual é o endereço ao qual o navegador deve fazer a próxima requisição. Essa técnica, em que fazemos um *redirect* do lado do cliente logo após um `post` válido, é um padrão conhecido na web, chamado de **Always Redirect After Post**, e deve ser sempre utilizado.

ESTE PADRÃO NÃO É MUITO COMUM NO JSF

Por mais que esse padrão seja muito conhecido na web, o JSF não nos força a utilizar essa boa prática por conta do seu estilo de manter estado dos componentes no servidor. Aqui, é apenas uma questão de gosto. Este autor não enxerga a vantagem em deixar de trabalhar com o modelo padrão. Prefiro deixar para usar essa manutenção do estado quando realmente for necessário, por exemplo, quando estivermos trabalhando com validação. Além disso, acabamos de tirar um código que não tinha nada a ver com nossa lógica da aplicação, que era a limpeza dos objetos, apenas para não mantermos os dados no nosso formulário.

4.5 EXIBINDO A MENSAGEM DE SUCESSO

Ainda sobre o nosso *redirect*, geralmente é necessário que indiquemos para o usuário que tudo ocorreu bem. Isso é feito exibindo uma mensagem de

sucesso para o cliente em questão.

Uma das maneiras de realizar essa tarefa é utilizando uma classe do JSF, de que você ainda vai ouvir falar bastante, chamada `FacesContext`.

```
@Transactional
public String save(){
    populateBookAuthor();
    bookDAO.save(product);

    FacesContext facesContext =
        FacesContext.getCurrentInstance();
    facesContext.addMessage(null,
        new FacesMessage("Livro gravado com sucesso"));

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

A classe `FacesContext` é usada internamente pelo JSF e contém várias informações associadas a um request na aplicação. A seguir, temos alguns métodos de exemplo:

- `getExternalContext()` – para acessarmos objetos da especificação de Servlets;
- `getMessagesList()` – para termos acesso a todas as mensagens adicionadas;
- `addMessage(...)` – para adicionarmos uma mensagem a ser exibida na view;
- `isValidationFailed()` – para verificar se houve falhas no processo de validação.

Ainda existem vários outros métodos. Perceba que, inclusive, usamos o `addMessage` para adicionar a mensagem de sucesso. O segundo parâmetro é até bem claro, a classe `FacesMessage` é a abstração do JSF para representar uma mensagem que vai ser adicionada à tela. Já o primeiro parâmetro, que no nosso caso foi `null`, descobriremos o motivo em alguns minutos.

Adicionamos a mensagem, mas uma pergunta que fica é: *onde e como vamos mostrar essa informação para o usuário?* No nosso caso, queremos que ela seja exibida na tela de listagem dos livros, mas só quando realmente tiver mensagens. Vamos alterar o arquivo `lista.xhtml`, e incluir a tag do JSF responsável por isso.

```
...
<h:body>
    <h:messages/>
    <h:dataTable value="#{adminListBooksBean.books}" var="book">
        ...
    </h:body>
```

A tag `messages` realiza todo o trabalho de pegar as mensagens adicionadas por meio do `FacesContext` e exibir na nossa tela.

Só para não esquecermos, vamos comentar sobre o `null` que está sendo passado como argumento no método `addMessage`. Esse primeiro parâmetro serve para indicarmos o `id` do elemento com o qual a mensagem deve estar associada. Como não queremos ligar a mensagem a nenhum elemento, passamos o `null` e a deixamos global, podendo ser exibida pela tag `messages`.

Agora, o engraçado é que, caso você realize um novo cadastro, a mensagem ainda não vai ser exibida. Quando adicionamos mensagens com o método `addMessage`, elas ficam disponíveis durante a execução do *request* atual. Como estamos fazendo um *redirect* do lado do cliente, elas se perdem, já que, para exibir a listagem, é feita uma nova requisição. Não se preocupe, esse é um problema clássico em aplicações web, e todos os frameworks famosos já resolvem isso para nós.

```
@Transactional
public String save(){
    populateBookAuthor();
    bookDAO.save(product);

    FacesContext facesContext =
        FacesContext.getCurrentInstance();
    facesContext.getExternalContext().getFlash()
```

```
        .setKeepMessages(true);
        facesContext.addMessage(null,
            new FacesMessage("Livro gravado com sucesso"));
        return "/livros/list?faces-redirect=true";
    }
```

O método `getFlash` retorna um objeto do tipo `Flash`. Invocando o `setKeepMessages` nesse objeto, estamos dizendo ao JSF que ele deve manter as mensagens até o próximo *request*. Pronto, com essa alteração, nossa mensagem de sucesso vai começar a ser exibida na tela da listagem.

ESCOPO FLASH É UM AJUSTE FINO

O jeito mais comum de passar informações em um *request* é por meio de parâmetros. Pensando do jeito mais normal possível, já que queremos passar a mensagem de sucesso para o *request* que vai retornar a tela de listagem, poderíamos ter o seguinte retorno no método `save`:

```
return "/livros/list?faces-redirect=true
        &sucesso=Livro gravado com sucesso";
```

Poderíamos acessar o parâmetro `sucesso` a partir do nosso bean responsável pela tela de listagem, e adicionar a mensagem no `FacesContext`.

```
HttpServletRequest request = (HttpServletRequest)
    FacesContext.getCurrentInstance().getExternalContext()
        .getRequest();
facesContext.addMessage(null,
    new FacesMessage(request.getParameter("sucesso")))
```

Como isso daria muito mais trabalho, o escopo `flash` cai como uma luva, mantendo a mensagem até o próximo *request*!

4.6 ISOLANDO O CÓDIGO DE INFRAESTRUTURA

Vamos analisar com um pouco mais de cuidado o código que acabou sendo escrito no método `save`.


```
@Transactional
public String save(){
    populateBookAuthor();
    bookDAO.save(product);

    FacesContext facesContext =
        FacesContext.getCurrentInstance();
    facesContext.getExternalContext().getFlash()
        .setKeepMessages(true);
    facesContext.addMessage(null,new
        FacesMessage("Livro gravado com sucesso"));

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

Acabamos com mais linhas de código dedicadas a usar detalhes internos do JSF do que para resolver o nosso problema em si, que é o de gravar novos livros. Perceba que estamos escrevendo o código necessário para recuperar um `FacesContext` quando, na verdade, só estamos interessados em usá-lo. Aqui, de novo, temos uma motivação clara para usar o conceito de injeção de dependências: isolar a criação do objeto do seu uso.

```
@Model
public class AdminBooksBean {

    @Inject
    private FacesContext facesContext;

    @Transactional
    public String save(){
        populateBookAuthor();
        bookDAO.save(product);

        facesContext.getExternalContext().getFlash()
            .setKeepMessages(true);
        facesContext.addMessage(null,
            new FacesMessage("Livro gravado com sucesso"));
        return "/livros/list?faces-redirect=true";
    }
}
```

```
    ...
}
```

Perceba que pedimos para receber injetado o objeto do tipo `FacesContext` em vez de ficar criando todas as vezes em que precisarmos dele. Agora, quando tentamos subir a aplicação, recebemos a seguinte exception:

```
WELD-001408: Unsatisfied dependencies for type FacesContext with
qualifiers @Default at injection point [BackedAnnotatedField]
@Inject private br.com.casadocodigo.loja.managedbeans.admin
    .AdminBooksBean.facesContext at
br.com.casadocodigo.loja.managedbeans.admin.AdminBooksBean
    .facesContext(AdminBooksBean.java:0)
```

Ela indica que o `WELD` não foi capaz de injetar o objeto no atributo `facesContext` da classe `AdminBooksBean`. A classe `FacesContext` é abstrata, portanto, não pode ser instanciada. Sua implementação padrão, a `FacesContextImpl`, tem um construtor que recebe objetos específicos do JSF também.

```
public FacesContextImpl(ExternalContext ec,
                        Lifecycle lifecycle){
    ...
}
```

Até a versão 2.2, versão atual do JSF, não tem uma integração completa com o CDI. Alguns componentes, como o `FacesContext`, ainda não são gerenciados pelo container de injeção. A parte legal é que isso não é um problema muito grande para nós, basta que ensinemos ao CDI como ele deve criar um objeto do tipo `FacesContext`.

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

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.context.RequestScoped;
import javax.enterprise.inject.Produces;
import javax.faces.context.FacesContext;
```

```
@ApplicationScoped
public class FacesContextProducer {

    @Produces
    @RequestScoped
    public FacesContext get(){
        return FacesContext.getCurrentInstance();
    }
}
```

O método anotado com `@Produces` deve ser sempre usado quando o processo de criação de um objeto não é o padrão. Outra situação é quando queremos produzir um objeto de uma classe que não é gerenciada pelo CDI. O `FacesContext`, por coincidência, se enquadra nas duas situações. Ainda usamos duas anotações referentes ao escopo:

- `@ApplicationScoped` – para dizer que só é necessária uma instância do `FacesContextProducer`;
- `@RequestScoped` – para dizer que o método `get` deve ser chamado uma vez a cada novo *request*.

É importante que o `FacesContext` seja novamente produzido a cada *request*, caso contrário, você pode acabar injetando uma instância que foi populada com informações de outra requisição.

JSF 2.3 MAIS INTEGRADO COM O CDI

A nova versão do JSF virá bem mais integrada ao CDI. Componentes internos já vão ter produtores default, nos poupando de criar este código dentro do nosso projeto.

Caso queira saber mais sobre as novas integrações, acesse <http://jdevelopment.nl/jsf-23/>.

Isolando a adição de mensagens no Flash

Nosso código melhorou um pouco, mas ainda temos 2 linhas que são responsáveis por adicionar uma mensagem no flash.

```
facesContext.getExternalContext().getFlash()
    .setKeepMessages(true);
facesContext.addMessage(null,
    new FacesMessage("Livro gravado com sucesso"));
```

Este é um dos trechos de código que também geram dor de cabeça. Para o nosso caso, o `addMessage` só faz sentido se você chamar a linha de cima. A chance de isso ser esquecido é relativamente alta e, provavelmente, vamos acabar perdendo um tempo precioso para achar a causa do bug.

Um outro fato que devemos considerar é que estamos programando orientando a objetos, e podemos criar abstrações um pouco melhores para que o código fique mais fácil de ser entendido.

```
@Model
public class AdminBooksBean {

    @Inject
    private MessagesHelper messagesHelper;

    @Transactional
    public String save(){
        populateBookAuthor();
        bookDAO.save(product);

        messagesHelper.addFlash(
            new FacesMessage("Livro gravado com sucesso"));
        return "/livros/list?faces-redirect=true";
    }
}
```

Perceba que agora temos um método que diz claramente o que está sendo feito. Usamos uma nova classe chamada de `MessagesHelper`, que foi criada justamente para isolar esse comportamento.

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

import javax.enterprise.context.ApplicationScoped;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import javax.inject.Inject;

public class MessagesHelper {

    @Inject
    private FacesContext facesContext;

    public void addFlash(FacesMessage facesMessage) {
        facesContext.getExternalContext().getFlash()
            .setKeepMessages(true);
        facesContext.addMessage(null, facesMessage);
    }
}
```

4.7 CONCLUSÃO

Neste capítulo, evoluímos nosso cadastro. Além disso, vimos um conceito importante em aplicações web, que é o escopo `Flash` e a capacidade de fazermos *redirects* no lado do cliente. Este último, apesar de ser um conceito natural em uma aplicação web, não é muito usado no mundo JSF, por conta do modelo de programação imposto pelo framework.

Também começamos uma discussão sobre os mecanismos de conversão dos valores vindos dos *requests* para os tipos do nosso sistema e, infelizmente, caímos em uma situação ruim, causada pelo próprio JSF.

Para fechar, discutimos um pouco de boas práticas de código, e vimos como o CDI nos ajudou a deixar o nosso código um pouco mais desacoplado e coeso.

No próximo capítulo, vamos tentar melhorar a conversão da nossa lista de autores e também tratar sobre a validação dos valores vindos do formulário. Caso ainda esteja com fôlego, não pare agora e continue a leitura.

CAPÍTULO 5

Validação e conversão de dados

O nosso cadastro já está funcional, mas estamos deixando de fazer uma coisa básica: validar os dados de entrada. Neste momento, caso um usuário queira cadastrar um novo livro, ele tem a opção de deixar todos os campos em branco.

5.1 VALIDAÇÃO BÁSICA

A primeira opção de solução para essa situação é escrever o código de validação dentro do próprio método `save`, da classe `AdminBooksBean`.

```
@Transactional
public String save(){
    populateBookAuthor();

    if(product.getTitle()==null ||
```

```
product.getTitle().trim().isEmpty()){
    messagesHelper.addMessage
        (new FacesMessage("titulo obrigatorio"));
}

if(product.getDescription()==null ||
    product.getDescription().trim().isEmpty()){
    messagesHelper.
        addMessage
            (new FacesMessage("descrição obrigatoria"));
}

if(messagesHelper.hasMessages()){
    //volta para a mesma tela
    return "/livros/form";
}

bookDAO.save(product);
messagesHelper.addFlash
    (new FacesMessage("Livro gravado com sucesso"));
clearObjects();
return "/livros/list?faces-redirect=true";
}
```

Apesar de adicionarmos uns `ifs` e simularmos uns novos métodos na classe `MessageHelper`, a lógica não tem nada de complicado, na verdade, deve ser bem parecida com várias das quais o leitor já presenciou. Um ponto negativo dessa solução é que o Managed Bean, que idealmente deveria apenas ficar chamando lógicas do sistema e controlando qual é o próximo passo de navegação, agora também está responsável pelo código de validação.

Como uma das ideias originais do JSF é ser muito útil para aplicações com muitos formulários, suas tags de input já vêm com atributos que habilitam esse tipo de validação básica.

```
<h:body>
    <h:messages/>
    <h:form>
        <div>
```



```
<h:outputLabel>Titulo</h:outputLabel>
<h:inputText
    value="#{adminBooksBean.product.title}"
    required="true"/>
</div>
<div>
    <h:outputLabel>Descrição</h:outputLabel>
    <h:inputTextarea cols="20" rows="10"
        value="#{adminBooksBean.product.description}"
        required="true"/>
</div>
<div>
    <h:outputLabel>Número de páginas</h:outputLabel>
    <h:inputText
        value="#{adminBooksBean.product.numberOfPages}"
        required="true">
        <f:validateLongRange minimum="80"/>
    </h:inputText>
    required="true"/>
</div>
<div>
    <h:outputLabel>Preço</h:outputLabel>
    <h:inputText value="#{adminBooksBean.product.price}"
        required="true">
        <f:validateDoubleRange
            minimum="10" maximum="100"/>
    </h:inputText>
</div>
<div>
    <h:outputLabel>Autores</h:outputLabel>
    <h:selectManyListbox
        value="#{adminBooksBean.selectedAuthorsIds}"
        required="true">
        <f:selectItems
            value="#{authorsList.get()}" var="author"
            itemLabel="#{author.name}"
            itemValue="#{author.id}" />
    </h:selectManyListbox>
```

```

    </div>
    <h:commandButton value="Gravar"
        action="#{adminBooksBean.save}" />
</h:form>
</h:body>
</html>

```

O atributo `required` é útil apenas para verificar se o campo foi preenchido ou não. Além desse atributo, ainda podemos associar validadores mais específicos com os inputs. Por exemplo, usamos o `validateDoubleRange` para colocar um limite nos possíveis valores do livro. É importante ressaltar que essa validação é realizada do lado do servidor, pois as tags de validação do JSF não geram qualquer código JavaScript.

Para exibir as mensagens de erro, também não precisamos alterar nada, dado que usamos a mesma tag `messages`. Para cada problema encontrado, o JSF adiciona mensagens exatamente do jeito que nós fizemos durante a lógica de criação de um livro.

Com essas alterações, sem nem precisarmos alterar código Java, já conseguimos nos precaver de formulários preenchidos de maneira incorreta.

O leitor mais atento deve ter percebido que acontece uma falha de validação: o usuário é automaticamente levado para a mesma tela em que ele estava, que é justamente o comportamento que nós desejamos. Além disso, também não foi necessário nenhum código extra para manter os valores dos inputs, situação muito comum em outros frameworks MVC. Esse é um ponto que devemos agradecer ao modelo manutenção de estado dos componentes do JSF, também conhecido como *stateful*.

Um pouco sobre o ciclo de vida

Quando realizamos a requisição, o JSF atualiza a árvore de componentes relativa à tela atual, aplicando os valores que foram preenchidos no formulário. Veja a seguir um trecho de código da classe responsável por isso.

```

/**
 * ApplyRequestValuesPhase executes <code>processDecodes</code>
 * on each component in the tree so that it may update it's
 * current value from the information included in the current

```

```
* request (parameters, headers, cookies and so on.)
*/
public class ApplyRequestValuesPhase extends Phase {
    public void execute(FacesContext facesContext)
        throws FacesException {

        if (LOGGER.isLoggable(Level.FINE)) {
            LOGGER.fine("Entering ApplyRequestValuesPhase");
        }

        UIComponent component = facesContext.getViewRoot();
        assert (null != component);

        try {
            component.processDecodes(facesContext);
        }

        ...
    }
}
```

O comentário está em inglês, mas ele basicamente fala a mesma coisa do parágrafo anterior. Depois que os valores são associados aos componentes, ele vai aplicar as regras de validações e, aí, quando elas falham, basta ele reescrever o HTML baseado no estado de cada elemento da tela.

Quando trabalhamos no modo convencional, geralmente nós temos de tomar conta de uma parte desse fluxo. Por isso que, na opinião deste autor, o modelo híbrido geralmente é o melhor. Mantemos o estado quando necessário, e recriamos tudo do zero quando for mais interessante, como no capítulo 4 (seção 4.4) com o *redirect*.

5.2 EXIBINDO AS MENSAGENS DE ERRO DE MANEIRA AMIGÁVEL

Por mais que o formulário esteja sendo validado, as mensagens de erro ainda não dizem muita coisa sobre o problema em questão.

```
j_idt4:j_idt8: Validation Error: Value is required.
j_idt4:j_idt12: Validation Error: Value is required.
j_idt4:j_idt16: Validation Error: Value is less than allowable
    minimum of '100'
j_idt4:preco: Validation Error: Value is required.
j_idt4:j_idt26: Validation Error: Value is required.
j_idt4:j_idt28: Validation Error: Value is required.
```

Pensando no cliente, essas mensagens e nada dizem praticamente a mesma coisa. Perceba que, antes de cada mensagem de validação, aparece uma espécie de `id`. Como não informamos nenhum `id` específico para nossos elementos, o JSF gera os dele para controle interno. Então, uma primeira ação para melhorar a mensagem é a de colocar `ids` para nossos elementos.

```
<h:form>
  <div>
    <h:outputLabel>Titulo</h:outputLabel>
    <h:inputText value="#{adminBooksBean.product.title}"
      required="true" id="titulo"/>
  </div>
  ...
```

Agora, se tentarmos cadastrar um livro sem título, a mensagem de validação será parecida com a seguinte:

```
j_idt4:titulo: Validation Error: Value is required.
```

Perceba que ainda sobrou um identificador gerado pelo JSF. Ele é o `id` do formulário ao qual o input pertence. Podemos usar o atributo `prependId`, da tag `form`, para dizer que não queremos que ele fique sendo adicionado como parte da identificação do elemento.

```
<h:form prependId="false">
  <div>
    <h:outputLabel>Titulo</h:outputLabel>
    <h:inputText value="#{adminBooksBean.product.title}"
      required="true" id="titulo"/>
  </div>
```

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

Agora, quando ocorrer uma falha de validação, a mensagem será parecida com essa:

```
titulo: Validation Error: Value is required.
...
```

Esta alteração já deixa o nosso usuário um pouco mais feliz, entretanto, essa mensagem em inglês ainda não tem muito a ver com o nosso projeto. Para tentar melhorar, podemos recorrer a outro atributo das tags de formulário.

```
<h:inputText value="#{adminBooksBean.product.title}"
  required="true"
  id="titulo"
  requiredMessage="Campo obrigatório"/>
...
```

O atributo `requiredMessage` permite que você especifique uma mensagem de validação relacionada ao atributo `required`. Com essa última alteração, quando tentamos cadastrar um livro sem dizer qual o título, recebemos a seguinte mensagem de validação:

```
Campo obrigatório
Campo obrigatório
Campo obrigatório
...
```

Até que a mensagem ficou do jeito que nós queríamos, porém, perdemos o detalhe que informava qual campo tinha dado problema. Para resolver este problema, podemos recorrer à tag `message`.

```
<div>
  <h:outputLabel>Titulo</h:outputLabel>
  <h:inputText value="#{adminBooksBean.product.title}"
    required="true" id="titulo"
    requiredMessage="Campo obrigatório"/>
  <h:message for="titulo"/>
</div>
```

Ela permite que associemos a mensagem com algum input da nossa página. Dessa forma, conseguimos exibir a mensagem onde for mais pertinente para a aplicação.

5.3 TROCANDO AS MENSAGENS DEFAULT DO JSF

Até que chegamos a uma solução de validação justa, só tem um detalhe que ainda pode nos incomodar, especificamente relacionado com a mensagem de validação. Decidimos que, quando o campo é obrigatório, vamos usar o texto de alerta “Campo obrigatório”. Caso você só tenha um formulário na aplicação, até que não faz diferença, mas em um contexto um pouco mais amplo, essa mensagem ficará espalhada em todos os lugares.

Todas as mensagens de conversão e validação que são exibidas pelo JSF podem ser customizadas. Elas, por default, encontram-se no arquivo `Messages.properties`, que vem dentro do `jar` da especificação do JSF.

TODAS AS MENSAGENS DE CONVERSÃO

Caso fique curioso para saber todas as possibilidades, acesse <http://bit.ly/messages-properties-jsf>.

Para alterar essas mensagens, precisamos criar a nossa versão do arquivo e substituir as chaves que têm relação com a nossa aplicação. Para começar, vamos criar um arquivo chamado `jsf_messages.properties` dentro de `src/main/resources`, com o seguinte conteúdo:

```
javax.faces.component.UIInput.REQUIRED={0}: Campo obrigatório
```

Usamos a mesma chave definida pelo JSF, só alteramos o valor. O parâmetro de índice **zero** é justamente o `id` do elemento. Agora, precisamos ensinar ao JSF que ele sempre deve procurar as mensagens primeiro no nosso arquivo e, depois, no dele. Para isso, vamos alterar o arquivo `faces-config.xml`, que está em `src/main/webapp/WEB-INF`.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee">
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    version="2.2"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">

<application>
    <message-bundle>jsf_messages</message-bundle>
</application>
</faces-config>
```

Você poderia ter colocado o arquivo em qualquer lugar no seu `classpath`, basta que informe na tag `message-bundle`. Com essa alteração, podemos inclusive apagar o uso do atributo `requiredMessage` dos nossos inputs. Agora, caso você tenha mantido a tag `messages` no início do seu formulário, quando acontecer um erro de validação, a mensagem será exibida do jeito que gostaríamos.

titulo: Campo obrigatório

Alterando a mensagem de conversão

Voltando um pouco para os problemas de validação, um outro erro que pode existir no formulário de cadastro é a entrada de um valor não condizente com o tipo do campo. Por exemplo, um usuário pode digitar “dez” em vez de “10” no campo de número de páginas.

Esse tipo de situação também acontece, em geral, com campos de valores de ponto flutuante, como é o caso do preço do nosso livro. Caso a pessoa digite “59,90”, ocorrerá um problema de conversão para nosso tipo `BigDecimal`, já que a representação de um valor deste tipo deve ser feita com `.` (ponto), e não com `,` (vírgula). Será exibida a seguinte mensagem, também através da tag `messages`:

preco: '59,90' must be a signed decimal number.

Como podemos ver, é uma mensagem nada amigável para o usuário comum.

Para exibir mensagens amigáveis e relativas aos erros de conversão, adicionamos as chaves que começam com `avax.faces.converter`.

`javax.faces.converter.IntegerConverter.INTEGER=''{2}''` deve ser um número `javax.faces.converter.BigDecimalConverter.DECIMAL={2}` deve ser um valor separado apenas por "."

Ex: 450.50 ou 4100.50

`javax.faces.component.UIInput.REQUIRED={0}`: Campo obrigatório

Quando algum problema de conversão acontece, o índice `{2}` recebe o `id` do input que originou o erro.

5.4 INTEGRAÇÃO COM A BEAN VALIDATION

A nossa validação já funciona integralmente, mas ainda tem um ponto que, pelo menos para este autor, é um pouco ruim. As nossas regras de validação estão na nossa view. Quem olha para nosso modelo não faz ideia do que é obrigatório para criar um livro. Sem contar que, caso tenhamos uma forma alternativa de cadastrar um livro – por exemplo, por meio de uma importação de um arquivo de texto –, não vamos conseguir reaproveitar as regras.

Esse tipo de situação é tão comum que virou até especificação, a *Bean Validation*. E como não podia ser diferente, o JSF é completamente integrado com ela. Essa é uma das vantagens de se construir um projeto baseado no Java EE, uma vez que cada vez mais as especificações se integram de maneira transparente.

A implementação que vamos utilizar é a que já vem por padrão no Wildfly, feita pelo time do Hibernate, chamada de *Hibernate Validator*.

Só precisamos ensinar a Bean Validation o que deve ser validado na nossa classe `Book`.

```
@Entity
public class Book {

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

    @NotBlank
```



```
@NotNull
private String title;

@NotBlank
@NotNull
@Length(min = 10)
private String description;

@Min(50)
private int numberOfPages;

@DecimalMin("20")
private BigDecimal price;

@ManyToMany
@Size(min = 1)
@NotNull
private List<Author> authors = new ArrayList<>();
```

A seguir, veja as explicações para algumas das annotations:

- `NotBlank` – verifica se a string é diferente de `null` e se também não é vazia;
- `Length` – verifica se o tamanho da string está entre um mínimo e um máximo;
- `Min` – verifica se o número está entre o mínimo e o máximo;
- `DecimalMin` – verifica se o ponto flutuante está entre o mínimo e o máximo;
- `Size` – mesmo comportamento do `@Length`, mas pode ser aplicado para listas, arrays etc.

É importante ressaltar que as validações são bem coesas. Por exemplo, a annotation `@NotEmpty` apenas dispara a validação de string vazia. A annotation `@NotNull` só verifica se o valor for diferente de `null`. Por conta

disso, usamos a `NotBlank`, que é uma annotation específica do Hibernate Validator que une os dois comportamentos.

Nesse exato momento, estamos com dois mecanismos de validação: o padrão da JSF e o da Bean Validation. Pensando na aplicação em si, não tem nenhum problema. Estamos protegidos por todos os lados. Entretanto, não é esse o jeito que você encontra no mercado. Hoje em dia, a Bean Validation é o jeito padrão de validar objetos no mundo Java, então vamos continuar com ela a partir daqui.

A primeira coisa que você deve fazer é apagar o atributo `required` das tags, e também retirar qualquer tag de validação que você tenha adicionado nos inputs do formulário. Com essas alterações, quando tentamos realizar um novo cadastro com dados inválidos, recebemos as seguintes mensagens de erro:

```
may not be empty
length must be between 10 and 2147483647
may not be empty
must be greater than or equal to 50
may not be null
size must be between 1 and 2147483647
```

Aqui temos dois problemas: a tag `messages` voltou a exibir as mensagens de erro, sem associar com o devido campo; e as mensagens são as default, definidas pelo próprio Hibernate Validator.

Para resolver o primeiro, vamos voltar ao arquivo de mensagens default do JSF.

```
javax.faces.validator.BeanValidator.MESSAGE={0}
```

Na chave que define a mensagem associada aos erros de validação da Bean Validation, foi configurado para só exibir o erro, sem informar qual é o campo. Esse caso é ideal se você optar por mensagens associadas diretamente ao input.

```
<div>
  <h:outputLabel>Titulo</h:outputLabel>
  <h:inputText value="#{adminBooksBean.product.title}"
```

```
        id="titulo"/>
    <h:message for="titulo"/>
</div>
```

Agora, caso queiramos exibir todas as mensagens juntas, já acaba ficando confuso. A decisão vai depender do estilo do sistema. Para o nosso, vamos optar por exibir todas juntas. Para isso funcionar, vamos sobrescrever a chave do arquivo de mensagens do JSF e adicionar o nome do campo que falhou na validação.

```
...
javax.faces.validator.BeanValidator.MESSAGE={1}:{0}
```

Com essa alteração, toda vez que a validação de um campo falhar, será exibida a mensagem com o seguinte padrão:

idDoElemento:mensagem de validação

Alterando as mensagens da Bean Validation

Para fechar, só falta customizarmos as mensagens de validação. Elas ainda, por default, estão aparecendo em inglês, mas como a Casa do Código é uma empresa brasileira, devemos exibi-las em português. Para fazermos isso, assim como já fizemos com as mensagens padrões do JSF, teremos de alterar o valor de algumas chaves já definidas pela especificação e pelo Hibernate Validator. O arquivo com essas chaves já vem dentro do `jar` do próprio Hibernate Validator, e seu nome é `ValidationMessages.properties`. Na sequência, veja algumas chaves que já vêm por default:

```
javax.validation.constraints.AssertFalse.message =
    must be false
javax.validation.constraints.AssertTrue.message  =
    must be true
javax.validation.constraints.Past.message         =
    must be in the past
javax.validation.constraints.Pattern.message      =
    must match "{regex}"
javax.validation.constraints.Size.message         =
    size must be between {min} and {max}
```

```
...
```

```
org.hibernate.validator.constraints.NotBlank.message =
    may not be empty
org.hibernate.validator.constraints.NotEmpty.message =
    may not be empty
```

Para forçarmos as nossas mensagens, devemos criar um arquivo de mesmo nome na raiz do nosso `classpath`, no caso, em `src/main/resources`. Aqui, é importante lembrar para o leitor que o nome desse arquivo está na especificação da Bean Validation, não temos a opção de querer trocar de nome. Com o arquivo criado, basta redefinir as mensagens.

```
org.hibernate.validator.constraints.NotEmpty.message =
    não pode ser em branco
org.hibernate.validator.constraints.NotBlank.message =
    não pode ser em branco
```

Pronto, combinando um pouco de especialização do JSF com um pouco de especialização da Bean Validation, conseguimos chegar ao resultado que queríamos. E ainda podemos escolher a melhor forma de exibir nossas mensagens.

VEJA TODAS AS MENSAGENS QUE PODEM SER ALTERADAS

Para ver as chaves que podem ser sobrescritas sem a necessidade de ficar abrindo o `jar`, acesse <https://github.com/hibernate/hibernate-validator/blob/master/engine/src/main/resources/org/hibernate/validator/ValidationMessages.properties>

5.5 CONVERTENDO A DATA

Na Casa do Código, muitos livros são escritos o tempo todo. Para ter um fluxo interessante de lançamentos, foi decidido que os livros cadastrados devem ter uma data de lançamento.

```
@Entity
public class Book {
```

```
...
@NotNull
@Future
private Calendar releaseDate;

...
}
```

Perceba que já estamos usando as annotations de validação da Bean Validation. Usamos a `@Future` para garantir que as datas de lançamento são sempre do dia atual para a frente. Agora, precisamos colocar um campo novo no nosso formulário.

```
...

<div>
  <h:outputLabel>
    Data de lançamento
  </h:outputLabel>
  <h:inputText
    value="#{adminBooksBean.product.releaseDate}" />
</div>
```

Nosso código usa um simples input de texto para receber uma data. É sempre educado facilitar a vida do usuário, e o HTML 5 já traz um input que automaticamente abre um calendário. Para o usarmos, devemos setar o atributo `type` com o valor `date`.

```
<h:inputText value="#{adminBooksBean.product.releaseDate}"
  type="date"/>
```

O problema é que o JSF ignora este nosso atributo, já que ele não faz parte dos atributos especificados para essa tag. Esse, na verdade, era um problema citado por vários usuários do framework; ele nos impedia de ter um controle mais fino sobre o HTML gerado. Só que, desde a última versão, o JSF vem tentando atacar este tópico ao fornecer mais controle aos elementos da view. Por exemplo, para este caso, podemos usar um novo `namespace`, que permite forçarmos o uso das propriedades que bem entendermos.

```

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

    ...

    <h:inputText
        value="#{adminBooksBean.product.releaseDate}"
        pt:type="date"/>
</html>

```

Esse é o objetivo do namespace <http://xmlns.jcp.org/jsf/passthrough>. Perceba que, para trocarmos o `type` do input, usamos `pt:type`. Os atributos associados a esse namespace são ignorados pela fase de renderização do JSF e, dessa forma, ele será mostrado na página final.

Agora, quando tentamos cadastrar um novo livro informando a data de lançamento, recebemos a seguinte mensagem de conversão:

```
Conversion Error setting value '2015-06-10' for 'null Converter'.
```

Caímos em um ponto específico do JSF. Como queremos que ele converta o valor inserido para um objeto que represente uma data, precisamos indicar que queremos usar um conversor específico. Perceba que o formato da data foi `yyyy-MM-dd`. Este é o formato usado pelo input do tipo `date` do HTML 5.

Essa padronização é interessante, já que ajuda no tratamento das datas do lado do servidor. Assim, sabemos que sempre vamos receber a data nesse formato.

```

<div>
    <h:outputLabel>Data de lançamento</h:outputLabel>
    <h:inputText
        value="#{adminBooksBean.product.releaseDate}"
        <f:convertDateTime pattern="yyyy-MM-dd"/>
    </h:inputText>
</div>

```

Usamos a tag `convertDateTime` para associar o conversor com o input em questão. Agora, quando tentamos cadastrar, caímos em outro problema.

```
javax.servlet.ServletException:  
    java.lang.IllegalArgumentException: Cannot convert 6/10/15  
    9:00 PM of type class java.util.Date to class java.util.Calendar
```

A exception nos diz que não foi possível converter o tipo `Date` para `Calendar`, que é justamente o tipo do nosso atributo. O conversor associado pega o valor passado no request, e gera um objeto do tipo `Date`. E esse é justamente o nosso problema.

Aqui, de novo vemos um pouco de preguiça na equipe da especificação, já que `Calendar` é um tipo bem antigo no Java e já merecia ter a conversão automática.

Criando um Converter específico

Para resolver este problema, vamos ser obrigados a criar a nossa implementação de `Converter`. O que não é nenhum problema, já que o JSF já vem preparado para ser estendido nessa parte.

```
package br.com.casadocodigo.loja.converters;  
  
import java.util.Calendar;  
import java.util.Date;  
  
import javax.faces.component.UIComponent;  
import javax.faces.context.FacesContext;  
import javax.faces.convert.Converter;  
import javax.faces.convert.DateTimeConverter;  
import javax.faces.convert.FacesConverter;  
  
@FacesConverter(forClass=Calendar.class)  
public class CalendarHtml5Converter implements Converter{  
  
    private static DateTimeConverter originalConverter =  
        new DateTimeConverter();  
  
    static {
```

```
        originalConverter.setPattern("yyyy-MM-dd");
    }

    @Override
    public Object getAsObject(FacesContext context,
        UIComponent component,
        String value) {
        Date date = (Date)
            originalConverter.getAsObject(context,
                component, value);
        if(date == null) {
            return null;
        }

        Calendar newCalendar = Calendar.getInstance();
        newCalendar.setTime(date);
        return newCalendar;
    }

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

        Calendar calendar = (Calendar) value;
        return originalConverter.getAsString(context, component,
            calendar.getTime());
    }
}
```

A grande jogada aqui é a annotation `FacesConverter`. Ela é a única configuração necessária para termos um novo conversor no sistema. Como já sabemos para qual tipo queremos converter, ainda usamos o atributo `forClass`. Dessa forma, o JSF já sabe que, sempre que associarmos um input

a uma propriedade do tipo `Calendar`, ele deve usar nosso conversor.

Para não implementar todo o código da conversão, tiramos proveito da classe que já existe no framework, a `DateTimeConverter`.

As implementações da interface `Converter` têm um papel muito importante no JSF. Elas são usadas tanto para transformar os valores vindos do request para objetos do nosso modelo quanto para transformar os nossos objetos em Strings que devem ser associadas com os inputs exibidos na view.

Perceba que foi necessário implementar dois métodos:

- `getAsObject` – invocado para transformar o valor passado como parâmetro no tipo esperado pelo modelo;
- `getAsString` – invocado para transformar o valor do modelo na String que deve ser associada ao input da view.

5.6 CONVERTER PARA ENTIDADES

Apesar de já termos adicionado nossas validações e tratado da conversão da data, ainda tem um ponto que incomoda este autor.

```
<div>
  <h:selectManyListbox
    value="#{adminBooksBean.selectedAuthorsIds}"
    converter="javax.faces.Integer">
    <f:selectItems value="#{adminBooksBean.authors}"
      var="author"
      itemLabel="#{author.name}"
      itemValue="#{author.id}" />
  </h:selectManyListbox>
</div>
```

Associamos os autores a uma lista de inteiros para, só depois, realmente criar nossa lista de objetos do tipo `Author` e associar ao livro em questão. A seguir, segue o código para lembrá-los:

```
private void populateBookAuthor() {
    selectedAuthorsIds.stream().map( (id) -> {
```

```

        return new Author(id));
    }).forEach(product :: add);
}

```

Adicionamos uma complexidade no nosso Managed Bean só porque o JSF não quis fazer um `Converter` mais inteligente. Agora chegou o momento de resolvermos isso e ficarmos mais felizes!

Para termos uma ideia do que queremos, vamos começar alterando a nossa página.

```

<div>
  <h:outputLabel>Autores</h:outputLabel>
  <h:selectManyListbox
    value="#{adminBooksBean.product.authors}">
    <f:selectItems value="#{adminBooksBean.authors}"
      var="author"
      itemLabel="#{author.name}" itemValue="#{author}" />
  </h:selectManyListbox>
</div>

```

A ideia é que sejamos capazes de adicionar os autores selecionados diretamente no objeto do tipo `Book` que mantemos na classe `AdminBooksBean`. Para refletir essa alteração, vamos também alterar o Managed Bean, de modo que não é mais necessário ficar criando os objetos do tipo `Author` na mão.

```

@Model
public class AdminBooksBean {

    ....

    @Transactional
    public String save(){
        productDAO.save(product);

        messagesHelper.addFlash(
            new FacesMessage("Livro gravado com sucesso"));

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

```

```
}
```

Bem mais direto! Agora, quando tentamos rodar, recebemos a seguinte exception:

```
Caused by: java.lang.IllegalArgumentException:
Can not set java.lang.Integer field
br.com.casadocodigo.loja.models.Author.id to java.lang.Integer
```

A exception é nova, mas o motivo é o mesmo que nos levou a usar uma lista de inteiros em vez de trabalhar com a lista de autores. Por conta de detalhes do `generics`, nossa lista de autores do objeto do tipo `Book` acabou preenchida com um monte de inteiros.

Caso você olhe toda a *stack*, verá que a exception, na verdade, é gerada quando o Hibernate vai persistir a lista e que não tem nada a ver com o JSF, pelo menos diretamente. Para tentar contornar essa situação, vamos criar nosso `Converter` genérico para converter entidades da JPA.

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

...

import org.picketbox.util.StringUtil;

@FacesConverter
public class EntityConverter implements Converter {

    @Override
    public Object getAsObject(FacesContext context,
        UIComponent component, String value) {

        if (StringUtil.isNullOrEmpty(value)) {
            return null;
        }

        UISelectItems uiComponent = (UISelectItems)
            component.getChildren()
                .get(0);
```

```
Collection<?> objects = (Collection<?>)
    uiComponent.getValue();

Object foundEntity = objects.stream()
    .filter((entity) -> {
        return getAsString(context, uiComponent,
            entity).equals(value);
    }).findFirst().get();

return foundEntity;
}

@Override
public String getAsString(FacesContext context,
    UIComponent component, Object value) {
    Field idField = findIdField(value);
    return getIdValue(value, idField);
}

private String getIdValue(Object value, Field idField) {
    try {
        Field field = value.getClass()
            .getDeclaredField(idField.getName());
        field.setAccessible(true);
        return field.get(value)
            .toString();
    } catch
        (IllegalArgumentException | IllegalAccessException
         | NoSuchFieldException | SecurityException e) {
        throw new RuntimeException(e);
    }
}

private Field findIdField(Object value) {
    Field idField =
        Arrays.stream(value.getClass().getDeclaredFields())
            .filter((field) -> field.getAnnotation(Id.class)
                != null)
```

```
        .findFirst().get();  
        return idField;  
    }  
  
}
```

Respire fundo, esse não é um código tão simples. Vou pedir para não olhar muito para os métodos privados, eles usam um pouco de *reflection* e estão aí apenas para dar um apoio. Vamos começar pelo mais simples, que é relembrando o objetivo.

Escrevemos esse `Converter` para que possamos adicionar diretamente os objetos selecionados no `combo` dentro da respectiva propriedade, que nesse caso é a `List<Author>` da classe `Book`. Como queremos fazer um conversor que possa ser aplicado para qualquer entidade, já implementamos um genérico.

Agora, vamos começar a entender um pouco mais do código. Começaremos pelo método mais direto, que é o `getAsString`. Como já estudamos (seção 5.5), ele é responsável por pegar um objeto e gerar a sua representação em `String`, para que o valor possa ser associado a um input da view. No nosso caso, queremos pegar qualquer entidade gerenciada pela JPA, acessar o atributo anotado com `@Id` e recuperar seu valor. Por isso, tivemos de usar um pouco da API de *reflection*.

O código foi quebrado em dois métodos privados justamente para que você pudesse olhar um pouco mais e, quem sabe, matar um pouco da curiosidade. A *reflection* em si não é tão importante, por isso não vamos gastar tempo explicando linha por linha.

O método `getAsObject` já requer um pouco mais de cuidado, pois ele tira proveito do estilo `stateful` do JSF. Vamos prestar atenção nas linhas de código que estão logo a seguir:

```
@Override  
public Object getAsObject(FacesContext context,  
    UIComponent component, String value) {  
    ...  
  
    UISelectItems uiComponent = (UISelectItems)
```

```
component.getChildren().get(0);

Collection<?> objects = (Collection<?>)
    uiComponent.getValue();

...
}
```

O parâmetro do tipo `UIComponent` indica qual foi o elemento da árvore de componentes utilizado. Na página, usamos a tag `selectManyListBox`, que é representada pela classe `HtmlSelectManyListbox`. Com essa informação em mente, também já sabemos que o filho direto desse elemento é a tag `selectItems`, que pode ser referenciada por meio da classe `UISelectItems`. É por isso que fazemos o *cast* do primeiro elemento retornado pelo método `getChildren`. Uma implementação mais precavida buscaria na lista retornada por um objeto do tipo `UISelectItems`.

O objeto do tipo `UISelectItems` possui o método chamado `getValue`, que retorna para nós a lista associada ao elemento. Esse código só é possível de ser executado porque, a cada novo `post` para o sistema, o JSF remonta a árvore de componentes necessária, para aplicar os valores dos parâmetros, nos possibilitando acessar o seu estado.

Agora, vem a segunda parte do código. Nela, o JSF vai invocar este método para cada `id` selecionado no `combo`. Pensando nisso, é necessário descobrir exatamente quais objetos da lista possuem os `ids` que foram escolhidos na hora do cadastro.

```
Object foundEntity = objects.stream().filter((entity) -> {
    return getAsString(context, uiComponent,
        entity.equals(value);
}).findFirst().get();
```

O parâmetro `value` representa o valor selecionado na tela. A nossa lógica simplesmente varre todos os objetos da lista e procura por algum que possua a representação em `String` igual ao valor que foi enviado, já que este foi gerado justamente pelo método `getAsString`. Dessa forma, conseguimos retornar as instâncias do objeto em questão – no nosso caso, o `Author`

–, e poupar todo aquele código de conversão que estava no nosso Managed Bean.

Associando o Converter com o componente

Por fim, para que o JSF use o nosso conversor mágico, é necessário associá-lo com o componente na tela.

```
<h:selectManyListbox value="#{adminBooksBean.product.authors}"
    converter="entityConverter">
    ...
</h:selectManyListbox>
```

Só que ainda não demos nenhum identificador para nosso Converter.

```
@FacesConverter("entityConverter")
public class EntityListConverter implements Converter {
    ...
}
```

Para fechar com chave de ouro, precisamos apenas fazer a última alteração. Durante o uso da tag `selectItems`, fizemos referência à propriedade `id` do objeto do tipo `Author`.

```
<f:selectItems value="#{authorsList.get()}" var="author"
    itemLabel="#{author.name}" itemValue="#{author.id}" />
```

Só que agora estamos usando um converter que sempre pega o `id` das entidades da JPA. Da maneira como está, nosso Converter vai procurar por um atributo anotado com `@Id` na classe `Integer`, e não na `Author`. Vamos promover essa alteração.

```
<f:selectItems value="#{authorsList.get()}" var="author"
    itemLabel="#{author.name}" itemValue="#{author}" />
```

5.7 CONCLUSÃO

Este foi um capítulo um pouco mais denso. Lidamos com os problemas de validação e conversão, típicos do dia a dia. A parte interessante é que passamos por vários detalhes do JSF e da Bean Validation.

Uma dica que posso dar é: sempre tente buscar o que já vem pronto dentro do framework, mas saiba o suficiente dele para conseguir ir além, quando necessário. Fizemos isso em algumas situações e valeu bastante a pena. O Converter para `Calendar` e o da lista podem ser usados em qualquer projeto que você venha a participar.

Acho que esse é um bom momento para você parar e refletir sobre tudo que foi estudado até aqui. Já trabalhamos com a JPA, CDI, JTA e bastante com o JSF. Ainda tem mais de JSF e de outras especificações por vir. Descanse e volte com toda força para continuar a leitura.

CAPÍTULO 6

Upload de arquivos

Uma característica importante das livrarias online é a disponibilização do sumário dos livros que estão à venda. A Casa do Código, como não poderia deixar de ser, também deve oferecer essa possibilidade a seus usuários.

6.1 RECEBENDO O ARQUIVO NO MANAGED BEAN

Vamos começar alterando a página da Casa do Código responsável pela entrada de dados de cada um dos novos livros.

```
<h:form>
    ...

    <div>
        <h:outputLabel value="Capa do livro"/>
```

```
<h:inputFile id="summary" label="Sumário do livro"/>
</div>
```

A única mudança foi a adição do input que vai receber o arquivo que, neste caso, deve ser do tipo `file`. Agora, se tentarmos enviar os dados desse formulário, devemos receber a seguinte exception:

```
javax.servlet.ServletException: UT010016:
    Not a multi part request
```

Not a multi part request significa que tentamos enviar o arquivo de uma forma que o JSF não consegue entender. Basicamente, o que acontece é que todos os dados do formulário são enviados de alguma maneira, e o padrão é que eles sejam submetidos usando o formato `application/x-www-form-urlencoded`. Dessa forma, os dados são enviados seguindo quase as mesmas regras usadas quando utilizamos o método `GET`. Uma sequência de chaves e valores.

A nossa situação atual exige que façamos uma alteração nesse formato, de modo que o arquivo possa ser integralmente passado do navegador para o servidor. É por isso que vamos utilizar o formato `multipart/form-data`.

```
<h:form enctype="multipart/form-data">
    ...

    <div>
        <h:outputLabel value="Capa do livro"/>
        <h:inputFile id="summary" label="Sumário do livro"/>
    </div>
```

O atributo `enctype` serve justamente para indicarmos como queremos mandar os dados do navegador para o servidor, e o `multipart/form-data` é o jeito utilizado quando precisamos transmitir arquivos. Por exemplo, já existe um trabalho sendo feito para que o seu `form` possa enviar os dados pelo formato JSON. Caso fique curioso, navegue até <http://www.w3.org/TR/html-json-forms/>.

Alterando o método do Managed Bean

Agora que nosso formulário já envia o arquivo, precisamos recebê-lo e lidar com ele. Faremos isso no método `save` da classe `AdminBooksBean`. O primeiro passo, na verdade, é fazer o *binding* entre o `inputFile` e alguma propriedade do nosso bean. O leitor mais atento devia até estar se perguntando qual era o motivo de não termos feito isso antes.

```
<div>
    <h:outputLabel value="Sumário do livro"/>
    <h:inputFile id="summary" value="#{adminBooksBean.summary}"
        label="Sumário do livro"/>
</div>
```

Aqui, assumimos que o bean vai ter uma propriedade chamada `summary`. É justamente nela que vamos guardar o arquivo que tiver sido enviado pelo formulário.

```
@Model
public class AdminBooksBean {
    ...
    private Part summary;

    public void setSummary(Part summary) {
        this.cover = cover;
    }

    public Part getSummary() {
        return cover;
    }

    @Transactional
    public String save(){

        System.out.println(summary.getName() + ";"
            +summary.getHeader("content-disposition"));
        ...
    }
}
```

A parte importante desse código é o atributo do tipo `Part`. Ele foi introduzido a partir da versão 3 da especificação de Servlets, e é a interface responsável por representar o arquivo que foi enviado pelo formulário.

Perceba que o nome do argumento é o mesmo nome do input que declaramos no formulário. Aqui usamos dois métodos:

- `getName()` – retorna o nome do input usado no formulário;
- `getHeader()` – usado para recuperar alguma informação sobre o que foi enviado.

Caso nós quiséssemos recuperar o nome do arquivo, como faríamos? O leitor mais experiente deve estar pensando que tem algum método que já retorna isso pronto, mas esse é justamente o problema.

A interface `Part` é a maneira mais primitiva de tratarmos os dados enviados por meio do `enctype multipart/form-data`. Para ficar um pouco mais claro, dê uma olhada na maneira como esses dados são enviados a partir do navegador.

▼ Request Payload

```

-----WebKitFormBoundarybD5VjnI2VannG8wf
Content-Disposition: form-data; name="j_idt4"

j_idt4
-----WebKitFormBoundarybD5VjnI2VannG8wf
Content-Disposition: form-data; name="j_idt4:j_idt8"

kfjdlkfjs
-----WebKitFormBoundarybD5VjnI2VannG8wf
Content-Disposition: form-data; name="j_idt4:j_idt12"

kljkfdsjfkjdsk
-----WebKitFormBoundarybD5VjnI2VannG8wf
Content-Disposition: form-data; name="j_idt4:j_idt16"

100
-----WebKitFormBoundarybD5VjnI2VannG8wf
Content-Disposition: form-data; name="j_idt4:preco"

200
-----WebKitFormBoundarybD5VjnI2VannG8wf
Content-Disposition: form-data; name="j_idt4:j_idt23"

2015-06-30
-----WebKitFormBoundarybD5VjnI2VannG8wf
Content-Disposition: form-data; name="j_idt4:j_idt27"

3
-----WebKitFormBoundarybD5VjnI2VannG8wf
Content-Disposition: form-data; name="j_idt4:summary"; filename="contaluz.pdf"
Content-Type: application/pdf

```

Fig. 6.1: Corpo do request para uma requisição multipart

Para os campos que não representam arquivos, o JSF já nos ajuda bastante e faz a ligação direta com as propriedades do nosso modelo. Para o arquivo enviado, vamos ter de fazer o parse e recuperar o nome dele, infelizmente. O nosso código vai ficar parecido com:

```

@Transactional
public String save(){

    System.out.println(extractFilename
        (summary.getHeader("content-disposition"));
}

```

```
private String extractFilename(String contentDisposition) {
    if (contentDisposition == null) {
        return null;
    }

    String fileNameKey = "filename=";
    int startIndex = contentDisposition.indexOf(fileNameKey);
    if (startIndex == -1) {
        return null;
    }
    String filename = contentDisposition.substring(startIndex
        + fileNameKey.length());
    if (filename.startsWith("\"")) {
        int endIndex = filename.indexOf("\"", 1);
        if (endIndex != -1) {
            return filename.substring(1, endIndex);
        }
    } else {
        int endIndex = filename.indexOf(";");
        if (endIndex != -1) {
            return filename.substring(0, endIndex);
        }
    }
    return filename;
}
```

Este é justamente o tipo de código que devemos evitar ter no nosso Managed Bean. Caso uma pessoa nova chegue ao seu projeto, ela vai perder mais tempo tentando entender isso do que prestando atenção no fluxo do projeto em si, que é o mais importante. Um pouco mais à frente, vamos usar o CDI para nos ajudar a deixar esse código mais coeso.

6.2 SALVANDO O CAMINHO DO ARQUIVO

Agora que já temos o objeto que representa o arquivo enviado em nossas mãos, é necessário gravá-lo em algum lugar. Vamos aproveitar também e já isolar esse código de recuperar o nome do arquivo e salvá-lo.

```
@Model
public class AdminBooksBean {
    ...
    private Part summary;
    @Inject
    private FileSaver fileSaver;

    @Transactional
    public String save(){
        String summaryPath =
            fileSaver.write("summaries", summary);
        ...
    }
}

package br.com.casadocodigo.loja.infra;

@RequestScoped
public class FileSaver {

    @Inject
    private HttpServletRequest request;

    private static final String CONTENT_DISPOSITION =
        "content-disposition";

    private static final String FILENAME_KEY = "filename=";

    public String write(String baseFolder, Part multipartFile) {
        String serverPath = request.getServletContext()
            .getRealPath("/") + baseFolder;

        String fileName = extractFilename(multipartFile
            .getHeader(CONTENT_DISPOSITION));

        String path = serverPath + "/" + fileName;

        try {
            multipartFile.write(path);
        }
    }
}
```

```
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    return baseFolder + "/" + fileName;
}

private String extractFilename(String contentDisposition) {
    if (contentDisposition == null) {
        return null;
    }
    int startIndex = contentDisposition
        .indexOf(FILENAME_KEY);
    if (startIndex == -1) {
        return null;
    }
    String filename = contentDisposition
        .substring(startIndex
            + FILENAME_KEY.length());
    if (filename.startsWith("\"")) {
        int endIndex = filename.indexOf("\"", 1);
        if (endIndex != -1) {
            return filename.substring(1, endIndex);
        }
    } else {
        int endIndex = filename.indexOf(";");
        if (endIndex != -1) {
            return filename.substring(0, endIndex);
        }
    }
    return filename;
}
}
```

Dê uma respirada e olhe com cuidado para o código. Perceba que não fizemos nada demais, apenas criamos outra classe responsável por gravar o arquivo em uma pasta da nossa aplicação. A ideia é não deixar que esse código atrapalhe o fluxo do método `save` no nosso Managed Bean. Além disso, é necessário criar o atributo `summaryPath` na classe `Book`, juntamente com

o seu getter e setter.

Dessa forma, caso seja necessário criar um link para o sumário, basta que façamos algo parecido com o que segue:

```
<a href="${book.summaryPath}">Sumário do livro</a>
```

Podemos inclusive acrescentar esse link na listagem atual dos livros.

...

```
<h:column>
  <f:facet name="header">Sumário</f:facet>
  <h:outputLink value="#{request.contextPath}
    /#{book.summaryPath}">
    #{book.summaryPath}
  </h:outputLink>
</h:column>
```

6.3 GRAVANDO OS ARQUIVOS FORA DO SERVIDOR WEB

Nossa lógica de upload grava os novos arquivos em uma pasta dentro da própria aplicação web. O problema dessa abordagem, mesmo quando estamos em ambiente de desenvolvimento, é que, a cada nova alteração que temos no projeto, a nossa IDE força um *hot deploy* no servidor. Em geral, o *hot deploy* destrói a aplicação que estava no servidor e cria uma nova, fazendo com que percamos os arquivos que já tinham sido enviados.

Além disso, um fato que incomoda o desenvolvedor é perceber que os arquivos enviados não ficam exatamente dentro do projeto que ele está visualizando – por exemplo, no Eclipse –, mas na instalação do servidor que foi escolhida. Para simplificar tudo isso, facilitando inclusive o processo de deploy que discutiremos mais para o fim do livro (capítulo 16), é que podemos usar um serviço como o *Amazon S3*.

Basicamente, a Amazon fornece um serviço onde podemos enviar nossos arquivos, e estes ficam disponíveis para serem acessados pela web. Eles possuem servidores dedicados só para isso.

Integração com o Amazon S3

Para quem estiver interessado nessa opção, a primeira coisa que devemos fazer é alterar o código que está na classe `FileSaver`, que é a responsável por gravar o arquivo.

```
public class FileSaver {

    ...

    public String write(String baseFolder,
        Part multipartFile) {
        AmazonS3Client s3 = client();
        String fileName = extractFilename(multipartFile
            .getHeader(CONTENT_DISPOSITION));
        try {
            s3.putObject("casadocodigo", fileName,
                multipartFile.getInputStream(),
                new ObjectMetadata());

            return "https://s3.amazonaws.com/casadocodigo/"
                +fileName;

        } catch (AmazonClientException | IOException e) {
            throw new RuntimeException(e);
        }
    }

    private AmazonS3Client client() {
        AWSCredentials credentials = new BasicAWSCredentials(
            ,
            );

        AmazonS3Client newClient =
            new AmazonS3Client(credentials,
                new ClientConfiguration());
        newClient.setS3ClientOptions(new S3ClientOptions()
            .withPathStyleAccess(true));
        return newClient;
    }
}
```

```
    }  
  
    ...  
  
}
```

Perceba que temos mais códigos relativos à integração com a Amazon do que relacionado ao CDI ou o JSF em si. De tudo o que foi escrito anteriormente, as linhas mais importantes são as que estão logo a seguir:

```
AmazonS3Client s3 = client();  
  
...  
  
s3.putObject("casadocodigo", fileName,  
            multipartFile.getInputStream(),  
            new ObjectMetadata());
```

O objeto do tipo `AmazonS3Client` encapsula toda a lógica e o protocolo de comunicação com a Amazon. O nosso único trabalho é passar os parâmetros.

- O primeiro é a pasta remota onde o arquivo vai ser salvo, também conhecida como *bucket*;
- O segundo é o nome do arquivo;
- O terceiro é alguma implementação da classe `InputStream`, que realmente representa o arquivo;
- O quarto são informações extras como: data de expiração do arquivo e qualquer outra informação que seja específica da aplicação.

A outra parte do código é a mesma para qualquer integração que você vá fazer com a Amazon. Sempre precisamos criar um objeto que contém as credenciais de acesso, para garantir que só o dono da conta possa fazer uploads para o bucket.

```
private AmazonS3Client client() {
    AWSCredentials credentials = new BasicAWSCredentials(
        ,
    );
    AmazonS3Client newClient = new AmazonS3Client(credentials,
        new ClientConfiguration());
    newClient.setS3ClientOptions(new S3ClientOptions()
        .withPathStyleAccess(true));
    return newClient;
}
```

Basicamente, estas seriam as alterações que teríamos de fazer no nosso código. Como deixamos encapsulada a lógica de gravação do arquivo, não somos obrigados a alterar nenhuma linha do nosso `Controller`.

Simulando o S3 localmente

O empecilho com a solução apresentada é que vamos ter de realmente criar uma conta na Amazon para conseguir realizar nossos testes, o que é completamente inviável. Para resolver este problema, existe um projeto chamado *S3 Ninja*, que pode ser encontrado em <http://s3ninja.net/>.

Ele foi criado com o objetivo de servir como um emulador para o S3 real, da Amazon. O S3 Ninja é uma aplicação web escrita em Java que sobe um servidor que aceita requisições vindas do próprio SDK da Amazon. Justamente o que estávamos procurando. Uma parte muito boa é que o seu uso é bem simples.

O primeiro passo é realizar o download do `.zip`, bastando acessar o endereço <https://oss.sonatype.org/content/groups/public/com/scireum/s3ninja/2.3/s3ninja-2.3.zip>.

Você também pode encontrar o link no `readme` do GitHub do projeto, em <https://github.com/asouza/casadocodigojavaee>.

Uma vez que você baixou o projeto, descompacte-o em uma pasta de sua preferência. Ainda dentro da pasta descompactada, precisamos criar uma estrutura simples de subpastas que vão simular os *buckets* da Amazon. Vamos criar primeiro a pasta `data` e, dentro dela, criaremos outra pasta, chamada `s3`. A estrutura final vai ficar parecida com a mostrada na figura a seguir:

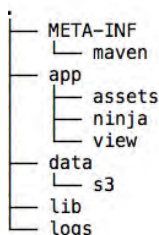


Fig. 6.2: Estrutura de pastas do S3 Ninja

Agora que a estrutura está criada, basta iniciarmos o servidor do S3 Ninja.

```
java IPL
```

Perceba que `IPL` é justamente um arquivo Java compilado que já vem na raiz do diretório descompactado. Após executar o comando, você deve ver a seguinte saída:

```
I N I T I A L   P R O G R A M   L O A D
```

```
-----
```

```
IPL from: /Users/alberto/ambiente/servidores/s3ninja-2.3-zip
```

```
IPL completed - Loading Sirius as stage2...
```

```
Opening port 9191 as shutdown listener
```

O servidor está rodando no endereço <http://localhost:9444/>. Você pode acessá-lo pelo navegador.



Fig. 6.3: Tela de boas-vindas do S3 Ninja

Agora que o nosso emulador está rodando, precisamos alterar o código de configuração de acesso à Amazon e fazê-lo apontar para o servidor local.

```
private AmazonS3Client client() {
    AWSCredentials credentials = new BasicAWSCredentials(
        ,
    );
    AmazonS3Client newClient = new AmazonS3Client(credentials,
        new ClientConfiguration());
    newClient.setS3ClientOptions(new S3ClientOptions()
        .withPathStyleAccess(true));

    //nova linha
    newClient.setEndpoint("http://localhost:9444/s3");
    return newClient;
}
```

Pronto! Quando cadastrarmos um novo livro, o arquivo do sumário vai ser gravado na pasta `data/s3/casadocodigo/nomedoarquivo`, no local que você escolheu para executar o S3 Ninja. Perceba que também alteramos o retorno do método `write`, da classe `FileSaver`.

```
public String write(String baseFolder, Part multipartFile) {
    AmazonS3Client s3 = client();
    String fileName = extractFilename(multipartFile
        .getHeader(CONTENT_DISPOSITION));
    try {
        s3.putObject("casadocodigo", fileName,
            multipartFile.getInputStream(),
            new ObjectMetadata());

        //url de acesso ao arquivo
        return "https://s3.amazonaws.com/casadocodigo/"
            +fileName+"?noAuth=true";

    } catch (AmazonClientException | IOException e) {
        throw new RuntimeException(e);
    }
}
```

```
}
```

A ideia é que nosso método já retorne o endereço de acesso ao arquivo enviado, para que possamos usá-lo nos links do nosso sistema.

Isolando a criação do `AmazonS3Client`

A nossa alteração nos levou a deixar o código de criação do objeto de acesso à Amazon misturado com o código da classe `FileSaver`. Como já tem sido a nossa prática, quando, por exemplo, recebemos injetado o `DAO`, o nosso objetivo nesse código é usar o cliente da Amazon, e não ficar criando. Então, vamos começar recebendo um objeto do tipo `AmazonS3Client` em vez de criá-lo.

```
public class FileSaver {

    @Autowired
    private AmazonS3Client s3;

    public String write(String baseFolder,
        MultipartFile multipartFile) {
        String fileName = extractFilename(multipartFile
            .getHeader(CONTENT_DISPOSITION));

        try {
            s3.putObject("casadocodigo", fileName,
                multipartFile.getInputStream(),
                new ObjectMetadata());

            return "http://localhost:9444/s3/casadocodigo/"
                + fileName + "?noAuth=true";

        } catch (AmazonClientException | IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Para este código funcionar, assim como fizemos quando foi necessário injetar o objeto do tipo `FacesContext`, precisamos criar um método para criar um objeto do tipo `AmazonS3Client`.

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

import javax.enterprise.inject.Produces;

import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.s3.AmazonS3Client;
import com.amazonaws.services.s3.S3ClientOptions;

public class AmazonS3ClientProducer {

    @Produces
    public AmazonS3Client s3Ninja() {
        AWSCredentials credentials = new BasicAWSCredentials(
            "AKIAI44QH8DHBVS7G13GL", "ajqanwseqovtzejezmkq"
        );

        AmazonS3Client newClient =
            new AmazonS3Client(credentials,
                new ClientConfiguration());
        newClient.setS3ClientOptions(new S3ClientOptions()
            .withPathStyleAccess(true));
        newClient.setEndpoint("http://localhost:9444/s3");
        return newClient;
    }
}
```

Um detalhe interessante é que esse nosso método produtor não deve ter nenhuma anotação de escopo. O problema é que a classe `AmazonS3Client` possui métodos marcados como `final`, e a especificação do CDI exige que todos os comportamentos da classe e, inclusive a classe, não estejam marcados como `final`, para suportar a injeção de objetos baseados em escopos diferentes do `Dependent`.

Agora você tem duas opções para salvar os arquivos enviados pelos seus usuários. Este autor sugere que você sempre guarde o arquivo em um local fora do servidor web. A complexidade adicionada ao projeto é compensada pela flexibilidade de manter os arquivos entre *reloads* da aplicação, em tempo de desenvolvimento.

Além disso, a instalação no ambiente de produção vai ser facilitada, já que você não vai ter de se preocupar em manter os arquivos que já tenham sido enviados pelos usuários. Tudo vai estar em um lugar separado.

Um detalhe simples, mas muito importante: **lembre-se sempre de subir o servidor do S3 Ninja antes de rodar sua aplicação!**

6.4 CONCLUSÃO

Neste capítulo, focamos unicamente no upload de arquivos. Essa é uma funcionalidade muito comum nas aplicações, mas muitas vezes esquecemos de olhar com detalhes para entender exatamente como funciona. O interessante é que tudo foi muito direto, não tivemos que perder tempo com nenhuma configuração extra. Basicamente, usamos tudo o que já estava pronto nas últimas versões das especificações do JSF e de Servlets.

Não pare agora, já vá para o próximo capítulo no qual vamos realizar o fechamento de uma compra. Vamos implementar um Carrinho, ver um pouco mais dos escopos do CDI e deixar o terreno preparado para realizarmos uma integração com outro sistema.

CAPÍTULO 7

Carrinho de compras

Já temos o cadastro de livros e a sua listagem na loja da Casa do Código, então chegou o momento de implementarmos o processo de compra. Para começarmos essa fase, precisamos, primeiramente, possibilitar que os usuários escolham os livros que mais lhes interessam e os coloquem em um carrinho de compras.

7.1 EXIBINDO OS LIVROS NA PÁGINA INICIAL

Na Casa do Código, os visitantes conseguem navegar para as páginas de detalhes dos livros a partir da listagem exibida na página inicial do site.



Fig. 7.1: Listagens da home

Perceba que temos duas listagens: uma com os últimos lançamentos e outra que exibe os últimos livros com preços promocionais. Nosso primeiro desafio é justamente reproduzir essa página.

Aqui, para ficar claro que realmente estamos reproduzindo o site da Casa do Código, será usado o mesmo HTML construído por eles. Como o código da página é relativamente grande, para ver a página completa, o leitor pode navegar até o link <https://github.com/asouza/casacodigojavaee/blob/master/src/main/webapp/site/index.xhtml>.

No trecho a seguir, veja algumas partes importantes. Apenas para fins de organização, esta deve ser criada em uma nova pasta chamada `site`, com o nome de `index.xhtml`.

```
<ui:repeat var="book" value="#{homeBean.lastReleases()}">
  <li class="col-left"><a href="/products/livro-plsql"
    class="block clearfix"> 
    <h2 class="product-title">#{book.title}</h2> <small
    class="buy-button">Lançamento!</small></a>

  </li>
</ui:repeat>
```

...

```
<ui:repeat var="book" value="#{homeBean.olderBooks()}">
  <li><a href="/products/livro-laravel-php"
    class="block clearfix">
      <h2 class="product-title">#{book.title}</h2>
      <small
        class="buy-button">Compre</small>
    </a></li>
</ui:repeat>
```

A página basicamente é um HTML, e usamos a tag `repeat` para imprimir os livros retornados pelos métodos em questão. Para concentrar as lógicas relativas à nossa home, foi criada a classe `HomeBean`.

```
package br.com.casadocodigo.loja.managedbeans.site;
```

```
//imports
```

```
@Model
```

```
public class HomeBean {
```

```
    @Inject
```

```
    private BookDAO bookDao;
```

```
    public List<Book> lastReleases(){
        return bookDao.lastReleases();
    }
```

```
    public List<Book> olderBooks(){
        return bookDao.olderBooks();
    }
```

```
}
```

Uma simples classe anotada com `@Model`, nada que você já não tenha

visto. Para o nosso código ficar completamente funcional, precisamos também adicionar os novos métodos na classe `BookDAO`.

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

//imports

public class BookDAO {

    @PersistenceContext
    private EntityManager manager;

    ...

    public List<Book> lastReleases() {
        return manager.createQuery("select b from Book b where
            b.releaseDate <= now() order by
                b.id desc", Book.class)
            .setMaxResults(3).getResultList();
    }

    public List<Book> olderBooks() {
        return manager.createQuery("select b from
            Book b", Book.class)
            .setMaxResults(20).getResultList();
    }

}
```

Aqui, usamos um pouco mais da JPA para conseguir fazer nossas queries. Um detalhe muito importante quando estamos trabalhando com a JPA é: pense na suas consultas como se estivesse realmente trabalhando com os objetos!

Claro que é importante conhecer SQL, mas quando você aceita trabalhar com uma ferramenta de Mapeamento Objeto-Relacional (ORM), é interessante que a maioria das suas queries consigam tirar proveito das vantagens oferecidas pela JPQL.

Pronto, com isso já conseguimos exibir todos os nossos livros cadastrados. Um detalhe importante para essa página funcionar bem é que todos os produtos cadastrados estejam com uma imagem associada. Para facilitar, simplesmente rode um comando SQL que atualize todas as capas para alguma imagem previamente cadastrada. Veja um exemplo:

```
update Book set coverPath =  
    'http://localhost:9444/s3/casadocodigo/  
covers_games-android-featured_medium.png?noAuth=true';
```

Um outro ponto importante é que, caso você tenha optado por usar o S3 Ninja, seu servidor deve estar rodando. Então, lembre-se de ir na pasta de instalação e rodar o seguinte comando:

```
java IPL
```

Pronto, já temos uma página inicial igual a da Casa do Código! E melhor ainda, não foi necessário usar nada de novo, trabalhamos apenas com o que já estudamos e conseguimos reproduzir uma aplicação bastante conhecida no mercado.

7.2 NAVEGANDO PARA O DETALHE DO PRODUTO

Agora que já temos a listagem dos livros, é necessário que os usuários da aplicação consigam navegar para a página de detalhes do livro. É nessa outra página que ele vai encontrar informações como: preço, autores e resumo do conteúdo da obra. Perceba que, atualmente, o link para os detalhes está fixo em nossa listagem.

```
<ui:repeat var="book" value="#{homeBean.lastReleases()}">  
    <li class="col-left"><a href="/products/livro-plsql"  
        ...  
    </li>  
</ui:repeat>  
  
...  
  
<ui:repeat var="book" value="#{homeBean.olderBooks()}">
```

```

    <li><a href="/products/livro-laravel-php" ...>
        ...
    </a></li>
</ui:repeat>

```

O `href` está sempre apontando para uma URL fixa, e precisamos deixar isso dinâmico. Essa é uma parte tranquila de resolver. Vamos dar uma olhada:

```

<ui:repeat var="book" value="#{homeBean.lastReleases()}">
    <li class="col-left"><a
        href="#{request.contextPath}/site/
            detalhe.xhtml?id=#{book.id}"
        ...
    </li>
</ui:repeat>

...

<ui:repeat var="book" value="#{homeBean.olderBooks()}">
    <li><a
        href="#{request.contextPath}/site/
            detalhe.xhtml?id=#{book.id}"
        ...>
        ...
    </a></li>
</ui:repeat>

```

Simplesmente apontamos agora para o endereço `detalhe.xhtml`, responsável por exibir os detalhes do livro. Temos também de passar o parâmetro indicando qual o `id` do livro que estamos buscando. Assim, usamos o `request.contextPath` para sempre garantir que estamos referenciando a raiz do contexto da nossa aplicação.

Agora é necessário que tenhamos essa página, para que esse nosso link funcione. Mais uma vez, vamos usar o layout original da Casa do Código e, como o HTML é muito grande, vamos apenas deixar os trechos importantes aqui no livro. O conteúdo inteiro pode ser encontrado em <https://github.com/asouza/casadocodigojavaee/blob/master/src/main/webapp/site/detalhe.xhtml>.

Essa página deve ser criada com o nome de `detalhe.xhtml`, e deve ficar na pasta `site`.


```
<html xmlns:h="http://java.sun.com/jsf/html"
xmlns:f="http://xmlns.jcp.org/jsf/core"
xmlns="http://www.w3.org/1999/xhtml"
xmlns:pt="http://xmlns.jcp.org/jsf/passthrough">

<head>

    ...

    <title>#{productDetailBean.book.title}</title>

    ...
</head>

...

<header>
    <div id="product-overview" class="container">
        
        <h1 class="product-title">
            #{productDetailBean.book.title}
        </h1>
        <p class="product-author">
            <span class="product-author-link">

                <ui:repeat var="author"
                    value="#{productDetailBean.book.authors}">
                    #{author.name}|
                </ui:repeat>

            </span>
        </p>

        <p class="book-description">
            #{productDetailBean.book.description}
        </p>
    </div>
</header>
```

```

    </div>
</header>

...

<section class="author product-detail">
<h2 class="section-title">

    <ui:repeat var="author"
        value="#{productDetailBean.book.authors}">
        #{author.name}|
    </ui:repeat>

</h2>
<span>

    <p class="book-description"><img class="author-image"
        src="" alt="" style="float: left; margin-right: 10px; "/>
    <a>Descricao sobre os autores</a>.</p>

</span>
</section>

<section class="data product-detail">
<h2 class="section-title">Dados do livro:</h2>
    <p>Número de páginas:
    <span>
        #{productDetailBean.book.numberOfPages}
    </span></p>

<p></p>
<p>
    Data de publicação:
        #{productDetailBean.book.releaseDate.time}
</p>

```

De novo, pelo menos até agora, é uma página apenas com códigos que já estudamos. Para lidar com a responsabilidade de carregar o produto, criamos

um novo bean, chamado `ProductDetailBean`.

```
package br.com.casadocodigo.loja.managedbeans.site;

//imports

@Model
public class ProductDetailBean {

    @Inject
    private BookDAO bookDAO;
    private Book book;

    @PostConstruct
    public void loadBook(){
        this.book = bookDAO.findById(id);
    }

    public Book getBook() {
        return book;
    }
}
```

Recebendo parâmetros e executando ações em requisições via GET

O leitor mais atento deve estar se perguntando de onde veio o `id` que está sendo usado para carregar o produto, e esse é um ótimo questionamento! A requisição que chega na página de detalhe vem a partir de um `GET`, sendo que, até o momento, qualquer método dos nossos beans só foi invocado a partir de requisições via `POST`.

Há muito tempo atrás, ainda na primeira versão do JSF, isso seria um problema para nós. Seríamos obrigados a fazer um `POST` apenas para satisfazer o framework e, com isso, a navegabilidade do nosso site estaria comprometida. Por exemplo, todos os links que são acessados via `POST` não são indexados pelo Google, o que seria extremamente prejudicial para a Casa do Código.

Percebendo que isso era um problema, a versão 2.0 do JSF trouxe o suporte para o recebimento de parâmetros que viessem a partir de um novo request. Assim, foi criada a *taglib* `viewParam`, que fica no pacote `core`,

bastando usá-la dentro de uma outra, chamada `metadata`.

Vamos alterar nossa página de detalhe:

```
<html xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:pt="http://xmlns.jcp.org/jsf/passthrough">

  <f:metadata>
    <f:viewParam id="id" name="id"
      value="#{productDetailBean.id}"/>
  </f:metadata>

  ...
```

Podemos receber o `id` e associá-lo com nosso bean, por meio dos métodos de acesso.

```
package br.com.casadocodigo.loja.managedbeans.site;

//imports

@Model
public class ProductDetailBean {

    @Inject
    private BookDAO bookDAO;
    private Book book;
    private Integer id;

    public void setId(Integer id){
        this.id = id;
    }

    public Integer getId() {
        return id;
    }

    @PostConstruct
```

```
public void loadBook(){
    this.book = bookDAO.findById(id);
}

public Book getBook() {
    return book;
}
}
```

Caso tentemos rodar este código, ainda receberemos uma exception. O problema é que o *binding* do parâmetro com seu *bean* só acontece na fase `UPDATE_MODEL_VALUES` do ciclo de vida do JSF. Este é o momento que o JSF vai setar os valores no seu *bean*; enquanto que a invocação do método anotado com `@PostConstruct`, nesse caso, acontece uma fase antes, na `PROCESS_VALIDATIONS`.

O momento que o *bean* é instanciado vai depender do seu uso dentro da página. Por exemplo, na home, ele só é instanciado no momento que o JSF vai escrever o HTML para o navegador, já que só o usamos para consumir informações. Já na página de detalhes, precisamos setar o `id` passado como argumento, assim, ele será instanciado na fase de validação para que qualquer verificação ou conversão necessária seja realizada.

É importante sempre lembrar das fases do JSF, conhecer um pouco de cada uma delas pode poupar bastante uma dor de cabeça na hora de desvendar bugs na aplicação.

Uma maneira simples de resolver este problema seria tirar o código do método anotado com `@PostConstruct`, e colocá-lo dentro do método `setId`. Só que, a partir do JSF 2.2, foi criada uma nova tag, chamada `viewAction`. O objetivo é que ela seja usada em páginas de um sistema escrito em JSF que foram acessadas via `GET`.

```
<f:metadata>
    <f:viewParam id="id" name="id"
        value="#{productDetailBean.id}"/>
    <f:viewAction action="#{productDetailBean.loadBook()}" />
</f:metadata>
```

Por padrão, a `viewAction` permite que invoquemos algum método do nosso bean na fase `INVOKE_APPLICATION`, justamente a usada pelo JSF para invocar nossas lógicas. Com isso, garantimos que o carregamento do livro só será feito quando o `id` já tiver sido setado.

Veja como o código final da classe `ProductDetailBean` deve ficar:

```
package br.com.casadocodigo.loja.managedbeans.site;
```

```
//import
```

```
@Model
```

```
public class ProductDetailBean {

    @Inject
    private BookDAO bookDAO;
    private Book book = new Book();
    private Integer id;

    public void setId(Integer id) {
        this.id = id;
    }

    public Integer getId() {
        return id;
    }

    public void loadBook() {
        this.book = bookDAO.findById(id);
    }

    public Book getBook() {
        return book;
    }
}
```

Para fechar, precisamos só adicionar o método `findById` na classe `BookDAO`.

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

```
//import

public class BookDAO {

    @PersistenceContext
    private EntityManager manager;

    ...

    public Book findById(Integer id) {
        return manager.find(Book.class, id);
    }

}
```

7.3 LIDANDO COM LAZYINITIALIZATIONEXCEPTION

Caso o leitor tenha acessado a página de detalhe do livro, vai perceber que uma exception foi lançada.

```
javax.servlet.ServletException: failed to lazily initialize a
collection of role: br.com.casadocodigo.loja.models.Book.authors,
could not initialize proxy - no Session
```

O problema acontece exatamente na linha a seguir, da página `detalhe.xhtml`:

```
<ui:repeat var="author"
    value="#{productDetailBean.book.authors}">
```

De novo temos um problema relacionado ao ciclo de vida, só que agora do `EntityManager`. O método `getAuthors` da classe `Book` usa um atributo mapeado com a annotation `@ManyToMany`. Por default, a implementação da JPA vai tentar carregar essa coleção apenas no momento do seu uso efetivo. Para que isso aconteça, é necessário que um `SQL` seja disparado no banco e, consequentemente, o `EntityManager` esteja aberto para possibilitar essa operação.

É justamente nesse ponto que o ciclo de vida nos passa uma rasteira. Por padrão, o `EntityManager` gerenciado pelo servidor de aplicação é criado no momento em que algum método da classe que o recebeu injetado for invocado. Só que todos os objetos que são carregados dentro do escopo do método deixam de ser gerenciados logo após o fim de sua execução, deixando o objeto em um estado chamado `detached`. Esse comportamento é o que nos leva ao problema que estamos vivenciando.

Uma maneira de alterar esse comportamento é definindo um escopo de transação. Por exemplo, caso tivéssemos anotado o método `loadBooks` com `@Transactional`, o `EntityManager` viveria até o fim do método, o que, infelizmente, também não é o suficiente para nós. Sem contar que a anotação `@Transactional` deve ser utilizada para demarcar métodos que realmente precisem de uma transação, como fizemos no momento de salvar um livro.

EntityManager extended e o EJB Stateful

Já existem soluções que podem ser usadas para resolver este problema. As opções mais comuns são:

- Criar um filtro da especificação de Servlets que comece uma transação no início do request, e que feche no fim;
- Criar um produtor de `EntityManager` do CDI e associá-lo com o escopo de requisição;
- Planejar todas as queries do sistema com o uso de *fetch joins*. Dessa forma, tudo o que é *lazy* já é carregado.

O planejamento de queries é a solução que mais agrada este autor. Ela já vem sendo utilizada durante o livro, só não tinha sido deixado claro que era por este motivo.

```
public class BookDAO {  
  
    @PersistenceContext  
    private EntityManager manager;
```



```
...

public List<Book> list() {
    return manager.createQuery("select b from Book b join
                               fetch b.authors",
                               Book.class).getResultList();
}
```

A parte negativa dessa solução é que ela aumenta consideravelmente a complexidade das queries que serão escritas no sistema, já que o tempo todo teremos de decidir exatamente o que precisa ser carregado para cada objeto.

Para balancear essa equação, podemos usar um outro recurso, já provido pela especificação. Primeiro, temos de deixar clara a nossa necessidade. Precisamos que o `EntityManager` fique aberto enquanto estivermos usando o objeto criado a partir da classe `ProductDetailBean`, uma vez que ele vive durante todo request.

A primeira atitude que podemos tomar é a de alterar a maneira que o `EntityManager` é associado ao objeto do tipo `BookDAO`. Atualmente, apesar de não termos configurado, ele é sempre ligado a uma transação.

```
public class BookDAO {

    @PersistenceContext(type=PersistenceContextType.TRANSACTION)
    private EntityManager manager;

    ...
}
```

Quase nunca especificamos o atributo `type`, já que o padrão é `PersistenceContextType.TRANSACTION` e, normalmente, é o que precisamos mesmo. Agora é necessário aumentar o seu tempo de vida para ele ir além de uma transação ou apenas da execução de um método.

```
public class BookDAO {

    @PersistenceContext(type=PersistenceContextType.EXTENDED)
    private EntityManager manager;
```

```
    ...
}
```

É justamente para isso que serve o `PersistenceContextType.EXTENDED`. Esse tipo de `EntityManager` deve ser injetado quando é necessário que os objetos carregados por ele permaneçam gerenciados enquanto o objeto que solicitou a injeção estiver sendo controlado pelo container. Isso já deveria ser o suficiente, uma vez que, como o nosso DAO não declarou nenhum escopo do CDI, é assumido que ele é `Dependent`. Como ele está sendo injetado dentro do `ProductDetailsBean`, deveria ficar vivo enquanto tal bean também estivesse.

Só que, apenas por uma questão da especificação, uma classe que declara a necessidade de um `EntityManager` estendido precisa também ser anotada com `@Stateful`.

```
@Stateful
public class BookDAO {

    @PersistenceContext(type=PersistenceContextType.EXTENDED)
    private EntityManager manager;

    ...
}
```

Classes anotadas com `@Stateful` automaticamente são tratadas pelo container como **Enterprise Java Beans**, e elas ficam vivas dentro do container enquanto o objeto que pediu sua injeção também estiver. Como, em geral, os beans do JSF vivem no escopo de requisição, os seus EJBs `stateful` também viverão.

Pronto, agora podemos acessar a página de detalhes, pois tudo deve funcionar normalmente. Resolver o problema do `LazyInitializationException` dessa forma não é uma abordagem normalmente encontrada, mas este autor acredita que ela é a mais simples e poderosa que temos.

EJB, O ANTIGO CENTRO DA ESPECIFICAÇÃO

Antes da chegada do CDI, os EJBs eram o centro da especificação. Quando era necessário usar os recursos providos pelo servidor de aplicação, era obrigatório transformar nosso objeto em um EJB. Hoje em dia, ainda vemos isso em alguns casos, mas a tendência é cada vez mais que simples objetos possam receber qualquer recurso fornecido pelo servidor.

7.4 SERÁ QUE O MELHOR É O DAO SER UM EJB STATEFUL?

A nossa solução, apesar de envolver muita teoria, ficou bem simples. Só tem um ponto que ainda pode incomodar. O DAO sabe que vai ser usado em um cenário que exige a extensão do ciclo de vida do `EntityManager`. Só que esse requisito tem muito mais relação com o nosso bean do JSF do que com DAO em si.

Um objeto dessa mesma classe já é usado em outros cenários na nossa aplicação, como por exemplo, para salvar um novo livro ou fazer a listagem. Para essas situações, não foi necessário usar o `PersistenceContextType.EXTENDED`.

Para tentarmos melhorar essa situação, podemos optar por uma solução não convencional. Preciso deixar claro para você, leitor, que: o que vou mostrar agora não é normalmente encontrado mundo a fora, mas me sinto na obrigação de mostrar uma estratégia que considero possuir um design melhor.

Como é o bean do JSF que está definindo o tempo de vida necessário do `EntityManager`, podemos transformá-lo em um EJB `stateful` em vez de DAO.

```
@Model
@Stateful
public class ProductDetailBean {
```

```
@PersistenceContext(type=PersistenceContextType.EXTENDED)
private EntityManager manager;

private BookDAO bookDAO;
private Book book;

@PostConstruct
private void loadManager(){
    this.bookDAO = new BookDAO(manager);
}

...
}
```

Perceba que, além dessa mudança, também paramos de injetar o `ProductDAO`. Agora, ele é criado manualmente no método anotado com `@PostConstruct`.

A ideia é que nós passamos para o DAO o `EntityManager` estendido. Dessa forma, você pode receber o DAO injetado automaticamente pelo container em situações comuns, mas também pode usar o mesmo DAO em situações que necessitem de um `EntityManager` estendido. O máximo que vai acontecer é você precisar instanciar alguns DAOs manualmente, mas nada que seja muito complicado.

Durante o restante do livro, será sugerido que você use essa abordagem, mas fique à vontade para refletir e usar a qual se encaixa melhor nos seus projetos.

Só precisamos alterar a classe `BookDAO` para ela também suportar o recebimento de um `EntityManager` pelo construtor.

```
public class BookDAO {

    @PersistenceContext
    private EntityManager manager;

    public BookDAO(){

    }

}
```

```
public BookDAO(EntityManager manager){  
    this.manager = manager;  
}
```

Como a annotation `@PersistenceContext` não pode ser usada em parâmetros, somos obrigados a manter a injeção via atributo privado. Também é necessário manter o construtor sem argumentos, para que o objeto seja instanciado e, posteriormente, a injeção feita.

O lado ruim dessa solução está justamente no fato de você ter de instanciar os DAOs na mão, além de receber um `EntityManager` que apenas é repassado para a frente. Como falei anteriormente, fique à vontade para refletir e decidir a maneira que melhor se encaixa nas suas necessidades.

7.5 FORMATANDO A DATA DE PUBLICAÇÃO

No fim da nossa página de detalhes, estamos exibindo a data de publicação do livro.

```
<p>Data de publicação: #{productDetailBean.book.releaseDate.time}
```

Por mais que funcione, a saída gerada não é exatamente a que esperamos. Para nos ajudar nesse tipo de cenário, podemos usar a tag `outputText` junto com a tag de conversão de datas.

```
<p>Data de publicação: <h:outputText  
    value="#{productDetailBean.book.releaseDate.time}">  
    <f:convertDateTime pattern="dd/MM/yyyy"/>  
</h:outputText></p>
```

A `outputText` simplesmente imprime o valor passado como argumento para o atributo `value`. O interessante é que podemos associar a tag `convertDateTime` e, com isso, o JSF já vai formatar o valor do jeito que precisamos. Essa é uma ótima prática, pois estamos usando o framework exatamente no ponto onde é necessário.

7.6 CARRINHO DE COMPRAS E O ESCOPO DE SESSÃO

Agora que já conseguimos navegar para a página de detalhes do livro, podemos começar a implementar a funcionalidade de adicioná-los ao carrinho de compras. Vamos apenas relembrar o trecho do `detalhe.xhtml` responsável por isso.

```
<form action="/cart/add" method="post" class="container">
  <ul id="variants" class="clearfix">
    <li class="buy-option">
      <input type="radio" name="id"
        class="variant-radio"
        id="product-variant-#{productDetailBean.book.id}"
        value="#{productDetailBean.book.id}"
        checked="checked" />

      <label class="variant-label"
        for="product-variant-#{productDetailBean
          .book.id}">
        E-book + Impresso
      </label>

      <small class="compare-at-price">
        R$ #{productDetailBean.book.price}
      </small>

      <p class="variant-price">
        R$ #{productDetailBean.book.price}
      </p>
    </li>
  </ul>

  <button type="submit" class="submit-image icon-basket-alt"
    title="#{productDetailBean.book.title}"></button>
</form>
```

O problema é que esse formulário não está ligado a nenhum `bean` da nossa aplicação. Para resolver isso, em vez de modificar o formulário inteiro para usar as tags do JSF, vamos utilizar uma nova funcionalidade que veio

com o JSF 2.2.

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

    ...

    <form jsf:id="form" method="post" class="container">
        <ul id="variants" class="clearfix">
            <li class="buy-option">
                <input type="radio" name="id"
                      class="variant-radio"
                      id="product-variant-#{productDetailBean
                                              .book.id}"
                      value="#{productDetailBean.book.id}"
                      checked="checked"/>

                <label class="variant-label"
                      for="product-variant-#{productDetailBean
                                              .book.id}">
                    E-book + Impresso
                </label>

                <small class="compare-at-price">
                    R$ #{productDetailBean.book.price}</small>
                <p class="variant-price">
                    R$ #{productDetailBean.book.price}</p>
            </li>
        </ul>

        <button type="submit"
              jsf:action="#{shoppingCartBean
                          .add(productDetailBean.id)}"
              class="submit-image icon-basket-alt"
              title="#{productDetailBean.book.title}"></button>
```

```
</form>
```

Perceba que usamos um novo *namespace*, o `xmlns:jsf="http://xmlns.jcp.org/jsf/T1/textquotedbl{}"`, que importa para a nossa página o recurso do **Pass Through Elements**. Estamos usando o prefixo `jsf` para referenciá-lo. Este recurso nos permite marcar alguns elementos que normalmente seriam simples componentes do HTML, como componentes que devem ser gerenciados pelo JSF.

```
<form jsf:id="form" method="post" class="container">
```

Só de usar um atributo prefixado com `jsf` já indicamos para o framework que ele deve interpretar esse elemento como um componente gerenciado. A parte bem interessante é que o JSF já sabe relacionar vários elementos do HTML com seus componentes do servidor. Por exemplo, um formulário já é associado com a classe `HtmlForm`. Usamos a mesma técnica quando usamos o botão dentro do formulário.

```
<button type="submit"
  jsf:action="#{shoppingCartBean.add(productDetailBean
    .book.id)}"
  class="submit-image icon-basket-alt"
  title="#{productDetailBean.book.title}"></button>
```

Usamos o atributo `action` em um elemento do tipo `button`, e isso só faz sentido porque estamos usando o prefixo `jsf`. Perceba que passamos como argumento justamente a referência para o método que deve ser chamado no nosso bean responsável por adicionar produtos no carrinho.

Este recurso veio para fechar uma lacuna importante que existia em aplicações escritas com o JSF. Muitos programadores reclamavam que não tinham controle sobre o HTML gerado, o que, a partir desta última versão, deixou de ser verdade. Você agora pode montar uma página inteira com HTML puro e só gerenciar os componentes que você realmente precisa.

Lógica para adicionar os produtos no carrinho

O atributo `action` do nosso botão está apontando para o método `add` do bean `ShoppingCartBean`. Vamos dar uma olhada em como está o código referente a esta parte do sistema.

```
package br.com.casadocodigo.loja.managedbeans.site;

//imports

@Model
public class ShoppingCartBean {

    @Inject
    private ShoppingCart shoppingCart;
    @Inject
    private BookDAO bookDAO;

    public String add(Integer id){
        Book book = bookDAO.findById(id);
        ShoppingItem item = new ShoppingItem(book);
        shoppingCart.add(item);
        return "/site/carrinho?faces-redirect=true";
    }

    public String remove(Integer id){
        Book book = bookDAO.findById(id);
        ShoppingItem item = new ShoppingItem(book);
        shoppingCart.remove(item);
        return "/site/carrinho?faces-redirect=true";
    }
}
```

O código em si não tem nada de novo. Apenas usamos outra classe com o objetivo de manter os produtos adicionados ao nosso carrinho. Vamos dar uma olhada nela:

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

```
public class ShoppingCart implements Serializable{

    /**
     *
     */
    private static final long serialVersionUID =

    private Map<ShoppingItem, Integer> items =
        new LinkedHashMap<ShoppingItem, Integer>();

    public void add(ShoppingItem item) {
        items.put(item, getQuantity(item) + 1);
    }

    public Integer getQuantity(ShoppingItem item) {
        if (!items.containsKey(item)) {
            items.put(item, 0);
        }
        return items.get(item);
    }

    public Integer getQuantity() {
        return items.values().stream()
            .reduce(0, (next, accumulator) -> next
                + accumulator);
    }

    public Collection<ShoppingItem> getList() {
        return new ArrayList<>(items.keySet());
    }

    public BigDecimal getTotal(ShoppingItem item) {
        return item.getTotal(getQuantity(item));
    }

    public BigDecimal getTotal(){
        BigDecimal total = BigDecimal.ZERO;

        for(ShoppingItem item : items.keySet()){
```

```
        total = total.add(getTotal(item));
    }
    return total;
}

public void remove(ShoppingItem shoppingItem) {
    items.remove(shoppingItem);
}

public boolean isEmpty() {
    return items.isEmpty();
}

}
```

Além dessa classe, também precisamos da classe `ShoppingItem`.

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

//imports

public class ShoppingItem {

    private Book book;
    // apenas para ficar mais facil de gerar
    // o hashCode e equals
    private Integer bookId;

    public ShoppingItem(Book book) {
        this.book = book;
        this.bookId = book.getId();
    }

    public Book getBook() {
        return book;
    }

    public BigDecimal getPrice() {
        return book.getPrice();
    }
}
```

```
public Integer getBookId() {
    return bookId;
}

public BigDecimal getTotal(Integer quantity) {
    return getPrice().multiply(new BigDecimal(quantity));
}

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

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

Os códigos completos dessas classes podem ser encontrados em <http://bit.ly/carrinho-casadocodigo>.

O leitor mais atento deve ter percebido que, após adicionar um produto no carrinho, fazemos um *redirect* para uma tela cuja responsabilidade é justamente exibir todos os itens adicionados.

```
@Model
public class ShoppingCartBean {

    @Inject
    private ShoppingCart shoppingCart;
    @Inject
    private BookDAO bookDAO;

    public String add(Integer id){
        Book book = bookDAO.findById(id);
        ShoppingItem item = new ShoppingItem(book);
        shoppingCart.add(item);
        return "/site/carrinho?faces-redirect=true";
    }

    ...
}
```

Como ainda não temos essa página pronta, chegou a hora de implementarmos. De novo, como ela é um tanto quanto grande, vamos apenas colocar os trechos importantes para a nossa leitura. A localização do arquivo deve ser `site/carrinho.xhtml`.

```
<form jsf:id="cartForm" method="post">
    <table id="cart-table">

        ...
        <tbody>

            <ui:repeat var="item" value="#{shoppingCart.list}">
                <tr>
                    <td class="cart-img-col">
                        
                    </td>
                </tr>
            </ui:repeat>
        </tbody>
    </table>
</form>
```

```

        <td class="item-title">#{item.book.title}</td>

        <td class="numeric-cell">R$ #{item.price}</td>

        <td class="quantity-input-cell">
            <input type="number" min="0"
                readonly="readonly"
                id="update
                name="upda
                value="#{shoppingCart.getQuantity(item)}"/>
        </td>

        <td class="numeric-cell">
            R$ #{shoppingCart.getTotal(item)}
        </td>

        <td class="remove-item">
            <a jsf:action="#{shoppingCartBean
                .remove(item.bookId)}">
                
            </a></td>
    </tr>
</ui:repeat>

</tbody>
<tfoot>
    <tr>
        <td colspan="3"><input type="submit" class="checkout"
            name="checkout" value="Finalizar compra "
            id="checkout"/>
        </td>
        <td class="quantity-input-cell"><input type="submit"
            class="update-cart" name="update" value=""/>
        </td>
        <td class="numeric-cell">R$ #{shoppingCart.total}</td>
        <td></td>
    </tr>
</tfoot>
</table>

```

```
</form>
```

Montamos novamente um HTML limpo, e só adicionamos alguns atributos do JSF para transformar alguns elementos em componentes gerenciados pelo framework. Um detalhe interessante foi que usamos a variável de nome `shoppingCart`, fazendo clara referência à nossa classe `ShoppingCart`.

Você deve estar se perguntando: *quer dizer que, independente da classe que eu crie, os objetos criados a partir dela estarão disponíveis via Expression Language?*

Caso isso fosse verdade, nossos DAOs que são instanciados via CDI estariam disponíveis na página e teríamos uma quebra de camadas muito perigosa. Para controlar quem deve ficar disponível ou não, podemos usar a anotação `@Named`. Além disso, ainda precisamos manter esse objeto vivo durante vários requests do mesmo usuário, contexto também conhecido como *Session*. Precisamos manter o objeto na *session*, uma vez que nosso usuário pode adicionar vários itens no carrinho.

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

//outros imports

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

@Named
@SessionScoped
public class ShoppingCart implements Serializable{

    ...

}
```

Implementamos `Serializable` apenas para deixar claro que o estado de um objeto desta classe pode ser persistido. Agora nosso usuário já pode adicionar os itens no carrinho e visualizar a lista de produtos que vão ser comprados.

7.7 CONCLUSÃO

Neste capítulo, implementamos várias funcionalidades da Casa do Código. Passamos por algumas dificuldades que foram importantes, já que nos revelaram novas técnicas. Por exemplo, usamos o **Pass Through Element** para manter nosso HTML fiel ao site e só adicionar na árvore do JSF alguns elementos. Além disso, também trabalhamos com a especificação dos EJBs, pois usamos a annotation `Stateful` para conseguir manter um `EntityManager` estendido.

Minha dica é que você respire um pouco, tente implementar ou digerir tudo o que foi exibido até aqui e aí, depois, você continua a leitura. Ainda tem muito coisa boa por vir. Já no próximo capítulo vamos nos integrar com outro sistema e trabalhar com execuções de códigos assíncronos.

CAPÍTULO 8

Fechamento da compra e processamento assíncrono

No capítulo anterior, começamos a implementar o processo de fechamento de compra da loja da Casa do Código, e já conseguimos concluir a parte de adicionar os itens em um carrinho de compras. Agora, precisamos realmente finalizar um novo pedido dentro da loja.

8.1 IMPLEMENTANDO A TELA DE FINALIZAÇÃO

A versão de tela de finalização de compra que vamos implementar deve ficar próxima da figura a seguir:

Fig. 8.1: Tela de checkout da Casa do Código

Primeiro, precisamos alterar um pouco nosso código da tela do carrinho para adicionar um link que nos leve para o endereço de finalização de compra.

```
<tfoot>
  <tr>
    <td colspan="3">
      <a href="{request.contextPath}/site/checkout.xhtml"
        id="checkout" class="checkout">Finalizar compra
      </a></td>
    ...
  </tr>
</tfoot>
```

Simplesmente adicionamos um link que levará o usuário para a tela de *checkout*. Por sinal, essa vai ser mais uma página na qual usaremos bastante o novo recurso do **Pass Through Elements**.

Basicamente, vamos ter o HTML já definido pela Casa do Código, e apenas serão modificados os elementos que devem ser gerenciados pelo JSF. O arquivo responsável por montar a tela de checkout deve ficar em `site/checkout.xhtml`.

Como já vem sendo feito, vamos apenas analisar as partes importantes do código. O arquivo completo pode ser encontrado em <http://bit.ly/checkout-casadocodigo>.

```
<form accept-charset="UTF-8" jsf:prependId="false"
  class="section__form
  js-shipping-address-with-selector"
  jsf:id="checkoutUserForm" method="post"
  <div style="display:none"></div>

  <div class="fieldset"
    data-email-check='#checkout_email_suggestion'>
    <div class="field field--required">
      <label for="checkout_email">E-mail</label>
      <input autocomplete="off"
        autocomplete="shipping email"
        jsf:id="email" data-autofocus="true"
        jsf:value="#{checkoutBean.systemUser.email}"
        placeholder="E-mail" size="30"
        spellcheck="false"
        type="email"/>
    </div>

    ...
  </div>

<h5>Endereço de entrega</h5>

<div class="fieldset" id="shipping-address"
  data-shipping-address="">
  <div class="field field--optional">
    <label for=
      "checkout_shipping_address_first_name">
    </label>
    Nome
    <input autocomplete="shipping given-name"
```

```
jsf:id="name"
jsf:value="#{checkoutBean.systemUser
    .firstName}"
placeholder="Nome" size="30"
type="text"/>

</div>
<div class="field field--required">
    <label for=
        "checkout_shipping_address_last_name">
        Sobrenome</label>
    <input autocomplete="shipping family-name"
        jsf:id="lastName"
        jsf:value="#{checkoutBean.systemUser
            .lastName}"
        placeholder="Sobrenome" size="30"
        type="text"/>
</div>

<div class="field field--required">
    <label for=
        "checkout_shipping_address_company">
        CPF/CNPJ
    </label>
    <input autocomplete="shipping organization"
        jsf:id="socialId"
        jsf:value="#{checkoutBean.systemUser
            .socialId}"
        placeholder="999.999.999-99" size="30"
        type="text">
</div>

...

<div class="field field--required"
    data-country-section="1">
    <label for=
        "checkout_shipping_address_country">
        País
```

```
</label>
<select
    jsf:value="#{checkoutBean.systemUser
                .country}"
    jsf:id="country" size="1">
    <f:selectItem itemValue=""
        itemLabel="Escolha o país"/>
    <f:selectItem itemValue="Portugal"
        itemLabel="Portugal"/>
    <f:selectItem itemValue="Brasil"
        itemLabel="Brasil"/>
    <f:selectItem itemValue="Africa do Sul"
        itemLabel="Africa do Sul"/>
    <f:selectItem itemValue="Argentina"
        itemLabel="Argentina"/>
    <f:selectItem itemValue="Colombia"
        itemLabel="Colombia"/>
</select>
</div>
```

É um formulário com muitos campos, e aqui só exibimos alguns. A parte interessante é que conseguimos manter o HTML original, e apenas adicionamos alguns atributos específicos do JSF para que o framework entenda que os inputs e o formulário em si devam ser tratados como componentes. Um que nos deu um pouco mais de trabalho foi o `select`.

```
<select jsf:value="#{checkoutBean.systemUser.country}"
    jsf:id="country" size="1">
    <f:selectItem itemValue="" itemLabel="Escolha o país"/>
    <f:selectItem itemValue="Portugal" itemLabel="Portugal"/>
    <f:selectItem itemValue="Brasil" itemLabel="Brasil"/>
    <f:selectItem itemValue="Africa do Sul"
        itemLabel="Africa do Sul"/>
    <f:selectItem itemValue="Argentina" itemLabel="Argentina"/>
    <f:selectItem itemValue="Colombia" itemLabel="Colombia"/>
</select>
```

O JSF sabe apenas que uma tag `select`, marcada com um atributo especial, deve virar um componente do tipo `HtmlSelectOneListBox`, mas

não tem ideia do que fazer com os elementos `option` que, pensando apenas em HTML, deveriam vir dentro de um `select`. Por conta dessa limitação, somos obrigados a usar a tag `selectItem`, o que realmente não impacta muito.

Outro detalhe bem interessante é que a usamos como filha de um `select`, justamente para deixar claro que o fato de marcar um elemento HTML com um atributo JSF realmente o faz ser gerenciado e entrar na árvore de componentes.

8.2 GRAVANDO AS INFORMAÇÕES DO USUÁRIO

O leitor mais atento já deve ter percebido o uso de um novo `bean` para lidar com a finalização do pedido. O `value` de todos os inputs estão apontando para `#{checkoutBean...}`, que é justamente esse novo bean. A primeira tarefa que vamos tratar é a de gravar as informações do novo comprador.

Apenas para refrescar sua memória, vamos olhar um dos inputs do formulário mais uma vez:

```
<input autocapitalize="off" autocomplete="shipping email"
      jsf:id="email" data-autofocus="true"
      jsf:value="#{checkoutBean.systemUser.email}"
      placeholder="E-mail" size="30" spellcheck="false"
      type="email"/>
```

...

```
<input class="btn" name="commit"
      jsf:action="#{checkoutBean.checkout()}"
      type="submit" value="Continuar" />
```

Perceba que ele está associado aos métodos de acesso que retornam um objeto que representa um usuário. Além disso, o botão de finalização deve fazer uso do método `checkout` na mesma classe. Vamos dar uma olhada para entender um pouco mais do código.

```
package br.com.casadocodigo.loja.managedbeans.site;
```

```
//imports

@Model
public class CheckoutBean {

    private SystemUser systemUser = new SystemUser();
    @Inject
    private SystemUserDAO systemUserDAO;
    @Inject
    private ShoppingCart cart;

    public SystemUser getSystemUser() {
        return systemUser;
    }

    public void setSystemUser(SystemUser systemUser) {
        this.systemUser = systemUser;
    }

    @Transactional
    public void checkout() throws IOException {
        systemUserDAO.save(systemUser);

        //vamos também gravar a compra
        //aprovar com um sistema externo
    }
}
```

A classe é um `bean` normal, nada que não tenhamos trabalhado até agora. Para que tudo funcione, precisamos apenas implementar a parte relativa ao usuário. Começaremos pela classe `SystemUser`.

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

//imports

@Entity
```

```
public class SystemUser {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Email
    @NotBlank
    @Column(unique=true)
    private String email;
    @NotBlank
    private String firstName;
    @NotBlank
    private String lastName;
    @NotBlank
    @Column(unique=true)
    private String socialId;
    @NotBlank
    private String address;
    @NotBlank
    private String city;
    @NotBlank
    private String state;
    @NotBlank
    private String postalCode;
    @NotBlank
    private String phone;
    @NotBlank
    private String country;
    private String password;

    //metodos de acesso
```

É simplesmente uma classe mapeada com annotations da JPA e da Bean Validation. Para fechar, é necessário que tenhamos o DAO responsável por gravar um novo usuário no banco de dados.

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

//imports
```



```
public class SystemUserDAO {  
  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    public void save(SystemUser systemUser) {  
        entityManager.persist(systemUser);  
    }  
  
}
```

Isso já é suficiente para gravarmos o usuário que está comprando um novo produto. Poderíamos ter verificado, por exemplo, se o e-mail já existe para não cadastrar de novo. Fica como desafio para o leitor incrementar ainda mais a aplicação.

Ajuda extra com o OmniFaces

Por conta da integração do JSF com a Bean Validation, nosso formulário inclusive já está sendo validado. Basta adicionarmos a tag `message` embaixo de cada input para exibirmos as mensagens de erro.

```
<input autocapitalize="off" autocomplete="shipping email"  
    jsf:id="email" data-autofocus="true"  
    jsf:value="#{checkoutBean.systemUser.email}"  
    placeholder="E-mail" size="30" spellcheck="false"  
    type="email"/>  
  
<p class="field_error-message"><h:message for="email"/>  
</p>
```

Esse código até funciona e não influencia diretamente no layout da aplicação, mas de todo jeito estamos exibindo o `<p>` sempre em vez de só exibir quando realmente existir um problema. Para contornarmos isso, teríamos de verificar se alguma mensagem foi adicionada ou não.

Com a intenção de ajudar o desenvolvedor que utiliza o JSF no dia a dia, Bauke Scholtz criou uma biblioteca chamada **OmniFaces**. A ideia desse pro-

jeto é oferecer classes e tags extras que realmente possam ajudar enquanto estamos desenvolvendo uma aplicação baseada em JSF.

Segue uma pequena lista de utilidades fornecidas pelo OmniFaces:

- Novos validadores;
- Mais facilidades com AJAX;
- Formataadores mais elaborados;
- Novos componentes.

Para o nosso caso, vamos usar uma versão diferente da tag `messages`, uma extensão provida pelo OmniFaces que se aplica muito bem para a nossa situação.

```
<html xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:pt="http://xmlns.jcp.org/jsf/passthrough"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:jsf="http://xmlns.jcp.org/jsf"
      xmlns:o="http://omnifaces.org/ui">

  <input autocapitalize="off" autocomplete="shipping email"
        jsf:id="email" data-autofocus="true"
        jsf:value="#{checkoutBean.systemUser.email}"
        placeholder="E-mail" size="30" spellcheck="false"
        type="email"/>

  <o:messages for="email" var="message">
    <p class="field_error-message">
      #{message.summary}</p>
  </o:messages>
```

Tivemos de importar o namespace `http://omnifaces.org/ui`. Perceba que agora usamos a tag `messages` associada ao prefixo `o:.` Essa tag só exibe o

conteúdo entre ela caso exista alguma mensagem associada ao componente referenciado pelo atributo `for`.

Não precisamos colocar todo o código do formulário aqui, use a mesma tag para exibir as mensagens de erros dos outros campos. Fique apenas atento ao `id` que você vai utilizar no atributo `for`.

Precisamos adicionar o OmniFaces ao nosso projeto. Para isso, basta que seja adicionada uma nova dependência no nosso arquivo `pom.xml`.

```
<project ....>
...

<dependencies>
...

<!-- omni -->

<dependency>
  <groupId>org.omnifaces</groupId>
  <artifactId>omnifaces</artifactId>
  <version>2.1</version>
</dependency>

...

</dependencies>

...
</project>
```

Tente não confundir o OmniFaces com uma biblioteca, como o PrimeFaces. Enquanto o último tenta trazer novos componentes de layout para o nosso código, o primeiro foca em classes e tags utilitárias.

8.3 VALIDAÇÃO SELETIVA COM A BEANVALIDATION

Para representar o usuário, foi usada a classe `SystemUser`. Um possível problema que podemos ter é que, mais à frente, quando adicionarmos a parte de segurança na aplicação, teremos dois tipos de usuários: um com todas as

informações preenchidas, representando tipicamente um comprador da Casa do Código; e outro que só precisará do e-mail e da senha, já que será um usuário com papel de administrador.

Analisando a classe `SystemUser`, percebemos que adicionamos as regras de validações em quase todos os atributos.

```
@Entity
public class SystemUser {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Email
    @NotBlank
    @Column(unique=true)
    private String email;
    @NotBlank
    private String firstName;
    @NotBlank
    private String lastName;
    @NotBlank
    @Column(unique=true)
    private String socialId;
    @NotBlank
    private String address;
    @NotBlank
    private String city;
    @NotBlank
    private String state;
    @NotBlank
    private String postalCode;
    @NotBlank
    private String phone;
    @NotBlank
    private String country;
    private String password;
```

E quando tivermos de cadastrar um novo usuário que é administrador? Como vamos fazer para ensinar a implementação da JPA que ela não deve

validar todas essas informações?

Mantendo as annotations e gerando as tabelas por meio do próprio Hibernate, inclusive as colunas do banco são geradas não aceitando valores nulos.

Pensando justamente neste tipo de situação, a Bean Validation já previu que você pode ter **grupos de validação**.

```
@Entity
public class SystemUser {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Email
    @NotBlank
    @Column(unique=true)
    private String email;
    @NotBlank(groups=BuyerGroup.class)
    private String firstName;
    @NotBlank(groups=BuyerGroup.class)
    private String lastName;
    @NotBlank(groups=BuyerGroup.class)
    private String socialId;
    @NotBlank(groups=BuyerGroup.class)
    private String address;
    @NotBlank(groups=BuyerGroup.class)
    private String city;
    @NotBlank(groups=BuyerGroup.class)
    private String state;
    @NotBlank(groups=BuyerGroup.class)
    private String postalCode;
    @NotBlank(groups=BuyerGroup.class)
    private String phone;
    @NotBlank(groups=BuyerGroup.class)
    private String country;
    private String password;
```

O atributo `groups` permite que você passe a referência para uma interface que define o grupo que pertence a alguma validação. Perceba que deixamos apenas os atributos `email` e `password` sem grupo algum. A

interface que define o grupo não precisa ter método algum, é a típica **interface de marcação**.

```
package br.com.casadocodigo.loja.models.validation.groups;

public interface BuyerGroup {

}
```

Como o JSF já é integrado com a *Bean Validation*, podemos dizer que cada input do formulário deve ter sua validação associada a um grupo específico. Vamos analisar novamente o arquivo `checkout.xhtml`:

```
<form accept-charset="UTF-8" jsf:prependId="false"
class="section__form js-shipping-address-with-selector"
jsf:id="checkoutUserForm" method="post">
    <div style="display:none"></div>

    <div class="fieldset"
        data-email-check='#checkout_email_suggestion'>
        <div class="field field--required">
            <label for="checkout_email">E-mail</label>
            <input autocapitalize="off"
                autocomplete="shipping_email"
                jsf:id="email" data-autofocus="true"
                jsf:value="#{checkoutBean.systemUser.email}"
                placeholder="E-mail" size="30"
                spellcheck="false" type="email">

            <f:validateBean
                validationGroups="br.com.casadocodigo.loja
                    .models.validation.groups.BuyerGroup"/>

            </input>
        </div>

        ...
    </div>
```

```
<h5>Endereço de entrega</h5>
```

```
<div class="fieldset" id="shipping-address"
    data-shipping-address="">
  <div class="field field--optional">
    <label for=
      "checkout_shipping_address_first_name">
      Nome
    </label>
    <input autocomplete="shipping given-name"
      jsf:id="name"
      jsf:value="#{checkoutBean.systemUser.firstName}"
      placeholder="Nome" size="30" type="text">

    <f:validateBean
      validationGroups="br.com.casadocodigo.loja
        .models.validation.groups.BuyerGroup"/>

  </input>

</div>
<div class="field field--required">
  <label for="checkout_shipping_address_last_name">
    Sobrenome</label>
  <input autocomplete="shipping family-name"
    jsf:id="lastName"
    jsf:value="#{checkoutBean.systemUser.lastName}"
    placeholder="Sobrenome" size="30" type="text">

  <f:validateBean
    validationGroups="br.com.casadocodigo.loja
      .models.validation.groups.BuyerGroup"/>

  </input>
</div>

<div class="field field--required">
  <label for="checkout_shipping_address_company">
```

```
CPF/CNPJ
</label>
<input autocomplete="shipping organization"
  jsf:id="socialId"
  jsf:value="#{checkoutBean.systemUser
              .socialId}"
  placeholder="999.999.999-99" size="30"
  type="text">

  <f:validateBean
    validationGroups="br.com.casadocodigo.loja
                     .models.validation.groups.BuyerGroup"/>
</input>
</div>

...

<div class="field field--required"
  data-country-section="1">
  <label for="checkout_shipping_address_country">
    País
  </label>
  <select
    jsf:value="#{checkoutBean.systemUser.country}"
    jsf:id="country" size="1">
    <f:validateBean
      validationGroups="br.com.casadocodigo.loja
                       .models.validation.groups.BuyerGroup"/>
    <f:selectItem itemValue=""
      itemLabel="Escolha o país"/>
    <f:selectItem itemValue="Portugal"
      itemLabel="Portugal"/>
    <f:selectItem itemValue="Brasil"
      itemLabel="Brasil"/>
    <f:selectItem itemValue="Africa do Sul"
      itemLabel="Africa do Sul"/>
    <f:selectItem itemValue="Argentina"
      itemLabel="Argentina"/>
    <f:selectItem itemValue="Colombia"
```



```
        itemLabel="Colombia"/>
    </select>
</div>

...

```

Devemos fazer isso para todos os inputs que devam ser validados quando uma compra na loja estiver acontecendo. De novo, estamos usando o poder do **Pass Through Elements**. Transformamos um simples `input` em um componente do JSF e, com isso, ganhamos o direito de associar outros componentes do framework a esse elemento. Os grupos de validação são previstos na especificação; não tenha medo de usá-los.

8.4 SALVANDO AS INFORMAÇÕES DO CHECKOUT

Já estamos salvando as informações do usuário quando ele clica no botão para fechar o pedido. O próximo passo é armazenar todas as informações da compra em si.

```
@Model
@Path("/payment")
public class CheckoutBean {

    private SystemUser systemUser = new SystemUser();
    @Inject
    private SystemUserDAO systemUserDAO;
    @Inject
    private CheckoutDAO checkoutDAO;
    @Inject
    private ShoppingCart cart;

    public SystemUser getSystemUser() {
        return systemUser;
    }

    public void setSystemUser(SystemUser systemUser) {
        this.systemUser = systemUser;
    }
}

```

```

@Transactional
public void checkout() throws IOException {
    systemUserDAO.save(systemUser);

    Checkout checkout = new Checkout(systemUser, cart);
    checkoutDAO.save(checkout);
}
}

```

Para que o código anterior compile, é necessário que seja criada a classe `Checkout`, além do DAO. O código do DAO não tem nada novo, já que simplesmente precisamos salvar o objeto.

```

package br.com.casadocodigo.loja.daos;

public class CheckoutDAO {

    @PersistenceContext
    private EntityManager entityManager;

    public void save(Checkout checkout) {
        entityManager.persist(checkout);
    }

}

```

Já o código da classe `Checkout` precisa de um certo momento de reflexão. Precisamos gravar as seguintes informações:

- Usuário associado à compra;
- Valor total do carrinho;
- Os produtos associados à compra.

Uma ideia inicial seria, então, ter classes parecidas com as seguintes:

```

@Entity
public class Checkout {

```

```
@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
private Integer id;
@ManyToOne
private SystemUser buyer;
private BigDecimal value;
private List<Item> items = new ArrayList<>();

...
}

@Entity
public class FinalShoppingItem {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    @ManyToOne
    private Book book;
    private int quantity;
    private BigDecimal currentPrice;

    ...
}
```

Esse modelo até funciona, mas para o nosso caso é suficiente guardar uma representação mais simples do carrinho, apenas para consultas futuras. Então, em vez de modelar desse jeito, podemos fazer o seguinte:

```
@Entity
public class Checkout {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    @ManyToOne
    private SystemUser buyer;
    private BigDecimal value;
    private String jsonCart;

    //construtor para ser usado pelos frameworks
```

```
protected Checkout() {

}

public Checkout(SystemUser user, ShoppingCart cart) {
    this.buyer = user;
    this.value = cart.getTotal();
    this.jsonCart = cart.toJson();
}
}
```

A ideia é guardarmos as informações do carrinho como um JSON, um formato muito conhecido de representação de dados. Inclusive, os bancos mais conhecidos já suportam que dados sejam armazenados neste formato. Acesse o link <http://bit.ly/json-mysql> e leia um pouco mais.

Adicionamos o método `toJson` dentro da classe `ShoppingCart`.

```
public class ShoppingCart {
    ...

    public String toJson() {
        //como vamos gerar o JSON?
    }
}
```

Usando a nova especificação de geração de JSON

A JSR 353 (<https://www.jcp.org/en/jsr/detail?id=353>) define uma das novas especificações do Java EE 7, a **Java API for JSON Processing**. Isso quer dizer que você já tem suporte à geração de JSON dentro de qualquer servidor que implemente a última versão da *spec*. Isso se encaixa perfeitamente para as nossas necessidades!

```
public String toJson() {
    JsonArrayBuilder itens = Json.createArrayBuilder();
    for (ShoppingItem item : getList()) {
        itens.add(Json.createObjectBuilder()
            .add("title", item.getBook().getTitle())
            .add("price", item.getBook().getPrice())
```

```
        .add("quantity", getQuantity(item).intValue())
        .add("sum", getTotal(item));

    }
    return itens.build().toString();
}
```

Perceba que a montagem do JSON é um pouco manual, mas, em compensação, nos permite de forma clara decidir exatamente o que precisamos que seja gerado. Vamos dar uma olhada em cada das classes envolvidas.

A classe `Json` é seu ponto de partida. Ela possui os métodos necessários para o início da criação do seu JSON. Por exemplo, no nosso caso, foi preciso pedir pela construção de um `Array`, pois precisamos guardar todos os itens do carrinho. Além disso, de dentro do `loop`, é necessário criar uma representação para cada item do nosso carrinho. Quando queremos criar apenas um objeto JSON, usamos o método `createObjectBuilder` e vamos adicionando as informações necessárias. O resultado gerado é parecido com o seguinte:

```
[
  {
    "title": "javaee",
    "price": 100,
    "quantity": 1,
    "sum": 100
  },
  {
    "title": "spring mvc",
    "price": 100,
    "quantity": 1,
    "sum": 100
  }
]
```

Estamos quase lá. O nosso pedido já é gerado e associado com o novo comprador. Agora vamos, por enquanto, para a parte final do nosso processo. Precisamos validar o valor da nossa compra em um sistema externo!

JSON-B

No Java EE 8 será disponibilizado a especificação JSON-B (*Java API for JSON Binding*), que vai nos possibilitar transformar objetos em JSON e sair de JSON para objetos, de forma automática.

8.5 INTEGRANDO COM OUTRA APLICAÇÃO

Para a aplicação ficar ainda mais parecida com uma da vida real, vamos ter de nos integrar com um sistema externo para validar o valor final da compra.

A lógica é simples: só aceitamos pagamentos de até R\$500,00 reais. Somos obrigados a passar os dados usando o JSON, e vamos usar o HTTP como protocolo de integração.

Caso o valor tenha seguido a regra, é retornado o status *201*, que indica que um novo recurso foi criado, neste caso, o pagamento. Por outro lado, se tiver sido passado um valor maior que 500, é retornado o status *400*, o que indica que a requisição possui dados inválidos.

Caso você queira efetuar um teste e esteja usando Linux ou Mac, basta executar um `curl`.

```
curl --header "Content-type: application/json" --request POST
--data '{"value": 600}'
http://book-payment.herokuapp.com/payment
```

Um outro jeito de verificar é usando um plugin do Chrome, o **Postman**.

JAX-RS client

Vamos precisar de uma biblioteca que nos ajude a realizar a requisição HTTP, para que nossa integração funcione. Para a nossa sorte, na última versão do Java EE, foi adicionado o suporte a este tipo de cenário na especificação JAX-RS. Vamos lidar mais diretamente com essa especificação em um capítulo mais para a frente, onde teremos que expor os dados do nosso sistema em outros formatos.

Para não perder muito tempo, vamos ver como ficaria o código final da finalização do checkout.

```
@Model
public class CheckoutBean {
    ...

    @Transactional
    public void checkout() throws IOException {
        systemUserDAO.save(systemUser);

        Checkout checkout = new Checkout(systemUser, cart);
        checkoutDAO.save(checkout);

        //código de integração
        Client client = ClientBuilder.newClient();
        PaymentData paymentData = new PaymentData(total);
        String uriToPay =
            "http://book-payment.herokuapp.com/payment";
        Entity<PaymentData> json = Entity.json(paymentData);
        WebTarget target = client.target(uriToPay);
        Builder request = target.request();
        request.post(json, String.class);
    }
}
```

Respire um pouco e vamos pensar sobre o trecho de código que acabamos de escrever.

- 1) Criamos o objeto do tipo `Client` para fazer a requisição;
- 2) Geramos o JSON de um objeto por meio da classe `Entity`;
- 3) Definimos o endereço da requisição por meio do método `target`;
- 4) Construímos a requisição em si, com o método `request`;
- 5) Disparamos um `post` com o JSON e informamos que simplesmente esperamos um String de resposta.

A API de fazer requisições ficou com um design bem interessante. Os passos são bem marcados e um depende do outro. O trecho final, em que montamos a requisição em si, poderia ter sido escrito de uma forma mais sucinta.

```
client.target(uriToPay).request().post(json, String.class);
```

Outro detalhe que pode chamar a atenção é a utilização da classe `PaymentData`. Qual o motivo de necessitarmos dela? O sistema de pagamento com o qual estamos realizando a integração pede que o formato do dado passado seja o seguinte:

```
{"value": 600}
```

É como se fosse um `Map`, precisamos de uma chave chamada `value` associada a um valor qualquer. Caso passemos o `BigDecimal` direto, qual seria o nome dessa chave? É justamente por isso que precisamos criar uma classe.

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

import java.math.BigDecimal;

public class PaymentData {

    private BigDecimal value;

    public PaymentData() {
    }

    public PaymentData(BigDecimal value) {
        this.value = value;
    }

    public BigDecimal getValue() {
        return value;
    }

}
```


Repare que temos um atributo que se chama exatamente `value`, justamente para que o `JSON` seja gerado com as chaves corretas.

Outro ponto em que precisamos ter atenção é para a semântica do código. Para uma pessoa que não conhece o sistema, o que esse conjunto de linhas quer dizer?

```
@Transactional
public void checkout() throws IOException {
    ...

    //código de integração
    Client client = ClientBuilder.newClient();
    PaymentData paymentData = new PaymentData(total);
    String uriToPay =
        "http://book-payment.herokuapp.com/payment";
    Entity<PaymentData> json = Entity.json(paymentData);
    WebTarget target = client.target(uriToPay);
    Builder request = target.request();
    request.post(json, String.class);
}
```

O que estamos fazendo é aprovando um pagamento com um sistema externo, mas nosso código contém apenas um monte de linhas usando uma API. Para melhorar, vamos deixá-lo da seguinte forma.

```
@Model
public class CheckoutBean {
    ...

    @Inject
    private PaymentGateway paymentGateway;

    @Transactional
    public void checkout() throws IOException {
        systemUserDAO.save(systemUser);

        Checkout checkout = new Checkout(systemUser, cart);
        checkoutDAO.save(checkout);
    }
}
```

```
        paymentGateway.pay(total);  
    }  
}
```

Agora, basta que a classe `PaymentGateway` seja criada.

```
package br.com.casadocodigo.loja.services;  
  
import java.math.BigDecimal;  
  
import javax.ws.rs.client.Client;  
import javax.ws.rs.client.ClientBuilder;  
import javax.ws.rs.client.Entity;  
  
public class PaymentGateway {  
  
    public PaymentGateway() {  
    }  
  
    public void pay(BigDecimal total) {  
        Client client = ClientBuilder.newClient();  
        PaymentData paymentData = new PaymentData(total);  
        String uriToPay =  
            "http://book-payment.herokuapp.com/payment";  
        Entity<PaymentData> json = Entity.json(paymentData);  
        client.target(uriToPay).request()  
            .post(json, String.class);  
    }  
}
```

8.6 EXECUTANDO OPERAÇÕES DEMORADAS ASSINCRONAMENTE

Agora que nosso código de integração já funciona, vamos tentar pensar um pouco adiante. A Casa do Código é uma empresa de sucesso, o Brasil inteiro compra livros de tecnologia nela. Em certos momentos do ano, como na *Black Friday*, ela libera certas promoções que fazem com que o site fique muito movimentado e, com isso, temos muitos usuários concluindo compras ao mesmo tempo. Algo similar acontece em todas as outras empresas.

Não é raro nessas situações de uso intenso que os sistemas fiquem um pouco mais lentos, ou até que caiam e fiquem fora do ar por um tempo, o que pode gerar um prejuízo grande. Em geral, todos os sistemas têm seus pontos de gargalo e, como não poderia ser diferente, nós temos o nosso também.

Já se perguntou quanto tempo pode levar para a nossa aplicação fazer uma requisição para o sistema de pagamento? Caso esse outro sistema demore, o que pode acontecer com o nosso site?

Neste exato momento, o funcionamento é basicamente o seguinte: quando uma nova requisição chega ao nosso servidor web, ele faz um tratamento inicial e delega a responsabilidade para a Servlet configurada, nesse caso, a do JSF. A `FacesServlet` vai descobrir quais beans devem ser acionados para tratar a requisição, e fazer o trabalho para que isso realmente aconteça.

Todo esse fluxo acontece dentro de uma Thread criada pelo servidor web, para que ele possa tratar várias requisições simultaneamente. Essa Thread só é liberada quando o método do nosso bean acaba de fazer o trabalho e indica para qual endereço devemos ir.

Esse modelo de trabalho, em que alguém fica esperando o outro acabar para aí sim continuar seu trabalho, é conhecido como modelo **síncrono**. O problema desse modelo é que o servidor web tem um número limitado de Threads que podem ser criadas e, caso esse número chegue ao limite, as requisições dos nossos usuários vão começar a ser enfileiradas e podemos ter uma lentidão nas aplicações.

Como já comentamos, em um pico de acesso, podemos sofrer deste problema justamente no momento em que precisamos realizar a integração com

um sistema externo, no qual não temos nenhum controle sobre o tempo de resposta.

A grande sacada para esse tipo de situação é tentar liberar o servidor para atender outras requisições enquanto realizamos a integração com o nosso meio de pagamento. É exatamente para isso que, desde a versão 3 da especificação de Servlets, você pode criar uma Servlet que consegue trabalhar de modo assíncrono! E, para a nossa sorte, o Java EE 7 nos fornece uma abstração sobre isso.

Antes de entrarmos na implementação provida pela especificação, vamos pensar no exemplo de código que gostaríamos de ter.

```
@Transactional
public void checkout() throws IOException {
    systemUserDAO.save(systemUser);

    Checkout checkout = new Checkout(systemUser, cart);
    checkoutDAO.save(checkout);

    new Thread(() -> {
        paymentGateway.pay(checkout.getValue());
    }).start();
}
```

Esse seria o jeito mais simples, mas temos um problema principal. Uma vez que você disparou a `thread`, o processamento do request vai acontecer normalmente, e não é bem isso que queremos. Precisamos que a resposta só seja gerada quando a lógica de aprovar uma nova compra for processada. De dentro da nossa `thread`, necessitamos de algum objeto que nos permita notificar ao servidor o exato momento que ele está liberado para devolver a resposta para o cliente.

Infelizmente, a especificação do JSF não foi concebida para tratarmos requisições dessa maneira. Por conta disso, vamos ter de quebrar nosso processo de checkout em duas etapas.

De dentro do nosso `bean` do JSF, vamos apenas gravar os dados no banco de dados. Quando essa operação acabar, vamos redirecionar a requisição para

outro endereço responsável apenas por aprovar a compra. Daqui a pouco, trataremos dessa outra parte do sistema. Agora vamos resolver a parte do JSF.

```
@Model
public class CheckoutBean {

    private SystemUser systemUser = new SystemUser();
    @Inject
    private SystemUserDAO systemUserDAO;
    @Inject
    private CheckoutDAO checkoutDAO;
    @Inject
    private ShoppingCart cart;
    @Inject
    private FacesContext facesContext;

    @Transactional
    public void checkout() throws IOException {
        systemUserDAO.save(systemUser);

        Checkout checkout = new Checkout(systemUser, cart);
        checkoutDAO.save(checkout);

        String contextName = facesContext.getExternalContext()
            .getContextName();
        HttpServletResponse response = (HttpServletResponse)
            externalContext.getResponse();
        response.setStatus(307);
        response.setHeader("Location", "/" + contextName + "/"
            + services/payment?uuid="+checkout.getUuid());
    }
}
```

Usamos o `HttpServletResponse` para realizar um *redirect* para um endereço que não tem nada a ver com JSF. Os *redirects* que usam o parâmetro do `faces-redirect=true`, por meio do `FacesContext`, só podem ser feitos para URLs tratadas pelo Servlet do próprio JSF.

Um outro ponto a se observar nesse código é o uso do status **307**, cujo significado é *Temporary Redirect*. O endereço que vai fazer a parte final da integração deve ser acessado via `POST`, já que não queremos que o navegador grave o endereço e possibilite um acesso descuidado de um usuário.

Esse status serve justamente para informar para o cliente que o próximo redirecionamento deve ser feito com o mesmo verbo usado para acessar o endereço atual, que no caso foi um `POST`. O header `Location` indica qual endereço o cliente deve acessar para efetuar o redirecionamento.

Outro detalhe interessante é que passamos um argumento chamado `uuid`. A ideia é sempre gerar um identificador um pouco mais complicado para cada pedido, para que nenhum usuário tente alterar a URL e coloque um ID de compra válida. Para não termos que lembrar de definir esse `uuid` o tempo inteiro, vamos utilizar um recurso provido pela JPA.

```
@Entity
public class Checkout {

    ...
    private String uuid;

    ...

    @PrePersist
    public void prePersist(){
        this.uuid = UUID.randomUUID().toString();
    }

    ...
}
```

Você pode anotar um método da sua entidade com a annotation `@PrePersist`, e ele vai ser invocado logo antes da invocação método `persist`, do `EntityManager`.

Esses métodos são conhecidos como **métodos de callback**, e podem ser de grande ajuda quando temos lógicas que sempre devem acontecer sob determinadas circunstâncias.

Existem outras annotations de callback da JPA:

- `PreUpdate` – o método é executado logo antes de um `merge`;
- `PostUpdate` – o método é executado logo após um `merge`;
- `PreRemove` – o método é executado logo antes de um `remove`;
- `PostRemove` – o método é executado logo após um `remove`;
- `PostLoad` – o método executado logo após um `find` ou um `getReference`.

8.7 JAX-RS PARA TODAS AS OUTRAS REQUISIÇÕES HTTP

Até agora estávamos usando o JSF para tratar as requisições web referentes à nossa aplicação. Entretanto, como ele não oferece suporte para tratar as mesmas requisições de forma assíncrona, vamos ter de optar por outro pedaço da especificação, o JAX-RS.

O JAX-RS, acrônimo para *The Java API for RESTful Web Services*, é a especificação que nos possibilita participar de integrações puramente baseadas no protocolo HTTP. Ainda vamos explorar mais dessas possibilidades em um capítulo especial só para isso, mas nesse momento já necessitamos da especificação por conta da sua integração com os contextos assíncronos providos pela especificação de Servlets.

A ideia é que criemos um `bean` do JAX-RS, também conhecido como `Resource`, para carregar a compra pelo `uuid` e fazer a integração com o sistema externo.

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

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriBuilder;
```

```
@Path("payment")
public class PaymentResource {

    @Inject
    private CheckoutDAO checkoutDao;
    @Inject
    private PaymentGateway paymentGateway;

    @POST
    public Response pay(@QueryParam("uuid") String uuid) {
        String contextPath = ctx.getContextPath();
        Checkout checkout = checkoutDao.findByUuid(uuid);

        BigDecimal total = checkout.getValue();

        paymentGateway.pay(total);

        URI redirectURI = UriBuilder
            .fromPath("http://localhost:8080"
                + contextPath + "/site/index.xhtml")
            .queryParams("msg", "Compra realizada com sucesso")
            .build();
        Response response = Response.seeOther(redirectURI)
            .build();

        return response;
    }
}
```

Por conta do CDI, conseguimos receber todas as nossas dependências injetadas. O código em si, até a parte da integração, é o mesmo que tinha no bean do JSF. A parte nova é o tipo de retorno do nosso método. Perceba que, quando acabamos a integração, devolvemos um objeto do tipo `Response`, e aqui entra um pouco mais de teoria sobre o HTTP.

O método `seeOther` da classe `Response` monta um objeto cuja res-

posta gerará o status 303, também conhecido como **See Other**. Esse status deve ser usado quando queremos notificar o cliente da requisição de que ele deve pegar a resposta e acessar a URI retornada por meio de um GET.

Este é um dos pilares do JAX-RS: prover abstrações para trabalharmos de maneira direta com o HTTP, usando o protocolo como alicerce para as integrações que vamos implementar dentro da nossa aplicação.

Outro detalhe importante é o uso da annotation `@Path` para indicar qual o endereço base do nosso recurso. Qualquer método que vá ser acessado dentro dessa classe será solicitado por meio de um endereço no estilo `payment/*`. Por exemplo, configuramos o método que realiza a integração para ser acessado apenas via POST.

```
@POST
public Response pay(@QueryParam("uuid") String uuid) {
    ...
}
```

Poderíamos, inclusive, ter adicionado mais um `@Path` em cima do método. Dessa forma, teríamos um endereço específico para ele. Optamos apenas por diferenciar o acesso através do verbo HTTP. Perceba o endereço que aparece no navegador, depois de clicarmos no botão de finalização de compra.

```
http://localhost:8080/casadocodigo/services/payment
?uuid=1234567890123456789
```

Ainda tem uma parte do endereço, o `/services`, que não desvendamos. Segure um pouco a ansiedade, temos de caminhar com calma.

Tratando a requisição de maneira assíncrona

Para fechar, só precisamos fazer com que o código de integração com o sistema externo rode em um contexto assíncrono. Isso já é suportado pelo próprio JAX-RS, por conta da sua integração com a especificação de Servlets.

```
@Path("payment")
public class PaymentResource {

    private static ExecutorService executor =
```

```
        Executors.newFixedThreadPool(50);
    @Context
    private ServletContext ctx;
    @Inject
    private CheckoutDAO checkoutDao;
    @Inject
    private PaymentGateway paymentGateway;

    @POST
    public void pay(@Suspended final AsyncResponse ar,
        @QueryParam("uuid") String uuid) {

        String contextPath = ctx.getContextPath();
        Checkout checkout = checkoutDao.findByUuid(uuid);

        executor.submit(() -> {

            BigDecimal total = checkout.getValue();

            try {
                paymentGateway.pay(total);

                URI redirectURI = UriBuilder
                    .fromUri(contextPath +
                        "/site/index.xhtml")
                    .queryParams("msg",
                        "Compra realizada com sucesso")
                    .build();

                Response response = Response
                    .seeOther(redirectURI).build();
                ar.resume(response);

            } catch (Exception exception) {
                ar.resume(exception);
            }
        });
    }
```

```
    }  
}
```

Respire fundo. Vamos olhar com carinho para o código que acabamos de escrever. O primeiro detalhe é que usamos um objeto do tipo `ExecutorService`.

```
private static ExecutorService executor =  
    Executors.newFixedThreadPool(50);
```

Existem várias implementações dessa interface para criarmos um *pool de threads*, muito similar ao que já fazemos quando criamos um *pool de conexões*. A ideia é não ter de ficar criando novas threads o tempo inteiro. O atributo é estático, porque você quer usar o mesmo pool para tratar todas as requisições.

O `ExecutorService` possui o método `submit`, no qual precisamos passar uma instância de alguma implementação de `Runnable`. Para não termos de ficar usando classes anônimas, recorreremos para o uso dos *lambdas* do Java 8.

```
executor.submit(() -> {  
  
    BigDecimal total = checkout.getValue();  
  
    try {  
        paymentGateway.pay(total);  
  
        URI redirectURI = UriBuilder  
            .fromUri(contextPath + "/site/index.xhtml")  
            .queryParams("msg",  
                "Compra realizada com sucesso")  
            .build();  
  
        Response response = Response.seeOther(redirectURI)  
            .build();  
        ar.resume(response);  
  
    } catch (Exception exception) {  
        ar.resume(exception);  
    }  
})
```

```
    }
  });
```

Agora vem a parte que faltava para o nosso código funcionar: o objeto do tipo `AsyncResponse`. Essa é a abstração do JAX-RS para notificarmos o servidor de que nossa execução assíncrona foi finalizada.

```
@POST
public void pay(@Suspended final AsyncResponse ar,...){
    executor.submit(() -> {

        BigDecimal total = checkout.getValue();

        try {

            //codigo de integração

            Response response = Response.seeOther(redirectURI)
                .build();
            ar.resume(response);

        } catch (Exception exception) {
            ar.resume(new WebApplicationException(exception));
        }
    });
}
```

O método `resume` é o responsável por fazer a notificação de que nossa tarefa assíncrona foi finalizada. Para que tudo funcione, precisamos anotar o parâmetro do tipo `AsyncResponse` com a annotation `@Suspended`. Ela é usada para indicar para o container que o método em questão deve ser executado em um contexto assíncrono.

Pronto, agora fazemos o código de comunicação com o sistema externo sem bloquear o nosso servidor! Essa é uma ótima técnica, tarefas demoradas não precisam e não **devem** ser executadas na thread que atende a requisição em si.

8.8 CONFIGURANDO O JAX-RS

Para que tudo funcione, só faltou resolver a configuração responsável por associar o padrão de endereço `/services/*` para qualquer `Resource` do JAX-RS. Essa configuração não poderia ser mais simples, basta que criemos uma classe-filha de `Application`.

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

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/services")
public class JaxRsConfiguration extends Application {

}
```

Além de herdarmos de `Application`, anotamos a classe com `ApplicationPath` e definimos qual o endereço base. É justamente por isso que, quando fomos acessar o `Resource` responsável pelo pagamento, será usado o endereço `services/payment`.

Outra maneira de realizar a mesma configuração é utilizar o `web.xml`. Como nossa aplicação é web, basta que adicionemos um `Servlet` específico.

```
<servlet>
  <servlet-name>javax.ws.rs.core.Application</servlet-name>
</servlet>
<servlet-mapping>
  <servlet-name>javax.ws.rs.core.Application</servlet-name>
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

O detalhe importante aqui é o valor da tag `servlet-name`. É obrigatório que seja usado `javax.ws.rs.core.Application`. A especificação obriga que a implementação do JAX-RS registre, dinamicamente, uma `Servlet` exatamente com esse nome. E aí, no `url-pattern`, colocamos o nosso padrão de URL base.

Este autor sempre prefere configurações baseadas em código Java, então é com a primeira opção que vamos seguir no livro.

8.9 CURIOSIDADE: USANDO UM `EXECUTORSERVICE` GERENCIADO

No nosso código de integração, fomos nós que ficamos responsáveis pela criação de um *pool de threads*. Por mais que isso funcione, temos de ficar atentos pois, no momento que decidimos usar um recurso não injetado pelo servidor, passamos a ter que talvez nos preocupar com alguns detalhes a mais de infraestrutura. Por exemplo, tivemos de decidir o número de *threads* disponíveis nos nossos *pools*.

Sabendo que certos códigos precisam ser executados em outras *threads*, o Java EE, na versão 7, trouxe uma nova especificação, a *JAVA EE Concurrency Utilities*. Vamos ver como poderia ficar nosso código de integração com o meio de pagamento.

```
@Path("payment")
public class PaymentResource {

    @Context
    private ServletContext ctx;
    @Inject
    private CheckoutDAO checkoutDao;
    @Inject
    private PaymentGateway paymentGateway;
    @Resource(name =
        "java:comp/DefaultManagedExecutorService")
    private ManagedExecutorService managedExecutorService;

    @POST
    public void pay(@Suspended final AsyncResponse ar,
        @QueryParam("uuid") String uuid) {

        String contextPath = ctx.getContextPath();
        Checkout checkout = checkoutDao.findByUuid(uuid);

        managedExecutorService.submit(() -> {
```

```
BigDecimal total = checkout.getValue();

try {
    paymentGateway.pay(total);

    URI redirectURI = UriBuilder
        .fromUri(contextPath +
            "/site/index.xhtml")
        .queryParams("msg",
            "Compra realizada com sucesso")
        .build();

    Response response =
        Response.seeOther(redirectURI).build();
    ar.resume(response);

} catch (Exception exception) {
    ar.resume(exception);
}

});
}
```

A novidade é que, em vez de usar a interface `ExecutorService`, passamos a usar a `ManagedExecutorService`. Perceba que a injeção foi feita por meio da annotation `@Resource`. Também tivemos de usar a chave `java:comp/DefaultManagedExecutorService`, que é o nome imposto pela especificação que deve estar disponível na JNDI.

A alteração em si foi simples, mas ainda temos outros ganhos não tão explícitos. Por exemplo, nosso código pode necessitar de transações, EJBs, segurança e muitos outros detalhes que estamos analisando durante a construção do projeto. Quando você inicia uma *thread* manualmente, tudo isso é perdido, já que o servidor não tem conhecimento desse recurso. Quando usamos um `ExecutorService` gerenciado pelo servidor de aplicações, temos a garantia de que os devidos contextos serão propagados, o que nos permite focar na nossa regra de negócio do que em detalhes de infraestrutura.

8.10 CONCLUSÃO

Este foi um capítulo bem denso. Usamos novos status do HTTP, adicionamos o JAX-RS na nossa aplicação para lidar com requisições não relacionadas com o JSF, e ainda discutimos como lidar com requisições potencialmente demoradas, delegando a execução para o contexto assíncrono do servidor.

Minha sugestão, assim como foi no último capítulo, é que você pare um pouco para digerir tudo o que fizemos. No próximo passo da nossa aventura, vamos debater sobre cache e como isso pode ajudar na performance da aplicação, com certeza um tópico mais simples e de fácil compreensão.

CAPÍTULO 9

Melhorando a performance com cache

Sempre que entramos no site da Casa do Código, ele nos mostra um monte de livros que podemos comprar. O leitor mais atento já deve ter percebido que essa listagem não muda muito dentro de um determinado intervalo de tempo. E o fato de não mudar nos leva a um questionamento: *será que realmente precisamos ficar executando as duas consultas todas as vezes em que um usuário acessar essa página?*

9.1 CACHEANDO O RETORNO DAS CONSULTAS NA JPA

Apenas para refrescarmos a memória, na home da Casa do Código exibimos duas listagens de livros: a primeira com os principais lançamentos e a segunda

com outros livros da editora.

Atualmente, toda vez que um novo usuário acessa a aplicação, essas consultas devem ser realizadas no banco de dados. Caso os dados mudassem em cada uma das consultas, até faria sentido buscar as informações o tempo todo, mas nessa situação em específico, os livros retornados são sempre os mesmos.

Uma das principais ações que podemos realizar para melhorar a performance da aplicação é justamente minimizar o acesso ao banco de dados, guardando os retornos em um **cache**. Na verdade, isso pode ser aplicado para qualquer parte do seu sistema que busque informações de uma aplicação externa.

Para a nossa alegria, o Hibernate já suporta que o resultado de certas queries do sistema devam ser guardados na memória. Esse *cache* compartilhado entre várias instâncias do `EntityManager` é conhecido como *Second Level Cache*.

```
public class BookDAO {

    @PersistenceContext
    private EntityManager manager;

    ...

    public List<Book> lastReleases() {
        TypedQuery<Book> query = manager
            .createQuery(
                "select b from Book b where b.releaseDate
                <= now() order by b.id desc",
                Book.class).setMaxResults(3);

        query.setHint(QueryHints.HINT_CACHEABLE, true);

        return query.getResultList();
    }

    public List<Book> last(int number) {
        TypedQuery<Book> query = manager
            .createQuery("select b from Book b
```

```
        join fetch b.authors",
        Book.class).setMaxResults(number);

    query.setHint(QueryHints.HINT_CACHEABLE, true);

    return query.getResultList();
}

}
```

O método `setHint` espera como argumento uma `String` informando o nome da configuração que queremos definir, assim como seu valor. A constante `HINT_CACHEABLE` guarda o valor `org.hibernate.cacheable`.

Caso estivéssemos usando diretamente a API do Hibernate, poderia ter sido usado método `setCacheable`. O `setHint` foi a maneira encontrada pela especificação para podermos invocar alguns detalhes da implementação sem recorrer às suas classes.

Agora, se tentarmos acessar o endereço `site/index.xhtml` e observarmos o console do WildFly, vamos perceber que a query ainda está sendo executada todas as vezes. O *cache* de query não está habilitado por padrão. Como é uma configuração muito sensível, o Hibernate exige que você, por vontade própria, habilite o uso do cache de consultas. Precisamos alterar o arquivo `persistence.xml`.

```
<persistence-unit name="casadocodigo-persistence-unit"
    transaction-type="JTA">
    ...

    <properties>
        ...
        <property name="hibernate.cache.use_query_cache"
            value="true"/>
    </properties>
</persistence-unit>
```

A configuração de cache de queries ainda não é suportada pela especificação, então usamos chaves específicas do Hibernate. Após essa configuração, quando tentamos iniciar o servidor, recebemos uma *exception*.

Caused by:

```
org.hibernate.cache.NoCacheRegionFactoryAvailableException:  
Second- level cache is used in the application, but property  
hibernate.cache.region.factory_class is not given; please either  
disable second level cache or set correct region factory using  
the hibernate.cache.region.factory_class setting and make sure  
the second level cache provider (hibernate-infinispan, e.g.) is  
available on the classpath.
```

Informamos que queremos habilitar o *cache* de queries, mas ainda precisamos configurar o Hibernate para usar o *cache* de segundo nível de maneira geral. Podemos cachear muito mais do que queries, e vamos fazer isso na continuação deste capítulo!

```
<persistence-unit name="casadocodigo-persistence-unit"  
  transaction-type="JTA">  
  ...  
  
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>  
  <properties>  
    ...  
    <property name="hibernate.cache.use_query_cache"  
      value="true"/>  
  </properties>  
</persistence-unit>
```

A tag `shared-cache-mode` da JPA deve ser usada para informarmos como vamos decidir quais objetos devem ser cacheados. Vamos pensar um pouco mais sobre isso. Quando fazemos uma consulta por meio da JPA, geralmente é retornada uma lista de objetos. Essa configuração serve para termos a possibilidade de dizer quais tipos de objetos devem ser mantidos em *cache*. A opção `ENABLE_SELECTIVE` é a mais utilizada, pois permite que o programador defina exatamente quais entidades podem ser mantidas na memória.

Veja quais são as outras opções:

- `ALL` – todas as entidades podem ficar no cache. Geralmente não usamos essa opção, pois nem toda entidade merece ser cacheada.

- `NONE` – nenhuma entidade pode ir para o cache. Só seria usada se a aplicação realmente quisesse deixar explícito que nada pode ser cacheado.
- `DISABLE_SELECTIVE` – parte do princípio que todas as entidades podem ficar no cache. Dessa forma, o programador deve ir configurando cada entidade que não pode ser cacheada.

Com nossa configuração feita, chegou a hora de acessarmos a página inicial do site. Porém, quando o fazemos, percebemos um comportamento muito estranho: diversas consultas estão aparecendo no console do servidor.

Hibernate:

```
select
    book0_.id as id1_1_0_,
    book0_.coverPath as coverPat2_1_0_,
    book0_.description as descript3_1_0_,
    book0_.numberOfPages as number0f4_1_0_,
    book0_.price as price5_1_0_,
    book0_.releaseDate as releaseD6_1_0_,
    book0_.summaryPath as summaryP7_1_0_,
    book0_.title as title8_1_0_
from
    Book book0_
where
    book0_.id=?
```

Hibernate:

```
select
    book0_.id as id1_1_0_,
    book0_.coverPath as coverPat2_1_0_,
    book0_.description as descript3_1_0_,
    book0_.numberOfPages as number0f4_1_0_,
    book0_.price as price5_1_0_,
    book0_.releaseDate as releaseD6_1_0_,
    book0_.summaryPath as summaryP7_1_0_,
    book0_.title as title8_1_0_
from
    Book book0_
where
```

```
        book0_.id=?
Hibernate:
    select
        book0_.id as id1_1_0_,
        book0_.coverPath as coverPat2_1_0_,
        book0_.description as descript3_1_0_,
        book0_.numberOfPages as numberOf4_1_0_,
        book0_.price as price5_1_0_,
        book0_.releaseDate as releaseD6_1_0_,
        book0_.summaryPath as summaryP7_1_0_,
        book0_.title as title8_1_0_
    from
        Book book0_
    where
        book0_.id=?
Hibernate:
    select
        book0_.id as id1_1_0_,
        book0_.coverPath as coverPat2_1_0_,
        book0_.description as descript3_1_0_,
        book0_.numberOfPages as numberOf4_1_0_,
        book0_.price as price5_1_0_,
        book0_.releaseDate as releaseD6_1_0_,
        book0_.summaryPath as summaryP7_1_0_,
        book0_.title as title8_1_0_
    from
        Book book0_
    where
        book0_.id=?
```

Entendendo exatamente como funciona o cache de segundo nível

As implementações de *cache*, em geral, são estruturas baseadas em mapas. Quando pedimos para o Hibernate deixar o resultado das consultas no cache, o que ele faz, por baixo dos panos, é justamente adicionar uma entrada nesse mapa, associando o retorno da consulta. É justamente aqui que está nosso problema. Para o cache de query, são armazenados apenas os *ids* das entidades retornadas, não os objetos completos.

Internamente, as entradas do cache são divididas nas chamadas **regiões**. Elas servem para que a aplicação possa, por exemplo, limpar toda uma região de cache.

```
//exemplo para limpar o cache pela API do Hibernate  
entityManagerFactory.  
    unwrap(SessionFactory.class).getCache().  
    evictQueryRegion("algumaRegiao");
```

Também podemos acessar o cache de segundo nível diretamente pela API da JPA, mas ela ainda nos oferece menos opções. Não conseguimos limpar uma região especificamente, por exemplo. Veja um exemplo de como você poderia acessá-lo:

```
Cache cache = entityManagerFactory().getCache();  
cache.evictAll();
```

No fim, o que acontece é que, para cada `id` encontrado no cache da query específica, o Hibernate vai buscar por esse mesmo `id` na região do cache da entidade específica. Como ele não encontra, faz um `select` para buscar os dados de determinada entidade.

Habilitando a entidade para ser cacheada

Para resolvermos o nosso problema, basta que seja adicionada uma annotation na entidade que está participando da query.

```
package br.com.casadocodigo.loja.models;  
  
...  
import javax.persistence.Cacheable;  
  
@Entity  
@Cacheable  
public class Book {  
    ...  
}
```

Perceba que essa annotation também é da especificação. Agora sim, se tentarmos acessar a página inicial da aplicação duas vezes, vamos perceber

que, a partir da segunda, as consultas param de ser feitas. Sempre que você pensar em cache de segundo nível, comece anotando as entidades para não cair neste tipo de problema.

Com a entidade anotada, ainda ganhamos de brinde o cache da entidade quando usamos o método `find` do `EntityManager`. Quando você tentar acessar a página de detalhes de um livro, perceberá que só a consulta dos autores do livro vai ser disparada, por conta do *Lazy Load*, mas não mais a da entidade.

9.2 PROVIDER DE CACHE E SUAS CONFIGURAÇÕES

Quando optamos por usar o cache de segundo nível, somos obrigados a decidir qual será a implementação usada para manter os dados em memória. Em um sistema sendo executado fora de um servidor de aplicação, a opção mais natural tem sido usar o *EhCache*.

Como estamos rodando dentro do WildFly, já estamos utilizando uma implementação chamada *Infinispan*. Não foi necessário fazer nenhuma configuração relativa a isso, pois a implementação do `EntityManagerFactory` usada pelo WildFly já usa o *Infinispan* por default.

Configurando o timeout

Um ponto muito relevante sobre o cache é o tempo que os objetos devem viver nele. Por exemplo, os detalhes de um livro pouco mudam com o passar do tempo. O máximo que acontece é atualização de detalhes internos dos capítulos por conta de uma errata, ou atualização de tecnologia.

Pensando nesse cenário, podemos colocar um tempo de expiração bem alto, especificamente para a entidade `Book`. O *Infinispan* permite que essas configurações sejam adicionadas no próprio `persistence.xml`.

```
<persistence-unit name="casadocodigo-persistence-unit"
    transaction-type="JTA">
    ...
    <properties>
        ...
```



```
<!-- infinispán cache -->
<property
name="hibernate.cache.infinispán.br.com.casadocódigo
.loja.models.Book.expiration.lifespan"
value= "900000"/>

</properties>
</persistence-unit>
```

É importante notar que essa é uma configuração específica do Infinispán. Caso o seu projeto utilize, por exemplo, o EhCache, a configuração já será diferente. No caso específico do EhCache, ela seria feita em um arquivo separado, chamado de `ehcache.xml`. O valor deve ser passado em milissegundos, por isso informamos `900000`, que representa 15 minutos. Na verdade, podemos passar bem mais, pensando que os livros realmente não são atualizados frequentemente.

Outro ponto importante para ser notado é o nome da chave de configuração. Basicamente, ele obedece a um template imposto pelo Infinispán.

```
hibernate.cache.infinispán.NOME_COMPLETO_ENTIDADE
.expiration.lifespan
```

Caso você queira definir um tempo padrão para qualquer entidade anotada, pode usar a opção `hibernate.cache.infinispán.entity.expiration.expiration.lifespan`.

Existem diversas outras configurações que podem ser feitas. É aconselhado que o leitor acesse o link <http://bit.ly/conf-infinispán>.

9.3 INVALIDAÇÃO DO CACHE POR ALTERAÇÃO

Um último detalhe importante que deve ser lembrado é sobre a invalidação dos dados que estão presentes no cache. Por exemplo, todo cache de query é automaticamente invalidado quando realizamos qualquer operação com as entidades pertencentes ao cache.

No nosso caso, quando inserimos um novo livro, os caches das queries relacionadas à classe `Book` são desfeitos. A motivação é simples: a mesma

query, a partir de agora, pode trazer um resultado diferente e, por conta disso, o cache é invalidado.

Uma outra situação, não de invalidação, mas de alteração do estado do cache, acontece quando alteramos um objeto que está sendo buscado via `find`. Por exemplo, se um livro for colocado no cache e, posteriormente, seus dados forem alterados via `EntityManager`, as novas informações vão ser atualizadas no cache. Como estamos rodando dentro de um servidor de aplicação, a alteração vai ser realizada dentro da mesma transação JTA que o objeto tiver sido modificado.

9.4 CACHEANDO TRECHOS DA PÁGINA

Nesse exato momento, o nosso cache está restrito ao nosso acesso ao banco de dados, só que podemos ir ainda além. Na home da loja, o tempo inteiro processamos o trecho de `xhtml` responsável por exibir as lista, e a pergunta que fica é: *por que estamos fazendo isso?*

Como esses trechos não mudam por um bom tempo, não temos a necessidade de ficar processando eles sempre. O OmniFaces, biblioteca que já usamos quando foi necessário personalizar as mensagens de erro do formulário de checkout, também provê uma tag utilitária para guardar trechos de página no cache.

Vamos fazer uma leve alteração no arquivo `site/index.xhtml`.

```
<html xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:pt="http://xmlns.jcp.org/jsf/passthrough"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:o="http://omnifaces.org/ui">

    ....

    <o:cache>
        <ul id="home-highlight-collection">
            <ui:repeat var="book"
                value="#{homeBean.lastReleases()}">
```

```

        <li class="col-left">
            <a href="#{request.contextPath}/site
                /detalhe.xhtml?id=#{book.id}"
                class="block clearfix"> 
            <h2 class="product-title">
                #{book.title}
            </h2>
            <small class="buy-button">
                Lançamento!
            </small></a>
        </li>
    </ui:repeat>
</ul>
</o:cache>

...

<o:cache>
    <ul class="clearfix book-collection">

        <ui:repeat var="book"
            value="#{homeBean.olderBooks()}">
            <li><a href=
                "#{request.contextPath}/site/detalhe.xhtml?
                id=#{book.id}" class="block clearfix">
                <h2 class="product-title">
                    #{book.title}
                </h2>
                <small
                class="buy-button">Compre</small>
            </a></li>

```

```
        </ui:repeat>
    </ul>
</o:cache>

</html>
```

Adicionamos o *namespace* do OmniFaces na tag HTML e usamos a tag `cache`. O uso é bem direto, basta envolvermos o trecho que só queremos que seja executado uma vez com a tag `cache`, e a mágica está feita. O OmniFaces vai executar o trecho na primeira vez que o usuário entrar na página e, a partir da segunda, recuperará o valor do cache e entregará para o JSF.

O leitor mais atento vai perceber que, a partir do segundo acesso, nem os logs do Hibernate são mais gerados. Isso justamente porque a expression language envolvendo o `#{homeBean.metodo}` não é mais executada. Por padrão, o OmniFaces vai manter o cache atrelado à sessão do usuário e, caso o trecho possa ser reaproveitado para qualquer usuário, podemos usar o atributo `scope`.

```
<o:cache scope="application">
    ...
</o:cache>
```

Ainda é possível especificar diversas outras propriedades na tag de cache do OmniFaces.

- `reset` – você pode passar uma expression language cujo retorno é um booleano. O OmniFaces vai descartar o cache toda vez que a expressão retornar `false`.
- `time` – tempo em segundos para o seu cache expirar. Essa é uma propriedade bem importante, todo código cacheado deveria ter uma configuração clara de quando ele vai expirar.

Por fim, caso sua aplicação precise definir um valor padrão de maneira global, o mesmo pode ser feito no arquivo `web.xml`.

```
<context-param>
    <param-name>
```

```
org.omnifaces.CACHE_SETTING_APPLICATION_TTL  
</param-name>  
<param-value>900</param-value>  
</context-param>
```

Aqui dizemos que os trechos de cache associados à aplicação inteira devem permanecer na memória por, no máximo, 15 minutos.

Relembrando que o OmniFaces é uma biblioteca que realmente tem muitos componentes utilitários, considere-o como parte integrante do seu dia a dia de desenvolvimento com o JSF.

9.5 CONCLUSÃO

Este foi um capítulo mais curto, mas muito importante. Habilitar o cache pode fazer gerar um bom impacto de performance na sua aplicação. No entanto, seja pragmático e faça testes de performance para ver se o impacto esperado realmente é o atingido.

Outro ponto importante é que colocar objetos, trechos de página e qualquer outra coisa no cache, muitas vezes, tem a ver com uma decisão de negócio. Por exemplo, na Casa do Código, o dono do produto pode ter informado que os livros são atualizados semanalmente. Sem essa informação, não poderíamos ter deixado as listagens por tanto tempo na memória.

Encerrando as preocupações, lembre-se de que ter muitos objetos na memória incrementa o consumo, o que pode levar, por conta de um uso exagerado, à falta de memória em algum momento. Lembre sempre de definir políticas de expiração de cache para ter um controle mais fino relativo a isso.

Minha dica é que você não pare de ler agora, este capítulo foi mais tranquilo e o próximo também será! Nele, vamos possibilitar que os dados da nossa aplicação sejam exibidos por outros tipos de aplicações, não apenas navegadores.

CAPÍTULO 10

Respondendo mais de um formato

Nossa aplicação, até o presente momento, lida com requisições vindas de um navegador, só que agora vamos um pouco além do que já existe na Casa do Código. Sites de vendas muito grandes, como a Amazon e o Submarino, além de exporem os seus produtos, fazem parcerias com outros sites para que essas outras aplicações também possam exibi-los.

10.1 EXPONDO OS DADOS EM OUTROS FORMATOS

Quando falamos de integração com outras aplicações, o primeiro ponto em que temos que pensar é: qual é o formato que vamos usar para realizar a integração?

Atualmente, nossa aplicação só é capaz de retornar páginas para os clientes, no caso os navegadores, em HTML. Um outro tipo de cliente, hoje já muito comum, são os celulares com Android ou iOS. E como você já deve esperar, exibir dados através de HTML pode não ser o formato ideal para ser usado nesses aparelhos.

Para conseguir retornar formatos de respostas diferentes, teremos de usar o JAX-RS, já que o JSF não suporta outro tipo de resposta que não seja HTML. Como já fizemos toda parte de configuração do JAX-RS, basta que criemos um `Resource` para responder aos formatos que desejamos.

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

//imports

@Path("books")
public class BooksResource {

    @Inject
    private BookDAO bookDAO;

    @GET
    @Produces({ MediaType.APPLICATION_JSON })
    @Path("json")
    public List<Book> lastBooksJson() {
        return bookDAO.lastReleases();
    }
}
```

Perceba que o código foi bem direto. Criamos um método, retornamos a lista de livros e usamos a annotation `@Produces` para dizer que este método suporta devolver a resposta no formato JSON. Por conta do CDI, mais uma vez, podemos receber tudo o que precisamos injetado.

Para fechar, usamos a annotation `@Path` para definir o endereço base do `Resource`, e a annotation `@GET` para dizer que este método só pode ser acessado por meio de requisições que usem este verbo.

Uma pergunta que pode ficar na cabeça é: *por que o verbo de acesso só pode ser get?* O principal motivo é a semântica da operação. Um `get` re-

presenta que a informação está sendo buscada, e uma consequência disso é que o cliente pode fazer diversas requisições para este endereço, que ele sabe que **nunca** vai alterar o estado do servidor, por exemplo. Diferente de quando usamos um `POST` para validar o pagamento, no capítulo 8. O `POST` sempre indica que algo vai ser criado no servidor, então, se o cliente faz duas requisições com os mesmos dados usando este verbo, ele tem de estar consciente que algo pode dar errado.

Para testarmos se nosso código está retornando a resposta que deveria, podemos usar o `curl`.

```
curl -X GET "http://localhost:8080/casadocodigo/services  
/books/json"
```

O leitor mais curioso que está tentando executar a requisição deve ter percebido que o resultado não foi completo. Só foi retornando um livro e, mesmo assim, faltando a relação de autores. Veja um exemplo de resultado:

```
[  
  {  
    "id": 60,  
    "title": "JAVAEE",  
    "description": "Uma aplicação completa usando o que...",  
    "numberOfPages": 200,  
    "price": 50,  
    "author": []  
  }  
]
```

O motivo fica claro quando vamos olhar no console do servidor.

```
Caused by: org.hibernate.LazyInitializationException: failed to  
lazily initialize a collection of role:  
br.com.casadocodigo.loja.models.Book.authors,  
could not initialize proxy - no Session
```

Fomos, novamente, surpreendidos pelo *Lazy Load*. Vamos lembrar como está o método `lastReleases` na classe `BookDAO`:

```

public List<Book> lastReleases() {
    TypedQuery<Book> query = manager
        .createQuery(
            "select b from Book b where b.releaseDate
              <= now() order by b.id desc",
            Book.class).setMaxResults(3);
    query.setHint(QueryHints.HINT_CACHEABLE, true);
    return query.getResultList();
}

```

Os autores só vão ser carregados no momento em que o método `getAuthors` for invocado. Como este só é chamado na hora de transformar o objeto do tipo `List<Book>` para JSON, o objeto não está ligado ao `EntityManager`. É exatamente o mesmo problema que tivemos na tela de detalhes do livro, o objeto entra no estado `detached`. Para a nossa sorte, a solução é a mesma. Transformamos nosso `Resource` em EJB `Stateful` e trabalhamos com o `EntityManager extended`.

```

@Path("books")
@Stateful
public class BooksResource {

    @PersistenceContext(type=PersistenceContextType.EXTENDED)
    private EntityManager entityManager;
    private BookDAO bookDAO;

    @PostConstruct
    private void loadDAO(){
        this.bookDAO = new BookDAO(entityManager);
    }

    @GET
    @Produces({ MediaType.APPLICATION_JSON})
    @Path("json")
    public List<Book> lastBooksJson() {
        return bookDAO.lastReleases();
    }
}

```

De novo, por mais que essa solução nos faça escrever um pouco mais de código, deixamos tudo muito claro. Por sinal, esse problema também poderia ter sido resolvido planejando melhor as queries e usando mais *fetch joins*. Como já discutimos, o lado ruim do `fetch` é que as nossas queries ficarão mais complicadas.

Este autor prefere um uso conjunto, analisando cada uma das situações. Caso você tenha um objeto muito complexo, pode planejar a query e trazer só o necessário. Caso contrário, pode apostar um pouco mais no *Lazy Load*.

Antes de avançarmos, vale uma última observação. Só foi possível o uso do `@Stateful`, pois todo `Resource` do JAX-RS é criado e destruído por request e, com isso, é garantido que nosso `EntityManager` também só verá esse escopo. Caso tivéssemos um objeto com um escopo maior, por exemplo de sessão, o `EntityManager` *extended* viveria por um tempo mais longo, o que, na maioria dos casos, não é o que precisamos.

Agora, se rodarmos o mesmo comando para executar o *request*, recebemos a resposta completa! Nesse momento, nossa aplicação consegue retornar os livros em JSON, além das páginas HTML.

E se nosso cliente precisar de um retorno em XML? Teríamos de criar outro método e informar que o retorno é do tipo XML. ;(

```
@GET
@Produces({ MediaType.APPLICATION_XML })
@Path("xml")
public List<Book> lastBooks() {
    return bookDAO.lastReleases()
}
```

Agora, se tentarmos fazer a requisição para o novo endereço, esperamos obter o XML esperado. O problema é que, em vez disso, recebemos uma *exception*.

```
Could not find MessageBodyWriter for response object of type:
    java.util.ArrayList of media type: application/xml
```

JAX-B e um pouco de burocracia

A especificação responsável por definir as regras de serialização de objetos em XML (e vice-versa) é JAX-B. Ao contrário de várias outras especificações que foram evoluindo com o passar do tempo, ela ainda exige uma configuração maior do que a desejada. Por exemplo, perceba que, para gerar o JSON, não precisamos fazer nada de muito especial.

Seguindo a exceção ao pé da letra, entendemos que não conseguimos serializar um objeto do tipo `ArrayList`, pois o JAX-RS não encontrou ninguém capaz de realizar este trabalho. O problema é que o JAX-B exige que a classe do objeto sendo serializado utilize uma annotation chamada `@XmlRootElement`. Só que, parando para pensar, como vamos adicionar uma annotation na classe `ArrayList`, teríamos de criar uma nova classe, que teria a annotation e aí declararia um atributo do tipo `ArrayList`.

```
@XmlRootElement
public class Books {
    private List<Book> list = new ArrayList<>();
}
```

Para a nossa sorte, o *RestEasy*, implementação do JAX-RS usada pelo *WildFly*, suporta que mantenhamos o retorno do tipo `List`. Por baixo do pano, ele envelopa a lista retornada em um objeto específico dele.

De todo jeito, agora somos obrigados a anotar a classe que representa o tipo da lista, nesse caso, a `Book`.

```
@Entity
@Cacheable
@XmlRootElement
public class Book {
    ...
}
```

Da maneira como está, já conseguimos gerar um retorno do servidor. Caso executemos o `curl` mais uma vez, o resultado deve ser parecido com:

```
<collection>
  <book>
```

```

    <authors>
      <id>1</id>
      <name>Alberto Souza</name>
    </authors>
    <authors>
      <id>2</id>
      <name>Mauricio Aniche</name>
    </authors>
    <description>
      ...
    </description>
    <id>59</id>
    <numberOfPages>100</numberOfPages>
    <price>100.00</price>
    ...
  </book>
  <book>
    ...
  </book>
</collection>

```

Tem alguns detalhes estranhos nesse XML. O primeiro é o nome da tag pai, `collection`. Esse é justamente o nome atributo da classe do RestEasy, que serve de `wrapper` para a nossa coleção. Não que isso seja um problema do tamanho do mundo, mas, para o nosso cenário, o melhor é que a tag se chamasse `books`. Para deixar isso flexível, o RestEasy fornece uma `annotation`.

```

@GET
@Produces({ MediaType.APPLICATION_XML })
@Path("/xml")
@Wrapped(element="books")
public List<Book> lastBooks() {
    return bookDAO.lastReleases();
}

```

A `annotation` `@Wrapped` faz o trabalho que queremos. Agora, se pedirmos pelo XML, ele virá com a tag `books` como a principal. É importante ressaltar que essa é uma `annotation` do RestEasy, e não da especificação. Caso

you não queira sair da especificação, vai ter de partir para a criação da classe wrapper, só que manualmente.

Outro ponto estranho é que cada autor está especificado em uma tag chamada `authors`.

```
<authors>
  <id>1</id>
  <name>Alberto Souza</name>
</authors>
<authors>
  <id>2</id>
  <name>Mauricio Aniche</name>
</authors>
```

O mais comum seria algo parecido com:

```
<authors>
  <author>
    <id>4</id>
    <name>Paulo Silveira</name>
  </author>
  <author>
    <id>5</id>
    <name>Sergio Lopes</name>
  </author>
</authors>
```

Para atingirmos essa estrutura, vamos precisar usar mais algumas annotations do JAX-B.

```
...
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Book {

    ...
    @XmlElement(name="author")
    @XmlElementWrapper(name="authors")
    private List<Author> authors = new ArrayList<>();
```

```
    ...  
}
```

A `@XmlAccessorType` é usada para informarmos a implementação do JAX-B que nossas configurações vão ser baseadas nos atributos. O padrão é que ele busque as configurações em cima dos `getters`. Já a `@XmlElementWrapper` serve justamente para dizermos que todos os objetos da lista em questão devem vir dentro de uma tag chamada `authors`. Por fim, usamos a `@XmlElement` para configurar que cada objeto do tipo `Author` deva vir dentro de uma tag chamada de `author`.

Podemos customizar bastante a geração do XML, tudo vai depender da necessidade da aplicação que estiver consumindo. Claro que o melhor é não precisarmos mexer em nada, mas nem sempre isso é possível.

Também é importante prestar atenção no uso da annotation `@XmlElement`, que influencia na geração do JSON. Antes, o nome da propriedade que referenciava os autores era `authors`, mesmo nome do atributo. Agora é `author`, dada a nossa configuração. Este é um problema que a especificação do Java EE ainda precisa atacar: normalizar a geração de JSON e XML.

10.2 CONTENT NEGOTIATION

Ter um método para cada formato de resposta diferente até funciona, o único problema é que você vai acabar com códigos repetidos. Lembre-se de que o único ponto que vai mudar é a representação do retorno; a lógica para recuperar o dado vai ser a mesma. Vamos tentar resolver este problema.

Primeiro, precisamos deixar apenas um método, respondendo dois formatos:

```
@GET  
@Produces({MediaType.APPLICATION_JSON,  
           MediaType.APPLICATION_XML})  
public List<Book> lastBooksJson() {  
    return bookDAO.lastReleases();  
}
```

O ponto agora é: quando um cliente fizer o request para a URL `/services/books`, qual formato vamos retornar? Essa é a parte interessante, o protocolo HTTP já fornece um jeito de lidar com esse problema! Ele permite que o cliente indique qual o formato de resposta que ele prefere.

Veja um exemplo de requisição usando o `curl`:

```
curl -H "Accept:application/json" -X GET
"http://localhost:8080/casadocodigo/services/books"
```

```
curl -H "Accept:application/xml" -X GET
"http://localhost:8080/casadocodigo/services/books"
```

No primeiro exemplo, fizemos uma requisição indicando que desejamos o retorno no formato `application/json`. Isso é feito por meio do cabeçalho `Accept`. Perceba que no segundo já indicamos que queremos o XML como formato. Essa técnica é conhecida como *Content Negotiation*, e é muito utilizada em integrações baseadas no HTTP, também conhecida como REST.

Lembre-se de que REST é o conceito sobre o qual o JAX-RS é fundamentado. Caso seja necessário que todos os métodos do `Resource` suportem os dois formatos de resposta, podemos adicionar a annotation `@Produces` em cima da classe.

```
@Path("books")
@Stateful
@Produces({MediaType.APPLICATION_JSON,
           MediaType.APPLICATION_XML})
public class BooksResource {
    ...
}
```

10.3 SIMULANDO UM CLIENTE PARA O NOSSO SERVIÇO

Para ficar mais próximo ainda da realidade, vamos criar uma aplicação cliente para consumir nosso serviço. Não precisamos de muito, simplesmente crie um novo *Maven project* no seu Eclipse. Os detalhes da criação ficam por sua conta, leitor.

Vamos por uma classe que simula o consumo da listagem dos livros. Lembre-se de manter o servidor rodando.


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

//imports

public class BookList {

    public static void main(String[] args) {
        Client client = ClientBuilder.newClient();
        Response response = client.target(
            "http://localhost:8080/casadocodigo/services/books")
            .request(MediaType.APPLICATION_JSON).get();

        System.out.println(response.readEntity(String.class));
    }
}
```

Perceba que usamos a API do JAX-RS para fazer as requisições. Inclusive, já tínhamos utilizado essa API quando integramos com o validador de pagamentos externos. Essa API foi uma grande adição que foi feita na especificação, não precisamos mais ficar buscando entre várias opções que existiam. Fazemos a requisição especificando o tipo de retorno esperado, nesse caso o `application/json`. Fique à vontade para trocar por outros *media types*.

Um outro ponto importante é o método `readEntity`. Ele espera que passemos o tipo que deverá ser usado para desserializar o JSON. No nosso caso, só foi necessário imprimir no console, mas você poderia ter criado o seu tipo para representar a lista de livros.

Lembre-se de que, nesse momento, você não está dentro do servidor, e sim ao lado do cliente; portanto, não temos acesso a nenhuma classe que já foi criada.

Veja a seguir um esboço do código que você poderia criar no lado do cliente:

```
ClientBook[] books = response.readEntity(ClientBook[].class);
```

Lembre-se sempre de que a classe `ClientBook` teria de mapear todas as informações recebidas do servidor. Para que esse código funcione, precisamos adicionar as dependências da implementação do JAX-RS, no caso, o `RestEasy`.

```
<dependencies>
  <dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-client</artifactId>
    <version>3.0.11.Final</version>
  </dependency>

  <dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jackson-provider</artifactId>
    <version>2.3.4.Final</version>
  </dependency>
</dependencies>
```

Pronto, com isso conseguimos consumir o serviço no estilo REST que está exposto na nossa aplicação. Fique à vontade para ir além e tentar expor novos serviços.

ESPECIFICANDO O TIPO DE RETORNO NA URL

O RestEasy provê uma facilidade para que possamos especificar o formato de retorno do servidor. Basta que você adicione um `context-param` a mais no `web.xml`.

```
<context-param>
  <param-name>resteasy.media.type.mappings</param-name>
  <param-value>
    json : application/json, xml : application/xml
  </param-value>
</context-param>
```

Agora você pode ir no navegador e, por exemplo, acessar o endereço <http://localhost:8080/casadocodigo/services/books.json>.

10.4 CONCLUSÃO

Neste capítulo, foi abordado um tema que está em evidência: integração de sistemas via REST. Suportar o *Content Negotiation* é fundamental para você dar flexibilidade às aplicações clientes sobre qual formato elas preferem. Outro ponto importante foi o uso da annotation `@@Produces`, pois, por meio dela, você vai informar quais são os tipos de retorno suportados por seu serviço.

No próximo capítulo, trabalharemos para fechar uns pequenos detalhes relativos ao fechamento de uma compra. Por exemplo, é necessário que um e-mail seja enviado para notificar o nosso cliente! Como este capítulo foi até curto, não durma no ponto e continue a sua jornada.

CAPÍTULO 11

Mais de processamento assíncrono com JMS

Nosso processo de compra já está bem completo, até tratamos de outros tópicos nos últimos dois capítulos. Só que, para o fechamento da compra ficar ainda mais próximo com o do site da Casa do Código, faltou mandarmos um e-mail de finalização de compra e simularmos o pedido da geração de nota fiscal.

11.1 ENVIANDO O E-MAIL DE FINALIZAÇÃO

Vamos começar pelo envio do e-mail. A lógica não tem nada de complicado: assim que o valor da nossa compra for autorizado, precisamos enviar um e-mail para o usuário que solicitou o pedido.

```
@Path("payment")
public class PaymentResource {

    private static ExecutorService executor =
        Executors.newFixedThreadPool(50);
    @Context
    private ServletContext ctx;
    @Inject
    private CheckoutDAO checkoutDao;
    @Inject
    private PaymentGateway paymentGateway;
    @Inject
    private MailSender mailSender;

    @POST
    public void pay(@Suspended final AsyncResponse ar,
        @QueryParam("uuid") String uuid) {

        String contextPath = ctx.getContextPath();
        Checkout checkout = checkoutDao.findByUuid(uuid);

        executor.submit(() -> {

            BigDecimal total = checkout.getValue();

            try {
                paymentGateway.pay(total);

                String mailBody = "Nova compra. Seu código
de acompanhamento é "+checkout.getUuid();
                mailSender.send("compras@casadocodigo.com.br",
                    checkout.getBuyer().gete-mail(),
                    "Nova compra",
                    mailBody);

                ....
            }
        })
    }
}
```

```
        ...  
    }  
}
```

Para mantermos a clareza do nosso código, já isolamos o código de envio de e-mail em uma classe chamada `MailSender`.

```
package br.com.casadocodigo.loja.infra;  
  
//infra  
  
@ApplicationScoped  
public class MailSender {  
  
    public void send(String from,String to,String subject,  
        String body){  
        //precisamos enviar o e-mail  
    }  
}
```

Especificação JavaMail

Como já estamos fazendo no decorrer do livro, vamos usar mais uma especificação para resolver o nosso problema. A *JavaMail* define tudo o que precisamos quando o assunto é envio de e-mail, e é justamente ela que será usada pela nossa classe `MailSender`.

Como estamos dentro de um servidor de aplicação, nosso único trabalho é configurar o disparo do e-mail. Porém, antes de entrar na configuração específica do servidor, vamos analisar como ficaria o nosso código de envio em si.

```
package br.com.casadocodigo.loja.infra;  
  
import javax.annotation.Resource;  
import javax.enterprise.context.ApplicationScoped;  
import javax.mail.Message;  
import javax.mail.MessagingException;  
import javax.mail.Session;  
import javax.mail.Transport;
```

```
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

@ApplicationScoped
public class MailSender {

    @Resource(mappedName = "java:jboss/mail/gmail")
    private Session session;

    public void send(String from,String to,String subject,
        String body){
        Message mimeMessage = new MimeMessage(session);
        try {
            mimeMessage.setRecipients(javax.mail.Message
                .RecipientType.TO, InternetAddress.parse(to));
            mimeMessage.setFrom(new InternetAddress(from));
            mimeMessage.setSubject("Sua compra foi registrada");
            mimeMessage.setContent(body,"text/html");

            Transport.send(mimeMessage);
        } catch (MessagingException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Respire um pouco e vamos dissecar o código que acabamos de escrever. Perceba que a montagem do e-mail gira em torno do objeto do tipo `MimeMessage`. A partir dele, invocamos todos os métodos necessários para construir o e-mail:

- `setRecipients` – usamos para definir quem vai receber os e-mails;
- `setFrom` – define quem está enviando o e-mail;
- `setSubject` – define o assunto do e-mail;
- `setContent` – o corpo do e-mail em si. No nosso exemplo, informamos que estamos enviando um e-mail em HTML.

O método `send` da classe `Transport` é usado para efetivamente disparar o e-mail. Para concluir, só falta decifrar o objeto do tipo `Session`, que recebemos injetado associado um nome da JNDI. Esse objeto representa basicamente toda nossa configuração de envio de e-mail.

É dentro dele que estão guardadas informações como:

- Login e senha de acesso ao servidor de e-mail;
- Endereço do servidor de e-mail, por exemplo, `smtp.gmail.com`;
- Estilo de comunicação, se vai ser criptografado ou não.

Para que a injeção deste objeto funcione, é preciso adicionar algumas configurações no arquivo `standalone-full.xml`, que está em `caminhoInstalacaoWildFly/standalone/configuration`. Para que não seja necessário navegar por todo sistema de arquivos, podemos acessar o arquivo por meio do próprio Eclipse.

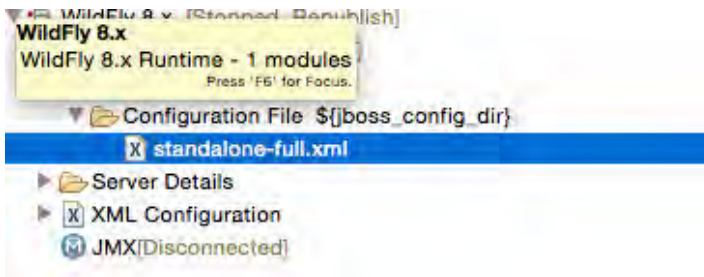


Fig. 11.1: Arquivo de configuração pelo Eclipse

É importante ressaltar que essa parte da configuração é específica do WildFly. Dentro do arquivo, procure pelo `subsystem` associado ao namespace `jboss:domain:mail:2.0`.

```
<subsystem xmlns="urn:jboss:domain:mail:2.0">
  <mail-session name="default"
    jndi-name="java:jboss/mail/Default">
    <smtp-server outbound-socket-binding-ref="mail-smtp"/>
  </mail-session>
</subsystem>
```

Perceba que o próprio WildFly já possui uma configuração padrão para envio de e-mail. Logo, é necessário adicionarmos a nossa.

```
<subsystem xmlns="urn:jboss:domain:mail:2.0">
  <mail-session name="default"
    jndi-name="java:jboss/mail/Default">
    <smtp-server outbound-socket-binding-ref="mail-smtp"/>
  </mail-session>
  <mail-session name="gmail"
    jndi-name="java:jboss/mail/gmail">
    <smtp-server
      outbound-socket-binding-ref="mail-smtp-gmail"
      ssl="true" username="seuUserName"
      password="seuPassword"/>
    </mail-session>
</subsystem>
```

Estamos fazendo a configuração baseada no Gmail, portanto, além de passar nosso usuário e senha, somos obrigados a informar que toda comunicação será feita de maneira criptografada. É exatamente por isso que adicionamos o atributo `ssl`.

Lembre-se de que essas configurações podem mudar em função do serviço de e-mail que estiver usando, mas a tendência é que o nosso exemplo seja suficiente para a maioria dos casos.

Outro ponto importante a se notar é o atributo `jndi-name` da tag `mail-session`. O valor passado foi justamente o mesmo que o referenciado por meio do nosso código.

```
@ApplicationScoped
public class MailSender {

    @Resource(name = "java:jboss/mail/gmail")
    private Session session;

    ...
}
```

A annotation `@Resource` serve, geralmente, para demarcar um ponto de injeção, assim como a `@Inject`. A tendência é que, cada vez mais, pas-

semos a usar a `@Inject`. O problema é que ainda vamos ter os casos em que a injeção vem de recursos mapeados via *JNDI*, e é justamente para essas situações que você usará a `@Resource`.

Para fechar a configuração do e-mail, vamos voltar rapidamente para o arquivo `standalone-full.xml`. Na tag `mail-server`, faltou analisarmos o atributo `outbound-socket-binding-ref`. Ele faz referência ao pedaço da configuração onde especificamos detalhes como endereço e porta. Essa configuração é feita na seção `socket-binding-group`.

```
<socket-binding-group ...>
  <socket-binding .../>
  ...

  <outbound-socket-binding name="mail-smtp">
    <remote-destination host="localhost" port="25"/>
  </outbound-socket-binding>

  <outbound-socket-binding name="mail-smtp-gmail">
    <remote-destination host="smtp.gmail.com" port="465"/>
  </outbound-socket-binding>
</socket-binding-group>
```

Além das que já existem, adicionamos mais uma tag `outbound-socket-binding` para especificar um endereço externo que o WildFly deverá usar. O valor do atributo `name` é exatamente o mesmo do atributo `outbound-socket-binding-ref`.

Agora, caso você feche uma compra, um e-mail já será disparado para o usuário utilizado no processo. Tome só cuidado para não usar um e-mail de mentira, e não ver o e-mail sendo entregue na sua caixa de entrada.

JAVA EE 7 E CONFIGURAÇÕES INDEPENDENTES DO SERVIDOR

Visando, cada vez mais, deixar as configurações independentes no servidor, a partir do Java EE 7 foi criada uma annotation chamada `MailSessionDefinition`. Configurar um envio de e-mail, pelo menos em tese, deveria ser resumido a criar uma classe no sistema e utilizar essa annotation.

```
@MailSessionDefinition(name = "java:jboss/mail/gmail",
    host = "smtp.gmail.com",
    user="user",
    password="password",
    transportProtocol = "smtps",
    from="test@gmail.com",
    properties = {
        "mail.smtp.port=465"
    })
public class MailConfiguration {

}
```

Perceba que fizemos quase todo o processo baseado em atributos já definidos na annotation. Só a porta do servidor que precisou ser especificada por meio de uma propriedade extra.

O problema é que essa configuração, pelo menos no WildFly 8.2, ainda não funciona corretamente e, por isso, optamos por realizar a configuração baseada em arquivos específicos do servidor.

11.2 UM POUCO MAIS SOBRE PROCESSAMENTO ASSÍNCRONO

No capítulo 8, no qual começamos a concluir o fechamento da compra, usamos o objeto do tipo `AsyncResponse` para permitir que o servidor não ficasse com a thread que atende o *request* travada enquanto estávamos nos comunicando com a outra aplicação.

Aqui devemos prestar atenção em um detalhe. Do ponto de vista do usuário da Casa do Código, o processamento continuou síncrono, já que ele teve de ficar esperando pela resposta enquanto nosso processo estava trabalhando. Agora, do ponto de vista do servidor, o processamento foi assíncrono, já que a thread que atende o *request* foi liberada e, só quando a integração foi finalizada, ela foi notificada para gerar a devida resposta para o cliente.

Além de já esperar pela integração com o sistema de validação de pagamento, o nosso usuário agora ainda está esperando pelo envio do e-mail e, também vai ser obrigado a esperar pela comunicação do sistema que gera a nota fiscal. Vamos ver como ficaria o nosso código agora:

```
@Path("payment")
public class PaymentResource {

    ...
    @Inject
    private MailSender mailSender;
    @Inject
    private InvoiceGenerator invoiceGenerator;

    @POST
    public void pay(@Suspended final AsyncResponse ar,
        @QueryParam("uuid") String uuid) {

        String contextPath = ctx.getContextPath();
        Checkout checkout = checkoutDao.findByUuid(uuid);

        executor.submit(() -> {

            BigDecimal total = checkout.getValue();

            try {
                paymentGateway.pay(total);

                String mailBody = "Nova compra.
                Seu código de acompanhamento é "+
                checkout.getUuid(); mailSender
                    .send("compras@casadocodigo.com.br",
```

```

        checkout.getBuyer().gete-mail(),
        "Nova compra", mailBody);

        invoiceGenerator.invoiceFor(checkout);

        ....
    }

    ...
}

```

Para não perdermos a linha de pensamento, vamos analisar a classe `InvoiceGenerator` só no fim do capítulo. O problema desse fluxo é que apenas a validação do valor é obrigatória para liberar a compra; as outras duas operações podem ser feitas sem que o nosso usuário fique esperando pelo retorno.

Para resolver este problema, em vez de realizar a lógica de maneira síncrona, podemos, mais uma vez, partir para uma abordagem assíncrona. Em vez de travar o código esperando pelo envio do e-mail e geração da nota fiscal, simplesmente emitimos uma **mensagem** no sistema informando que uma nova compra acabou de ser aprovada, o tratamento fica para depois.

Vamos dar uma olhada em um esboço de código:

```

@POST
public void pay(@Suspended final AsyncResponse ar,
    @QueryParam("uuid") String uuid) {

    String contextPath = ctx.getContextPath();
    Checkout checkout = checkoutDao.findByUuid(uuid);

    executor.submit(() -> {

        BigDecimal total = checkout.getValue();

        try {
            paymentGateway.pay(total);

```

```
        //emitimos um comunicado de nova compra
        ....
    }

    ...
}
}
```

A ideia é que possamos enviar uma nova mensagem, e que as classes responsáveis tratem isso de maneira assíncrona.

11.3 UTILIZANDO O JMS PARA MENSAGERIA

A especificação JMS (*Java Message Service*) nos possibilita justamente implementar essa forma de trabalhar. Basicamente, precisamos enviar uma mensagem que chegue para os objetos responsáveis por tratar os eventos de uma nova compra. No nosso caso, é necessário enviar o e-mail e solicitar a geração da nota fiscal.

Vamos começar pelo código de envio de mensagem:

```
@Path("payment")
public class PaymentResource {

    ...
    @Inject
    private JMSContext jmsContext;

    @Resource(lookup = "java:/jms/topics/checkoutsTopic")
    private Destination checkoutsTopic;

    @POST
    public void pay(@Suspended final AsyncResponse ar,
        @QueryParam("uuid") String uuid) {
        String contextPath = ctx.getContextPath();
        Checkout checkout = checkoutDao.findByUuid(uuid);
        JMSProducer producer = jmsContext.createProducer();
```

```
executor.submit(() -> {  
  
    BigDecimal total = checkout.getValue();  
  
    try {  
        paymentGateway.pay(total);  
  
        producer.send(checkoutsTopic,  
                        checkout.getUuid());  
  
        ...  
    } catch (Exception exception) {  
        ...  
    }  
});  
}
```

Temos alguns pontos para analisar nesse código. O objeto do tipo `JMSProducer` é o responsável por enviar uma mensagem. Para efetivamente disparar a mensagem para o sistema, usamos o método `send(destino, conteudo)`.

Existem dois tipos de destino no JMS:

- Quando a mensagem deve ser tratada apenas uma vez, usamos um destino conhecido como *Fila*;
- Quando a mensagem pode ser tratada por vários objetos, usamos um destino conhecido como *Tópico*.

Por exemplo, no nosso caso, é necessário que a mensagem do checkout seja tratada pelo objeto que envia e-mail e pelo gerador da nota fiscal, então, nesse caso usamos um tópico.

Agora, caso tivéssemos optado por validar o pagamento enviando uma mensagem, seria necessário que tal mensagem fosse tratada apenas uma vez e, para esse cenário, usar uma fila seria o ideal.

O destino no JMS é representado por meio de um objeto do tipo `Destination`. Assim como já fizemos quando foi necessário receber o contexto de uso do e-mail, temos de utilizar a annotation `@Resource`, pois vamos referenciar o tópico exposto na JNDI.

Além do destino, somos obrigados a passar o conteúdo da mensagem. No nosso caso, optamos por passar o identificador da compra. Dessa forma, cada objeto responsável pelo tratamento de finalização da compra pode carregá-la e realizar sua lógica.

```
@Path("payment")
public class PaymentResource {

    @Resource(name = "java:/jms/topics/checkoutsTopic")
    private Destination checkoutsTopic;

    ....
}
```

Agora, só falta resolver o trecho de código responsável por criar um `JMSProducer`.

```
...

@Inject
private JMSContext jmsContext;
@Resource(lookup = "java:/jms/topics/checkoutsTopic")
private Destination checkoutsTopic;

@POST
public void pay(@Suspended final AsyncResponse ar,
    @QueryParam("uuid") String uuid) {
    ...
    JMSProducer producer = jmsContext.createProducer();

    executor.submit(() -> {
        ...
    })
}
```

O objeto do tipo `JMSContext` representa o elo de conexão entre a aplicação e a infraestrutura necessária para enviar as mensagens. É através dele que criamos um objeto do tipo `JMSProducer`. Para recebê-lo injetado, podemos usar diretamente a annotation `@Inject`. Todo servidor de aplicação é obrigado a disponibilizar uma implementação padrão da interface e deixá-la disponível.

Configurando o tópico

Para que a injeção do `Destination` funcione, é necessário que seja informado ao servidor que o tópico seja criado e exposto com determinado nome na JNDI. Ao contrário da configuração do `DataSource` e do e-mail, a configuração dos destinos do JMS, a partir do Java EE 7, pode ser feita integralmente via annotations.

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

import javax.jms.JMSDestinationDefinition;

@JMSDestinationDefinition(
    name="java:/jms/topics/checkoutsTopic",
    interfaceName = "javax.jms.Topic"
)
public class ConfigureJMSDestinations {

}
```

Essa é a simplicidade que estamos buscando! Configurar um pedaço complexo do sistema de uma maneira extremamente simples. O atributo `name` recebe o nome que será exposto na JNDI. Já o atributo `interfaceName` recebe o nome completo da interface do JMS ligada a um tipo de destino. No nosso caso, passamos `javax.jms.Topic`, mas se fosse necessário trabalhar com uma fila, bastaria passar `javax.jms.Queue`. Ainda é possível, quando você tem vários destinos, configurar múltiplos destinos.

```
@JMSDestinationDefinitions({
    @JMSDestinationDefinition(
```

```
        ...
    ),
    @JMSDestinationDefinition(
        ...
    )
})
public class ConfigureJMSDestinations {

}
```

Talvez cause uma certa estranheza criar uma classe completamente vazia. Aqui esbarramos em uma limitação da linguagem, já que a annotation sempre deve estar associada a um código Java.

11.4 REGISTRANDO TRATADORES DE MENSAGENS COM MDBs

Já adicionamos o suporte ao tratamento assíncrono de partes do processamento de uma nova compra. A única pergunta que ficou é: cada vez que chega uma nova compra mandamos uma mensagem, mas quem está tratando? Precisamos associar objetos com o tópico especificado.

Para fazermos isso, vamos recorrer de novo aos EJBs, só que agora a um tipo especial chamado de *Message Driven Bean* (MDB). É um EJB especial que pode ser associado a um destino e receber as mensagens que forem chegando. Vamos ver como ficaria para criar um MDB responsável por mandar o e-mail da compra.

```
package br.com.casadocodigo.loja.listeners.checkout;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.inject.Inject;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

//outros imports

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(
        propertyName = "destinationLookup",
        propertyValue = "java:/jms/topics/checkoutsTopic")
    })
public class SendCheckoute-mailListener
    implements MessageListener{

    private Logger logger = LoggerFactory
        .getLogger(SendCheckoute-mailListener.class);
    @Inject
    private MailSender mailSender;
    @Inject
    private CheckoutDAO checkoutDao;

    @Override
    public void onMessage(Message message) {
        TextMessage text = (TextMessage) message;
        try {
            Checkout checkout = checkoutDao
                .findByUuid(text.getText());
            String e-mailBody = "<html><body>Compra realizada
                com sucesso. O código de acompanhamento é "
                + checkout.getUuid() + "</body></html>";

            mailSender.send("compras@casadocodigo.com.br",
                checkout.getBuyer().gete-mail(),
                "Sua compra foi registrada com sucesso",
                e-mailBody);

        } catch (Exception e) {
            logger.error("Problema no envio do e-mail",e);
        }
    }
}
```

```
}
```

Precisamos investir um tempo para entender partes deste código. Vamos começar pelo uso da annotation `@MessageDriven`. Ela é a responsável por informar ao servidor de aplicação que estamos criando um novo MDB. Sua configuração, porém, é meio complicada de olhar.

```
@MessageDriven(activationConfig = {  
    @ActivationConfigProperty(  
        propertyName = "destinationLookup",  
        propertyValue = "java:/jms/topics/checkoutsTopic")  
    })
```

Para dizer que queremos escutar as mensagens que forem chegando no tópico que configuramos, precisamos usar a annotation `@ActivationConfigProperty`. Na verdade, o atributo `activationConfig` recebe um array de `@ActivationConfigProperty`. Por meio dessa annotation, vamos passando as informações necessárias.

A lista de possíveis propriedades pode ser encontrada em <http://docs.oracle.com/cd/E19798-01/821-1841/bnbpo/index.html>.

Para o nosso caso, precisamos da `destinationLookup` para associar o nome da JNDI, onde será buscada a referência para o nosso tópico. É importante ressaltar que essa propriedade é prevista na especificação; não importando o servidor que você rode o código, tal configuração deve ser suportada.

Além da annotation relativa ao MDB, também somos obrigados a implementar a interface `MessageListener`, que define o método `onMessage`. É justamente ele que vai ser invocado toda vez que uma nova mensagem chegar.

```
@Override  
public void onMessage(Message message) {  
    TextMessage text = (TextMessage) message;  
    try {
```

```

Checkout checkout = checkoutDao
    .findByUuid(text.getText());
String e-mailBody = "<html><body>Compra realizada
    com sucesso. O código de acompanhamento é "
    + checkout.getUuid() + "</body></html>";

mailSender.send("compras@casadocodigo.com.br",
    checkout.getBuyer().gete-mail(),
    "Sua compra foi registrada com sucesso",
    e-mailBody);

} catch (Exception e) {
    logger.error("Problema no envio do e-mail",e);
}
}

```

Recebemos como argumento um objeto do tipo `Message`, que é a interface mãe de todas as possíveis mensagens do JMS. Como, na hora de mandar a mensagem, foi usado o código `producer.send(checkoutsTopic, checkout.getUuid())`, passando como segundo argumento uma `String`, é montado um objeto do tipo `TextMessage`. Como isso é garantido pela especificação, podemos simplesmente fazer o `cast` da mensagem que chega e recuperar a mensagem enviada.

```

TextMessage text = (TextMessage) message;
try {
    Checkout checkout = checkoutDao.findByUuid(text.getText());
    ...
}

```

Para completar o processo, podemos criar o MDB responsável por pedir a geração da nota fiscal da compra.

```

package br.com.casadocodigo.loja.listeners.checkout;

//imports

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(

```

```
        propertyName = "destinationLookup",
        propertyValue =
            "java:/jms/topics/checkoutsTopic")
    })

public class GenerateInvoiceListener
    implements MessageListener{

    private Logger logger = LoggerFactory.
        getLogger(GenerateInvoiceListener.class);
    @Inject
    private InvoiceGenerator invoiceGenerator;
    @Inject
    private CheckoutDAO checkoutDao;

    @Override
    public void onMessage(Message message) {
        TextMessage text = (TextMessage) message;
        try {
            Checkout checkout = checkoutDao
                .findById(text.getText());
            invoiceGenerator.invoiceFor(checkout);
        } catch (JMSEException e) {
            logger.error("Problema na geracao
                da nota fiscal {}",e);
        }
    }
}
```

O código em si não tem nada de novo. O leitor mais crítico deve até estar pensando: *por que o e-mail e a nota fiscal não foram processados pelo mesmo MessageListener?* O problema, nesse caso, de deixar toda a lógica no mesmo lugar, é que o mal funcionamento de uma pode influenciar na outra. Do jeito que fizemos, mesmo que ocorra um problema no envio do e-mail, a nota fiscal continua sendo gerada.

Uma outra questão é que deixar muito código de processamento longo dentro do mesmo `listener` pode fazer com que o próprio `listener` vire um gargalo, e algumas mensagens comecem a dar *timeout*. Sempre tente deixá-los com o mínimo de responsabilidade possível, e altamente coesos.

Para que a classe `InvoiceGenerator` não fique obscura, vamos dar uma olhada nela:

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

//imports
public class InvoiceGenerator {

    public void invoiceFor(Checkout checkout) {
        Client client = ClientBuilder.newClient();
        InvoiceData invoiceData = new InvoiceData(checkout);
        String uriToGenerateInvoice =
            "http://book-payment.herokuapp.com/invoice";
        Entity<InvoiceData> json = Entity.json(invoiceData);
        client.target(uriToGenerateInvoice).
            request().post(json, String.class);
    }
}
```

Ela usa a API de cliente do JAX-RS para fazer uma requisição a um sistema externo, para solicitar a geração de uma nova nota fiscal. Além disso, usamos o objeto do tipo `InvoiceData` para passar as informações necessárias para a geração da nota fiscal.

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

import java.math.BigDecimal;

import br.com.casadocodigo.loja.models.Checkout;

public class InvoiceData {

    private BigDecimal value;
    private String buyere-mail;

    public InvoiceData(Checkout checkout) {
        this.value = checkout.getValue();
        this.buyere-mail = checkout.getBuyer().gete-mail();
    }
}
```



```
public BigDecimal getValue() {  
    return value;  
}  
  
public String getBuyere-mail() {  
    return buyere-mail;  
}  
}
```

11.5 IMPLEMENTAÇÃO DO JMS UTILIZADA PELO WILD-FLY

Todo serviço de mensageria é provido por um outro servidor, que geralmente é integrado no servidor de aplicação de modo que a aplicação possa tirar proveito. No caso do WildFly, é utilizado o HornetQ (<http://hornetq.jboss.org/>)

A ideia, como vimos no decorrer do capítulo, é tentar minimizar o acoplamento com a implementação da especificação. De toda forma, é necessário entender quais implementações são usadas, já que você pode ser obrigado a recorrer a alguma configuração específica.

11.6 CAUTELA NO USO DO CÓDIGO ASSÍNCRONO

A execução de código de maneira assíncrona tende a elevar a capacidade de um sistema de escalar. Como passamos menos tempo para dar uma resposta, já que boa parte dela vai ser processada em outro momento, conseguimos atender mais requisições em um determinado intervalo de tempo. Entretanto, adicionamos uma certa complexidade no sistema, ponto que sempre deve ser pesado em qualquer decisão arquitetural.

Uma dica deste autor é que você, na maioria das vezes, comece sempre pensando em um código mais simples, geralmente síncrono e, quando necessário, evolua para o código assíncrono.

11.7 CONCLUSÃO

Este foi um capítulo um pouco mais denso! Enquanto a especificação relativa ao envio de e-mails é relativamente simples, a JMS é bem mais complicada. Lidar com código assíncrono sempre é um pouco mais complexo e exige mais da nossa atenção.

É importante lembrar de que sempre que você estiver dentro de um servidor de aplicação, um **Message Driven Bean** deve ser criado para conseguir ser notificado de novas mensagens. Outro ponto importante é a interface `MessageListener`, que deve ser implementada para que o método `onMessage` possa ser chamado dentro do **MDB**.

Minha dica é que você descanse um pouco a sua mente para poder processar tudo que estudamos. No próximo capítulo, vamos adicionar a parte de segurança e, mais uma vez, usaremos uma especificação do Java EE.

CAPÍTULO 12

Protegendo a aplicação

Até agora, todas as URLs do nosso sistema estão acessíveis por todos os usuários. Algumas até são liberadas, como a que leva para a página inicial, que deve exibir todos os livros. Só que temos algumas URLs que necessitam de um usuário logado, como as que compõem a parte de administração da loja.

Podemos até implementar toda essa parte de segurança na mão, mas, como já vem sendo feito no decorrer do livro, vamos usar uma especificação do Java EE que define soluções para várias das necessidades que teremos. É bom sempre lembrar que implementar uma estratégia de segurança não é trivial.

Além de forçar o login para algumas URLs, é necessário ter preocupação com pelo menos mais alguns itens, como:

- Quais perfis podem acessar as URLs, também conhecido como autorização;

- URLs acessadas por vários perfis, mas com trechos de página restritos;
- Fontes de dados diferentes para realização de login.

12.1 DEFININDO AS REGRAS DE ACESSO COM O JAAS

Java Authentication and Authorization Service (também conhecido como JAAS) é justamente a especificação do Java criada para tratar da parte de segurança em nossas aplicações. Inclusive, a especificação de Servlets também já é integrada a essa especificação.

Basicamente, precisamos configurar alguns detalhes para que os endereços da nossa aplicação comecem a ficar protegidos. Boa parte dessa configuração é feita diretamente no arquivo `web.xml`.

```
<web-app xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance" ...
...
    <security-constraint>
        <display-name>Administracao</display-name>
        <web-resource-collection>
            <web-resource-name>
                administracao
            </web-resource-name>
            <description>
                Urls que levam para paginas de administracao
            </description>
            <url-pattern>/livros/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>ADMIN</role-name>
        </auth-constraint>
    </security-constraint>
    <security-role>
        <role-name>ADMIN</role-name>
    </security-role>
    ...
</web-app>
```

Sempre vamos trabalhar com o seguinte par: URLs e `roles`. Declaremos endereços que só podem ser acessados por certos usuários de determinados perfis, definidos no sistema. Para cada associação que for necessária, vamos declarar uma tag `security-constraint` e definir as regras.

Vamos dar um zoom no conteúdo da tag, para deixar tudo completamente claro:

```
<security-constraint>
  <display-name>Administracao</display-name>
  <web-resource-collection>
    <web-resource-name>administracao</web-resource-name>
    <description>
      Urls que levam para paginas de administracao
    </description>
    <url-pattern>/livros/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>ADMIN</role-name>
  </auth-constraint>
</security-constraint>
```

A tag `web-resource-collection` define os padrões de endereço que devem ser protegidos. No nosso caso, informamos que, para acessar qualquer endereço que comece com `/livros`, o usuário deve estar logado. Além disso, informamos os verbos HTTP associados ao acesso. Para completar, usamos a tag `auth-constraint` para indicar quais grupos, também conhecidos como `roles`, podem acessar os endereços protegidos. Dentro da tag `web-resource-collection`, você pode adicionar quantas tags `url-pattern` você desejar, tudo vai depender da aplicação.

Caso você inicie o WildFly e tente acessar o endereço <http://localhost:8080/casadocodigo/livros/form.xml>, perceberá que é retornada uma página em branco, escrito `Forbidden`. Inclusive, se habilitar o *Firebug* ou o *Chrome tools*, vai perceber no console que o status retornado pelo servidor é 403, que indica justamente que você não tem autorização para acessar determinados recursos.

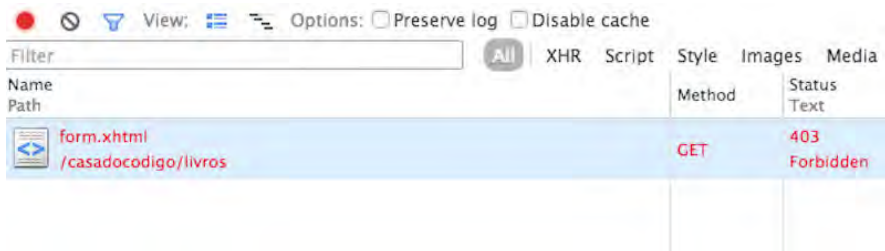


Fig. 12.1: Acesso negado

12.2 CONFIGURANDO O FORMULÁRIO DE LOGIN

Já começamos falando sobre os perfis autorizados para acessar determinados recursos, mas como isso ia ser possível se nem demos a chance de o usuário se logar? Para explicar para o JAAS que necessitamos que o usuário primeiro se autentique, para depois tentar acesso a qualquer recurso, vamos declarar a tag `login-config` no `web.xml`.

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...

  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>
        /users/login.xhtml
      </form-login-page>
      <form-error-page>
        /users/login.xhtml
      </form-error-page>
    </form-login-config>
  </login-config>

  <security-constraint>
    ...
  </security-constraint>
</web-app>
```

A tag `auth-method` indica que queremos trabalhar com um formulário de login, onde o usuário vai passar o seu login e sua senha. Já a tag `form-login-config` serve para indicarmos qual deve ser a página de login e qual deve ser a página quando acontecer alguma falha nele.

Para o nosso caso, apontamos as duas para a mesma página. Para que essa configuração funcione, precisamos criar a página de login em `webapp/users/login.xhtml`.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
  <form method="POST"
    action="${request.getContextPath()}/j_security_check">
    <div>
      <label for="e-mail">e-mail</label>
      <input type="text"
        name="j_username" value="${param.j_username}"
        id="e-mail"/>
    </div>
    <div>
      <label for="password">Senha</label>
      <input type="password" name="j_password"
        id="password"/>
    </div>

    <input type="submit" value="Login" />
  </form>
</body>
</html>
```

O leitor mais atento deve ter estranhado o endereço utilizado na tag `action`, assim como os nomes dos `inputs`. O ponto é que a especificação de Servlets já fornece a integração de autenticação baseado no JAAS, o servidor só precisa saber qual é o endereço que, quando acessado, deve dar início ao processo de autenticação.

Além disso, ele também precisa saber quais parâmetros indicam o login e a senha passados pelo usuário. Todos esses valores são definidos pela especificação de Servlets, e não podem ser alterados. No fundo, também não faz

sentido querer mudar, já que isso não influencia em nada o funcionamento da nossa aplicação.

O problema agora é que sempre que tentamos realizar um login na aplicação, somos barrados. Pensando um pouco melhor, faz até sentido, pois não adicionamos nenhuma senha para os nossos compradores da loja e também não cadastramos nenhum novo usuário com login, senha e perfis associados.

Incrementando o domínio para refletir o modelo de segurança

O domínio da nossa aplicação ainda não foi atualizado para refletir o nosso novo esquema de segurança. O primeiro passo é justamente associar o usuário com os seus respectivos perfis.

```
@Entity
public class SystemUser {
    ...

    @ManyToOne(fetch = FetchType.EAGER)
    private List<SystemRole> roles = new ArrayList<>();

    //getters e setters
}
```

Agora, precisamos da classe `SystemRole`:

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

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class SystemRole {

    @Id
    private String name;

    @Deprecated //apenas para os frameworks
    public SystemRole() {
    }
}
```



```
public SystemRole(String name) {
    this.name = name;
}

//getters e setters
}
```

Quando você subir a aplicação, por conta da configuração do Hibernate, as nossas tabelas já serão atualizadas. Agora podemos cadastrar um novo usuário com o perfil de administração. Podemos inserir alguns dados diretamente pelo console do MySQL.

```
insert into SystemUser(email,password)
values ('admin@casadocodigo.com.br',

);

insert into SystemRole values('ADMIN');

insert into SystemUser_SystemRole values
(id_gerado_para_o_admin,'ADMIN');
```

A senha usada foi gerada a partir do seguinte trecho de código:

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

import org.jboss.security.Base64Encoder;

public class PassGenerator {

    public static void main(String[] args) throws Exception {
        Base64Encoder.main(new String[]{"123456", "SHA-256"});
    }
}
```

A classe `Base64Encoder` é fornecida pelo próprio WildFly, e é útil para representar a senha usando a `Base64`. Além disso, após o texto ter sido *encodado*, é aplicado o algoritmo de *hash* `SHA-256`.

Mesmo com os dados cadastrados, ainda não conseguimos efetuar o login dentro da nossa aplicação. Afinal de contas, onde foi que explicamos para o

servidor da aplicação a maneira como queremos buscar os dados de login e senha passados no formulário?

12.3 CONFIGURANDO O LOGINMODULE DO JAAS

Para que o nosso login seja efetuado, é preciso ter uma classe que consiga recuperar os dados passados pelo usuário e, dessa forma, verifique se realmente esse usuário existe no banco de dados. O JAAS define uma interface chamada `LoginModule`, que deve ser implementada pela aplicação, justamente com o propósito necessitado.

Para a nossa sorte, estamos dentro de um servidor de aplicação, e todos eles já fornecem implementações prontas dessa interface, de modo a facilitar o nosso trabalho.

Configurando um LoginModule no WildFly

Para adicionarmos o módulo de login no WildFly, precisamos, mais uma vez, alterar o arquivo `standalone-full.xml`. Lembre-se de acessá-lo por meio do próprio Eclipse, para não precisar navegar por todo sistema de arquivos.

No arquivo de configuração, temos um módulo associado ao namespace `urn:jboss:domain:security:1.2`. Perceba que, dentro dele, já existem alguns domínios de segurança compostos por um ou mais `login modules`.

```
<security-domains>
  <security-domain name="other" cache-type="default">
    <authentication>
      <login-module code="Remoting" flag="optional">
        <module-option name="password-stacking"
          value="useFirstPass"/>
      </login-module>
      <login-module code="RealmDirect" flag="required">
        <module-option name="password-stacking"
          value="useFirstPass"/>
      </login-module>
    </authentication>
  </security-domain>
```

```

<security-domain name="jboss-web-policy"
  cache-type="default">
  <authorization>
    <policy-module code="Delegating" flag="required"/>
  </authorization>
</security-domain>

...
</security-domains>

```

Os que já estão declarados são usados internamente pelo WildFly. Por exemplo, o `security-domain` de nome `other` é utilizado internamente para verificar as credenciais de acesso das aplicações para EJBs remotos. Eles até podem combinar mais de um `login module`. Ainda no `security-domain` usado pelos EJBs, existem dois módulos de login. Para a maioria das aplicações, apenas um módulo já é suficiente, e esse é justamente o nosso caso.

Vamos adicionar mais um `security-domain` na lista já existente no WildFly.

```

<security-domains>
  ...

  <security-domain name="database-login"
    cache-type="default">
    <authentication>
      <login-module code="Database" flag="required">
        <module-option name="dsJndiName"
          value="java:jboss/datasources/
            casadocodigoDS"/>

        <module-option name="principalsQuery"
          value="select password from SystemUser
            where e-mail=?"/>

        <module-option name="rolesQuery"
          value="select roles_name,'Roles'
            from SystemUser_SystemRole as

```

```
        user_roles inner join SystemUser as
        su on su.id = user_roles.SystemUser_id
        where su.e-mail = ?"/>

        <module-option name="hashAlgorithm"
            value="SHA-256"/>

        <module-option name="hashEncoding"
            value="base64"/>

    </login-module>
</authentication>
</security-domain>
</security-domains>
```

Vamos respirar um pouco e entender toda essa configuração. O primeiro detalhe é que usamos o valor `Database` no atributo `code`. O WildFly já oferece várias implementações da interface `LoginModule` e, quando queremos usar uma delas, fazemos a referência através de uma chave já definida. Seguem alguns exemplos:

- `Database` –
`org.jboss.security.auth.spi.DatabaseServerLoginModule`
- `Ldap` –
`org.jboss.security.auth.spi.LdapLoginModule`
- `Simple` –
`org.jboss.security.auth.spi.SimpleServerLoginModule`
- `PropertiesUsers` –
`org.jboss.security.auth.spi.PropertiesUsersLoginModule`

POSSIBILIDADES DO MÓDULO DE SEGURANÇA

A lista completa pode ser encontrada em <https://docs.jboss.org/author/display/WFLY8/Security+subsystem+configuration>.

Outro detalhe importante é o atributo `flag`. Atribuímos o valor `required` para dizer que nosso usuário, obrigatoriamente, tem de passar por esse módulo para ter acesso à parte protegida da aplicação.

Para fechar essa parte, é necessário discutir as opções que foram passadas para o módulo de login. A maneira mais simples de justificá-las é dando uma olhada no código-fonte da classe que será usada pelo nosso módulo.

```
public class DatabaseServerLoginModule extends
    UsernamePasswordLoginModule

    public void initialize(Subject subject, CallbackHandler
        callbackHandler, Map<String,?> sharedState,
        Map<String,?> options)
    {
        addValidOptions(ALL_VALID_OPTIONS);
        super.initialize(subject, callbackHandler,
            sharedState, options);
        dsJndiName = (String) options.get(DS_JNDI_NAME);
        if( dsJndiName == null )
            dsJndiName = "java:/DefaultDS";
        Object tmp = options.get(PRINCIPALS_QUERY);
        if( tmp != null )
            principalsQuery = tmp.toString();
        tmp = options.get(ROLES_QUERY);

        ...
    }
}
```

Desse código todo, você precisa perceber apenas um detalhe: o método `initialize` recebe, em dos argumentos, um objeto do tipo `Map` por meio da variável `options`. Esse mapa é montado justamente a partir das opções que passamos na configuração em XML. O método `initialize` é definido pela especificação, e sempre é chamado para inicializar os valores necessários para cada `LoginModule`.

Agora fica mais claro o motivo das opções que passamos no arquivo de configuração serem no estilo *chave -> valor*. Também que os nomes ali usados

são específicos do container que você estiver usando.

```
<login-module code="Database" flag="required">
  <module-option name="dsJndiName"
    value="java:jboss/datasources/casadocodigoDS"/>

  <module-option name="principalsQuery"
    value="select password from SystemUser
          where e-mail=?"/>

  <module-option name="rolesQuery"
    value="select roles_name,'Roles'
          from SystemUser_SystemRole as user_roles
          inner join SystemUser as su
          on su.id = user_roles.SystemUser_id
          where su.e-mail = ?"/>

  <module-option name="hashAlgorithm"
    value="SHA-256"/>

  <module-option name="hashEncoding"
    value="base64"/>

</login-module>
```

Agora, vamos dissecar um pouco as opções para entender exatamente o que significa cada uma:

- `dsJndiName` – nome JNDI do nosso `datasource`, que já está configurado.
- `principalsQuery` – SQL necessário para recuperar um usuário baseado no login. No nosso caso, foi utilizado o e-mail.
- `rolesQuery` – SQL necessário para recuperar os perfis de determinado usuário, baseado no seu login.
- `hashAlgorithm` – algoritmo de hash que vai ser aplicado na senha para fazer a checagem no banco.

- `hashEncoding` – algoritmo usado para realizar o *encoding* da senha.

Um detalhe que pode chamar atenção é o uso da coluna inexistente `Roles` na query que recupera os perfis. Aqui é apenas um detalhe mais técnico imposto pelo próprio JAAS. Dentro da especificação, tanto o usuário logado quanto os perfis associados a ele são representados como classes que implementam a interface `Principal`.

O JAAS nos permite que agrupemos esses objeto em grupos de acesso. Veja alguns exemplos:

- Usuário administrador que pode fazer tudo no sistema;
- Usuário administrador que não pode deletar livros.

Esses grupos são representados por implementações da interface `Group`. O WildFly lhe obriga a retornar na query que carrega os perfis, a quais grupos cada uma delas pertence. Caso a aplicação não faça uso dos grupos, devemos especificar uma espécie de grupo padrão, e é exatamente por isso que adicionamos a coluna `Roles`. Já a palavra tem de ser `Roles`, por ser uma exigência do WildFly.

Associando a aplicação com um security domain

Mesmo configurando o novo `security-domain` no WildFly, ainda não conseguimos efetuar o login dentro da aplicação. O último passo é justamente informar que nossa aplicação precisa fazer uso do domínio configurado. Para que isso aconteça, é necessário que o projeto crie um arquivo chamado `jboss-web.xml`, na pasta `WEB-INF`.

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
  <security-domain>database-login</security-domain>
</jboss-web>
```

Lembre-se de que esse é um arquivo específico do WildFly!

Pronto, agora já conseguimos efetuar o login da nossa aplicação. Tente acessar os endereços que estão protegidos e perceba que tudo vai funcionar como deveria.

Caso você cadastre outros usuários com outros perfis, configure uma página de erro com o status 403, no `web.xml`. Dessa forma, quando alguém tentar acessar um endereço não permitido, será levado para uma página informativa.

Outra funcionalidade que você pode implementar é a de cadastro de novos usuários administradores. Lembre-se apenas de proteger o seu acesso!

12.4 EXIBINDO O USUÁRIO LOGADO E ESCONDENDO TRECHOS DA PÁGINA

Um detalhe comum é exibir o login do usuário nas páginas que ele acessa dentro do sistema. Por exemplo, queremos exibir o e-mail dele na página de listagem de livros.

```
<html xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
  <h:body>
    <div>
      Seja bem vindo, e-mailDoUsuario
    </div>
    ...
  </h:body>
</html>
```

O problema é: de onde vem essa informação? Na verdade temos mais coisas a pensar. Perceba que toda configuração que fizemos relativa ao JAAS não envolveu, em nenhum momento, as classes do nosso sistema. Tudo foi realizado usando os arquivos de configuração do servidor da aplicação.

O máximo de informação que conseguimos recuperar do usuário logado é o login passado por ele que, nesse caso, é justamente o e-mail. Para recuperar o objeto que contém essa informação, podemos usar diretamente o *request*.

```
<div>
  Seja bem vindo, #{request.getUserPrincipal().getName()}
</div>
```


Caso seja apenas essa a informação que você precisa do seu usuário logado, esse código é mais do que suficiente. O problema é que quase nunca você quer apenas isso. Muitas vezes vamos querer saber mais informações, como o nome do usuário, data de último login etc.

Para conseguirmos fazer isso, vamos precisar carregar as informações pertinentes a nossa entidade da aplicação, nesse caso a `SystemUser`. O melhor jeito de fazer isso é criando uma nova classe na aplicação e isolando essa lógica lá dentro.

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

//imports

@Model
public class CurrentUser {

    @Inject
    private HttpServletRequest request;
    @Inject
    private SecurityDAO securityDAO;
    private SystemUser systemUser;

    public SystemUser get(){
        return this.systemUser;
    }

    @PostConstruct
    private void loadSystemUser() {
        Principal principal = request.getUserPrincipal();
        if (principal != null) {
            this.systemUser = securityDAO
                .loadUserByUsername(principal.getName());
        }
    }
}
```

Lembre-se de que a annotation `@Model` já informa ao CDI que os objetos criados a partir desta classe devem viver no escopo de uma requisição web, e

que também devem ficar expostos na *expression language*. O objeto do tipo `Principal` representa o usuário que acabou de se logar no sistema e, como já vimos, recuperamo-lo por meio do *request*. O método `getName` retorna justamente o login do usuário.

Colocamos o código de carregamento dentro de um método anotado com `@PostConstruct`. Perceba que só carregamos de fato o usuário, caso ele esteja logado. Para não ficarmos com código sem explicação, vamos apenas dar uma olhada na classe `SecurityDAO`.

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

//imports

public class SecurityDAO {

    @PersistenceContext
    private EntityManager em;

    public SystemUser loadUserByUsername(String username)
        throws UsernameNotFoundException {
        String jpql = "select u from SystemUser u
            where u.e-mail = :login";
        SystemUser user = em.createQuery(jpql, SystemUser.class)
            .setParameter("login", username)
            .getSingleResult();
        return user;
    }
}
```

Agora, na nossa página, basta que usemos o objeto exposto pelo CDI na *expression language*.

```
<div>
    Seja bem vindo, #{currentUser.get().e-mail}
</div>
```

Lembre-se de que o padrão é expor o objeto associado a uma chave com o mesmo nome da classe, mas com a primeira letra minúscula.

Um ponto que pode ter chamado a atenção do leitor mais curioso é o fato de a classe `CurrentUser` não usar o escopo de sessão, já que ela guarda referência para o usuário que acabou de logar. O motivo é simples: já que as informações do usuário atual são mantidas no *request*, optamos por deixar a nossa classe no mesmo escopo.

A parte potencialmente ruim é que vamos fazer a query de busca de usuário o tempo todo. Entretanto, isso só será um problema se a sua aplicação realmente começar a apresentar problemas de performance. Caso o leitor prefira usar o escopo de sessão, fique à vontade para alterar a configuração da classe.

O código ficaria parecido com:

```
@Named
@SessionScoped
public class CurrentUser implements Serializable{
    ...
}
```

O único detalhe em que você precisa prestar atenção é com o objeto carregado pelo `EntityManager`. Uma vez que ele for fechado, o objeto entra no estado *detached*, e qualquer tentativa de *lazy load* vai gerar uma *exception*. Para essa situação em específico, o melhor cenário é usar uma query planejada, já trazendo todas as informações necessárias.

O uso do `EntityManager extended` é desencorajado, porque o objeto vai ficar no escopo de sessão, e a conexão com o banco seria mantida por um tempo maior do que nós gostaríamos.

Para este autor, manter o objeto no escopo de *request* é o mais simples e mais precavido. Por exemplo, caso o usuário logado atualize alguma informação, esta já será refletida no próximo *request*, uma vez que estamos fazendo o carregamento por requisição.

Apenas para lembrar: quase nunca precisamos começar a pensar na solução já considerando um requisito de performance. Tente ir pelo caminho mais simples e analise os resultados.

Menu de administração na home da Loja

Agora que temos uma área de administração protegida, podemos facilitar o acesso a ela. Nesse momento, todas as vezes em que o usuário quer acessar

o cadastro de livros, ele tem de digitar o endereço no navegador. Para facilitar, vamos adicionar mais uma opção no menu da página inicial da Casa do Código. Lembre-se de que o arquivo é o `site/index.xhtml`.

```
<nav id="main-nav">

    <ul class="clearfix">
        <li><a href="#{request.contextPath}/livros/list.xhtml"
            rel="nofollow">Administração</a></li>

        <li><a href="/cart" rel="nofollow">
            Seu carrinho
        </a></li>

        <li><a href="/pages/sobre-a-casa-do-codigo"
            rel="nofollow"> Sobre nós
        </a></li>

        <li><a href="/pages/perguntas-frequentes"
            rel="nofollow"> Perguntas Frequentes
        </a></li>

    </ul>
</nav>
```

O problema é que agora qualquer usuário pode ver esse menu, mas ele só deve ser exibido para os administradores. Podemos mudar um pouco o código para fazer essa verificação.

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
    xmlns:o="http://omnifaces.org/ui"
    xmlns:c="http://java.sun.com/jsp/jstl/core">

    ...

    <nav id="main-nav">
```

```

        <ul class="clearfix">
            <c:if test="#{currentUser.hasRole('ADMIN')}}">
                <li><a href="#{request.contextPath}/
                    livros/list.xhtml"
                    rel="nofollow">Administração</a></li>
            </c:if>

            <li><a href="/cart" rel="nofollow">
                Seu carrinho
            </a></li>

            <li><a href="/pages/sobre-a-casa-do-codigo"
                rel="nofollow"> Sobre nós
            </a></li>

            <li><a href="/pages/perguntas-frequentes"
                rel="nofollow"> Perguntas Frequentes
            </a></li>

        </ul>
    </nav>

    ...
</html>

```

Adicionamos mais um método na classe `CurrentUser`, o `hasRole`.

```

package br.com.casadocodigo.loja.security;

//imports

@Model
public class CurrentUser {

    @Inject
    private HttpServletRequest request;
    @Inject
    private SecurityDAO securityDAO;
    private SystemUser systemUser;

```

```
public SystemUser get(){
    return this.systemUser;
}

public boolean hasRole(String name) {
    return request.isUserInRole(name);
}

@PostConstruct
private void loadSystemUser() {
    Principal principal = request.getUserPrincipal();
    if (principal != null) {
        this.systemUser = securityDAO
            .loadUserByUsername(principal.getName());
    }
}
```

A implementação é mais fácil do que a gente poderia pensar. Como a API de Servlets é integrada com o JAAS, o próprio *request* já fornece o método *isUserInRole* para verificar se o usuário que se logou possui determinado perfil. Lembre-se de que quando damos início ao processo de login, o módulo de segurança configurado no WildFly será usado e os perfis do usuário serão carregados. Pronto, dessa forma, garantimos que a opção do menu só aparece para os usuários que estão logados como administradores.

Apenas para não passar em branco, foi utilizada a tag `if` da JSTL para verificarmos se o usuário possui o perfil ou não. Como só trabalhamos com elementos HTML padrão, não existe nenhuma restrição quanto ao uso dessa tag.

Para não perdermos o foco do capítulo, existe um texto muito bom, um pouco antigo, que detalha bem a mistura de tags da JSTL com componentes do JSF. Acesse o link <http://bit.ly/jstl-jsf> e entenda ainda mais sobre questões do ciclo de vida.

12.5 CONCLUSÃO

Neste capítulo, estudamos bastante sobre o JAAS. Apesar de ser uma especificação antiga e pouco atualizada, conseguiu atender nossas necessidades da nossa aplicação. Fique apenas atento a todas as configurações que fizemos no WildFly foram específicas dele, e não podem ser reaproveitadas quando você for utilizar outro servidor de aplicação.

Esse é justamente um dos pontos que vão ser atacados na próxima versão do Java EE. A API de segurança vai ser modernizada para que fique mais fácil de ser configurada. Você pode acompanhar o andamento em <https://jcp.org/en/jsr/detail?id=375>.

Um último ponto que pode, no primeiro momento, causar um pouco de desconfiança é o fato de os endereços protegidos serem especificados diretamente no `web.xml`. Esse, na verdade, é o cenário ideal. Tentar manter suas regras simples; é o melhor que você pode fazer. Por exemplo, se as URLs estivessem cadastradas no banco, como você teria certeza de que os endereços necessários realmente estão protegidos?

Por mais que pareça tentador deixar tudo dinâmico, tente começar pelo simples. Caso realmente precise que as associações entre perfis e URLs sejam dinâmicas, você vai ter de partir para uma solução mais manual, já que o JAAS não possui essa funcionalidade pronta para uso.

Não pare agora, no próximo capítulo vamos tentar modularizar nosso layout, diminuindo as partes que se repetem!

CAPÍTULO 13

Organização do layout em templates

Já implementamos algumas das funcionalidades que existem dentro do sistema real do site da Casa do Código. Entretanto, uma parte com que não nos preocupamos até agora é em relação à organização do layout da aplicação. E esse é um ponto bem relevante, já que o layout também sofre alterações durante o tempo de vida da aplicação, levando-nos, várias vezes, a problemas de manutenção justamente por não dar a devida importância a essa parte.

Por exemplo, atualmente, entre as páginas que já implementamos, temos a de exibição dos detalhes de um livro, a que exibe os produtos do carrinho e a que lista os livros disponíveis na loja. Olhando com atenção para elas, vamos perceber que ambas possuem o mesmo `header` e `footer`.

`<head>`

```

    ...
</head>
<header id="layout-header">
    ...
</header>
<nav class="categories-nav">
    ...
</nav>

<!-- resto da página-->

<footer id="layout-footer">
    ...
</footer>

```

Aqui temos o problema clássico de repetição. Caso alguma mudança seja necessária nesses trechos, vamos ter de sair caçando em todas as páginas. Uma maneira comum de resolver essa situação é utilizando o sistema de *includes*, que já é suportado em aplicações JSF, por meio do *Facelets*.

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:o="http://omnifaces.org/ui"
      xmlns:c="http://java.sun.com/jsp/jstl/core">

  <head>
    <meta charset="utf-8"/>
    <meta http-equiv="X-UA-Compatible" content="IE=edge,
      chrome=1"/>
    <meta name="viewport" content="width=device-width,
      initial-scale=1, maximum-scale=1"/>
    ...

  </head>
  <body class="index">

    <ui:include src="header.xhtml"/>

    ...resto da pagina

```

```
<ui:include src="footer.xhtml"/>
</body>
</html>
```

Até isolamos o `header` e o `footer`, só que sobrou esse início de declaração do HTML. Aí você pode pensar: *não tem problema, é só criar mais uma `include` e isso também vai estar isolado*. Só que aí é que entra um outro problema: para uma pessoa nova montar uma página, seguindo o padrão do sistema, ela tem de saber de todas essas *includes* e ainda tem de saber a ordem! A tendência é só piorar.

13.1 TEMPLATES

Para amenizar esse problema, a especificação do JSF já pensou em uma solução. A ideia é que possamos criar um modelo de página que possa ser usado por todas as outras telas. Esse é mais um mecanismo suportado pelo *Facelets*.

Vamos dar uma olhada em como vai ficar a página `site/index.xhtml`, utilizando essa espécie de molde.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:o="http://omnifaces.org/ui"
      xmlns:c="http://java.sun.com/jsp/jstl/core">

  <ui:composition template="/site/_template.xhtml">
    <ui:define name="extraCss">
      <link href=
        "#{request.contextPath}/resources/css
          /book-collection.css" rel="stylesheet"
        type="text/css" media="all" />
    </ui:define>

    <ui:define name="body">

      <section id="index-section"
        class="container middle">
```

```

        ...
    </section>
</ui:define>
</ui:composition>

</html>

```

A tag `composition` é usada para indicar que queremos usar um modelo padrão de página, também conhecido como *template*. Perceba que até usamos um atributo também chamado `template`, e passamos o caminho do arquivo que servirá de modelo para as páginas da nossa loja.

Antes de analisar o arquivo que vai servir de base, vamos continuar analisando as outras tags que usamos. A tag `define` deve ser utilizada para informar que a página atual passará alguns trechos de código que devem ser usados pela *master page*, de modo a completar a estrutura da página.

- `extraCss` – usamos para passar o HTML necessário com os estilos específicos da página;
- `body` – usamos para passar o HTML específico da página atual.

Podemos ter quantas tags `define` forem necessárias, tudo vai depender de como o `template` foi organizado. Vamos agora dar uma olhada no arquivo `site/_template.xhtml` para ver como podemos montar a estrutura da página.

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:o="http://omnifaces.org/ui"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <head>
        <ui:include src="head.xhtml"/>

        <ui:insert name="extraCss"/>

```

```
</head>
<body class="index">

    <ui:include src="header.xhtml"/>

    <ui:insert name="body"/>

    <ui:include src="footer.xhtml"/>
</body>
</html>
```

Perceba que é uma página normal, a única tag nova que usamos foi a `insert`. Ela serve justamente para demarcamos os pontos da página que serão preenchidos pela página que for usar esse template, como foi o caso da `site/index.xhtml`.

O valor do atributo `name` da tag `insert` tem de ser exatamente o mesmo do atributo `name` da tag `define`, para que o Facelets consiga fazer a associação e montar a página final.

Um último detalhe, não menos importante: as tags novas que usamos estão disponíveis no namespace <http://xmlns.jcp.org/jsf/facelets>, que foi referenciado pelo prefixo `ui` na nossa página.

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

    ...

</html>
```

Detalhe especial sobre a tag `composition`

Um ponto que pode ter ficado na sua cabeça foi o uso da tag `html` para importar as *taglibs* que usamos dentro das páginas `_template.xhtml` e `site/index.xhtml`. Analisando, fica parecendo que a tag `html` vai ser renderizada duas vezes: uma pelo template e outra pela página específica.

É justamente aí que entra o uso da tag `composition`. Tudo o que está fora dela é ignorado no momento da renderização da página e, por conta

disso, não corremos o risco de ter as tags duplicadas no resultado final.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:o="http://omnifaces.org/ui"
      xmlns:c="http://java.sun.com/jsp/jstl/core">

  <!-- Tudo que tem daqui pra cima é ignorado-->
  <ui:composition template="/site/_template.xhtml">
    ...
  </ui:composition>
  <!-- Tudo que tem daqui pra baixo é ignorado-->
</html>
```

Passando parâmetros para o template

Um outro detalhe que talvez tenha passado despercebido foi o uso do atributo `class`, na tag `body` do template principal.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:o="http://omnifaces.org/ui"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <head>
    <ui:include src="head.xhtml"/>

    <ui:insert name="extraCss"/>

  </head>
  <body class="index">

    <ui:include src="header.xhtml"/>

    <ui:insert name="body"/>

    <ui:include src="footer.xhtml"/>
```

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

Ela sempre está assumindo que a página que utilizará o template será a `index.xhtml`. O problema é que cada página tem a sua classe específica, por exemplo, a tela com os itens do carrinho usa a classe `cart`. Esse é um caso em que não podemos usar a tag `insert`, já que não queremos definir um trecho de HTML, mas sim apenas um parâmetro.

Para este tipo de situação, vamos usar a tag `param`. Vamos dar uma olhada como ficaria a tela do carrinho de compras:

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

  <ui:composition template="/site/_template.xhtml">
    <ui:param name="bodyClass" value="cart"/>
    <ui:define name="body">
      ...
    </ui:define>
  </ui:composition>
</html>
```

Perceba que usamos a tag `param` para definir um parâmetro de nome `bodyClass`. Agora, no template, basta que acessemos a variável que foi definida, via *expression language*.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:o="http://omnifaces.org/ui"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <head>
    <ui:include src="head.xhtml"/>
```

```
<ui:insert name="extraCss"/>

</head>
<body class="#{bodyClass}">

    <ui:include src="header.xhtml"/>

    <ui:insert name="body"/>

    <ui:include src="footer.xhtml"/>
</body>
</html>
```

Pronto! Agora temos um layout bem organizado e pronto para ser evoluído. Podemos alterar a ordem das informações, e não será necessário alterar nenhum arquivo que use o template. Além disso, os trechos definidos pela tag `insert` não são de passagem obrigatória, o que fornece ainda mais flexibilidade.

13.2 CONCLUSÃO

Este capítulo foi bem rápido, apenas organizamos um pouco mais nosso layout. Não usamos nenhuma especificação nova em si, mas mesmo assim foi importante, já que as telas compõem uma parte fundamental de qualquer aplicação.

Não pare agora, já comece o próximo capítulo para que possamos fazer com que nossa aplicação suporte vários idiomas.

Apenas como lembrete: todos os arquivos de layout podem ser encontrados no repositório do livro, em <https://github.com/asouza/casadocodigojavaee>.

Você também pode acessar o *commit* específico da organização do layout. É só acessar o link <http://bit.ly/commit-facelets>.

CAPÍTULO 14

Internacionalização

Até este momento, todos os textos da aplicação estão escritos diretamente nas páginas. Por exemplo, vamos analisar o menu de navegação da *include* que contém o `header` do projeto.

```
<div id="header-content">
  <nav id="main-nav">

    <ul class="clearfix">
      <c:if test="#{currentUser.hasRole('ADMIN')}">
        <li><a href="#{request.contextPath}/livros/
          /list.xhtml" rel="nofollow"> Administração
        </a></li>
      </c:if>

      <li><a href="/cart" rel="nofollow">
```

```

        Seu carrinho
    </a></li>

    <li><a href="/pages/sobre-a-casa-do-codigo"
        rel="nofollow">
        Sobre nós
    </a></li>

    <li><a href="/pages/perguntas-frequentes"
        rel="nofollow">
        Perguntas Frequentes
    </a></li>
</ul>
</nav>
</div>

```

Todos os textos da aplicação estão escritos diretamente na página e, em geral, talvez isso não seja um problema. Entretanto, o site da Casa do Código já tem a sua versão internacional, a *Code Crushing*. Eles têm o mesmo layout e só mudam em uma coisa: todos os textos são escritos em inglês.

14.1 ISOLANDO OS TEXTOS EM ARQUIVOS DE MENSAGENS

Caso continuemos com a estratégia de escrever os textos diretamente nas páginas, teríamos de duplicar cada uma delas. Para conseguirmos **internacionalizar** nossa aplicação, vamos ter de isolar os textos em um novo arquivo de propriedades, dessa vez o `messages.properties`.

```

menuPrincipal_admin = Administração
menuPrincipal_seuCarrinho = Carrinho
menuPrincipal_sobre = Sobre
menuPrincipal_perguntasFrequentes = Perguntas frequentes

```

Usamos um novo arquivo, já que o outro que criamos, o `jsf_messages`, foi configurado para ter as mensagens específicas do JSF, e não da aplicação em si. Esse novo arquivo também deve ficar no `classpath` do projeto, por

isso que ele deve ser criado em `src/main/resources`. A nomenclatura em um arquivo `properties` sempre é fonte de discussão.

Aqui estamos seguindo o seguinte: dividimos os grupos com o `_` e usamos o *camelCase* para os nomes dos itens em si. Agora que já isolamos os textos no arquivo `messages`, podemos usar as chaves nas nossas páginas.

```
<div id="header-content">
  <nav id="main-nav">

    <ul class="clearfix">
      <c:if test="#{currentUser.hasRole('ADMIN')}}">
        <li><a href="#{request.contextPath}/livros
          /list.xhtml" rel="nofollow">
          menuPrincipal_admin
        </a></li>
      </c:if>

      <li><a href="/cart" rel="nofollow">
        menuPrincipal_seuCarrinho
      </a></li>

      <li><a href="/pages/sobre-a-casa-do-codigo"
        rel="nofollow">
        menuPrincipal_sobre
      </a></li>

      <li><a href="/pages/perguntas-frequentes"
        rel="nofollow">
        menuPrincipal_perguntasFrequentes
      </a></li>
    </ul>
  </nav>
</div>
```

O problema é que precisamos acessar os valores associadas a essas chaves. Para isso, precisamos ensinar o JSF a carregar o arquivo e deixar as chaves disponíveis para serem usadas nas páginas. Vamos alterar o arquivo `faces-config.xml`, para adicionar a tag `resource-bundle`.

```

<application>
  <message-bundle>jsf_messages</message-bundle>
  <resource-bundle>
    <base-name>messages</base-name>
    <var>msg</var>
  </resource-bundle>
</application>

```

A tag `base-name` indica a localização do arquivo, dentro do `classpath` do nosso projeto. Como deixamos o arquivo em `src/main/resources`, que fica na raiz do `classpath`, só precisamos passar o nome. A tag `var` é utilizada para informar o nome da variável que ficará disponível para acessarmos nas páginas.

```

<div id="header-content">
  <nav id="main-nav">

    <ul class="clearfix">
      <c:if test="#{currentUser.hasRole('ADMIN')}">
        <li><a href=
          "#{request.contextPath}/livros/list.xhtml"
          rel="nofollow">#{msg.menuPrincipal_admin}
        </a></li>
      </c:if>

      <li><a href="/cart" rel="nofollow">
        #{msg.menuPrincipal_seuCarrinho}
      </a></li>

      <li><a href="/pages/sobre-a-casa-do-codigo"
        rel="nofollow">
        #{msg.menuPrincipal_sobre} </a></li>

      <li><a href="/pages/perguntas-frequentes"
        rel="nofollow">
        #{msg.menuPrincipal_perguntasFrequentes}
      </a></li>
    </ul>
  </nav>

```

```
</div>
```

A variável `msg` guarda a referência para um mapa carregado com as informações do arquivo de propriedades. O acesso às chaves por meio da sintaxe `{msg.chave}` é apenas um *syntax sugar* para `{msg[chave]}`.

14.2 ACCEPT-LANGUAGE HEADER

Até agora, temos apenas o arquivo `messages.properties` com os textos em português. Só que precisamos suportar as mensagens também em inglês. Para fazer isso, precisamos criar o arquivo com a extensão da localização que queremos suportar. Por exemplo, podemos criar o arquivo `messages_en.properties`.

```
menuPrincipal_admin = Admin
menuPrincipal_seuCarrinho = Shopping Cart
menuPrincipal_sobre = About
menuPrincipal_perguntasFrequentes = Frequent questions
```

Podemos também criar o arquivo com a extensão `_pt.properties`, para suportar o português padrão. Para que o JSF saiba quais locais ele deve aceitar, precisamos adicionar mais uma configuração no `faces-config`.

```
<application>
  <message-bundle>jsf_messages</message-bundle>
  <resource-bundle>
    <base-name>messages</base-name>
    <var>msg</var>
  </resource-bundle>
  <locale-config>
    <default-locale>pt</default-locale>
    <supported-locale>en</supported-locale>
  </locale-config>
</application>
```

Definimos qual a língua padrão e ainda adicionamos as outras que são suportadas. A pergunta que fica é: *como o JSF descobre qual idioma o navegador prefere?*

Fig. 14.1: Header com a opção de língua

A resposta está no cabeçalho da requisição HTTP. Perceba que há uma chave chamada `Accept-Language`, e é nela que vêm as línguas prediletas do seu usuário. No caso do navegador do autor deste livro, está como português. Tanto que a primeira opção é o `pt`. Porém, caso nosso sistema não suporte esse idioma, ele fala que prefere o `en-US`.

Carregar o arquivo de mensagens em função da língua do navegador é a maneira padrão de o JSF lidar com a internacionalização.

14.3 PASSANDO PARÂMETROS NAS MENSAGENS

Outra situação muito comum no momento de exibir as mensagens internacionalizadas é a necessidade de que a mensagem seja construída dinamicamente. Por exemplo, na página de listagem de livros, queremos dar boas-vindas ao usuário que acabou de logar no sistema.

```
Seja bem vindo, #{currentUser.get().email}
```

Vamos deixar inclusive essa mensagem internacionalizada. Para isso, vamos acrescentar a entrada no arquivo de mensagens.

```
usuarios.bemVindo = Olá {0}
```

Para representar os parâmetros, usamos os índices. Esta técnica já vem suportada por padrão no suporte à internacionalização presente no próprio Java. Agora, para usar na página, podemos usar a tag do JSF.

```
<h:outputFormat value="#{msg.usuarios_bemVindo}">  
    <f:param value="#{currentUser.get().email}" />  
</h:outputFormat>
```

A tag `outputFormat` deve ser utilizada sempre que você quiser escrever um texto parametrizado. Passamos o texto com os parâmetros para o atributo `value`, e fazemos uso da tag `param` para ir definindo os valores de cada índice. Caso você tenha mais de um índice, passe o número equivalente de tags `param`.

14.4 DEIXE QUE O USUÁRIO DEFINA A LÍNGUA

Respeitar o idioma sugerido pelo navegador é uma ótima forma de tentarmos acertar a língua preferida pelo usuário. O problema dessa abordagem é se o idioma configurado não for realmente o seu preferido.

Nesse momento, caso você esteja achando que todo mundo pode trocar isso facilmente, pense em todas as pessoas que você já conheceu que não tinham muito conhecimento de computação. Muitos usuários, por exemplo, apesar de usarem diariamente um navegador, não sabem como fazer essa troca.

Uma estratégia adotada por alguns sites é a de oferecer links para que o próprio usuário possa definir o idioma predileto.

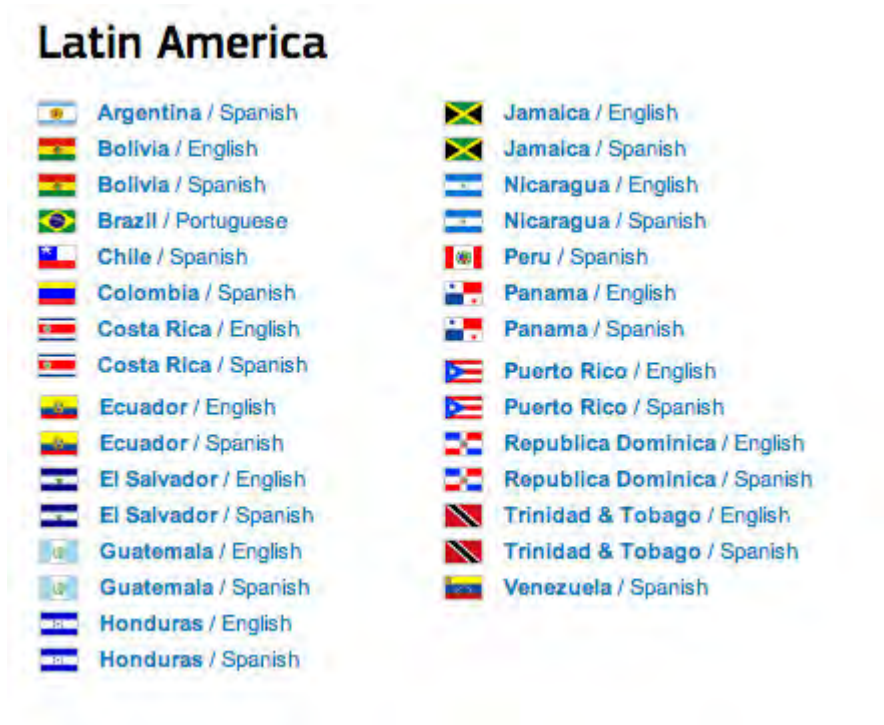


Fig. 14.2: Bandeiras dos países

Vamos adotar a mesma abordagem para a Casa do Código. Inicialmente, vamos adicionar os links para que o usuário possa escolher. Faremos isso no arquivo `header.xhtml`:

```
<nav id="main-nav">

    <ul class="clearfix">

        <li>
            <h:form>
                <h:commandLink action="#{i18nBean
                    .changeLocale('pt')}"
                    value="#{msg.lingua_pt}" />
            </h:form>
        </li>
    </ul>
</nav>
```



```

    </li>
    <li>
        <h:form>
            <h:commandLink action="#{i18nBean
                            .changeLocale('en')}"
                            value="#{msg.lingua_en}" />
        </h:form>
    </li>
    ...
</nav>

```

Já até definimos os links com as actions para um *bean* nosso, que vai ser o responsável por trocar o idioma da aplicação.

```

package br.com.casadocodigo.loja.managedbeans.admin;

//imports

@Model
public class I18nBean implements Serializable {

    @Inject
    private FacesContext context;

    public String changeLocale(String language){
        context.getApplication()
            .setDefaultLocale(new Locale(language));
        return "/site/index.xhtml?faces-redirect=true";
    }
}

```

O código anterior até funciona, mas o problema é que trocamos o `locale` para a aplicação toda. Na verdade, o que queremos é trocar o idioma apenas para o uso de um usuário.

A melhor maneira de fazer isso é mantendo o nosso bean no escopo de *session* em vez do escopo de *request*, para que o idioma setado seja mantido enquanto o usuário estiver navegando pela aplicação.

```

@Named
@SessionScoped
public class I18nBean implements Serializable {

    private Locale locale;
    @Inject
    private FacesContext context;

    public String changeLocale(String language){
        this.locale = new Locale(language);
        return "/site/index.xhtml?faces-redirect=true";
    }

    public Locale getLocale() {
        return locale;
    }
}

```

O detalhe estranho desse código é que em momento algum invocamos um método da API do JSF. Esse é justamente nosso trabalho, é necessário fazer com o que o `locale` escolhido pelo usuário seja passado para o JSF em toda requisição! É isso mesmo, o JSF não provê uma maneira padrão de manter o idioma escolhido entre várias requisições.

Para fazer isso, podemos usar a tag `view`. Como todas as telas da loja são baseadas no template definido no arquivo `_template.xhtml`, podemos alterá-lo para que o `locale` seja configurado para cada página.

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:o="http://omnifaces.org/ui"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <f:view locale="#{i18nBean.locale}"/>
    ...

```

As tags do namespace `http://java.sun.com/jsf/core`, cujo o prefixo padrão é

f, são geralmente usadas quando queremos fazer configurações que não têm a ver exatamente com a página, mas sim com alguma configuração do JSF. O atributo `locale` recebe como argumento um objeto do tipo `Locale`, justamente o que é retornado pelo método `getLocale` da nossa classe `I18nBean`.

Dessa forma, conseguimos sempre definir o `locale` para cada página acessada, e ainda permitimos que o usuário da aplicação o altere sempre que quiser.

Uma última pergunta que pode estar na sua cabeça é: *e quando o usuário acessar a loja pela primeira vez, qual o `locale` que será usado?* Para resolver isso, podemos recuperar o idioma padrão, que foi configurado no `faces-config`, e guardar no nosso atributo.

```
@Named
@SessionScoped
public class I18nBean implements Serializable {

    private Locale locale;
    @Inject
    private FacesContext context;

    @PostConstruct
    private void loadLDefaultLocale(){
        this.locale =
            context.getApplication().getDefaultLocale();
    }

    public String changeLocale(String language){
        this.locale = new Locale(language);
        return "/site/index.xhtml?faces-redirect=true";
    }

    public Locale getLocale() {
        return locale;
    }
}
```

Agora, toda vez que o usuário acessar a aplicação pela primeira vez, ele verá a página em português. Caso ele prefira outro idioma, basta clicar no link!

14.5 CONCLUSÃO

Este capítulo também foi tranquilo. Internacionalização é um tema muito comum entre todas as aplicações, lembre-se apenas de não ser radical. Caso você não tenha planos de fazer outras versões, não se preocupe com esse detalhe. Só o use se realmente for necessário, afinal de contas, você está deixando de dar manutenção só em um lugar para dar em dois! Antes era apenas a página, agora é a página e o arquivo de mensagens.

No próximo capítulo, discutiremos sobre uma técnica que está em evidência, a utilização de *WebSockets*. Através dela, vamos disparar notificações para o nosso navegador de uma maneira muito simples! Por conta disso, não pare de ler agora, e já devore o próximo capítulo.

CAPÍTULO 15

Enviando e recebendo informações via WebSocket

Uma funcionalidade desejada pela loja é a de poder avisar os clientes sobre promoções relâmpago. A ideia é que o administrador possa decidir que algum livro fique em promoção em determinado instante, e que todos os usuários que estejam em alguma página naquele momento sejam notificados.

15.1 COMO NOTIFICAR OS USUÁRIOS?

A pergunta que fica na nossa cabeça é: *como vamos enviar alguma coisa para a página que está sendo acessada pelo usuário, naquele instante?* Uma das soluções mais antigas era implementar um JavaScript que fica, de tempos em tempos, consultando uma URL do servidor para saber se algo novo aconte-

ceu.

```
//exemplo usando um pouco de jquery
setInterval(function(){
    $.ajax({
        url: "/sales",
        success: function(data){
            //Atualiza o valor de cada ação na tabela
        }, dataType: "json"});
}, 30000);
```

Poderíamos criar um *resource* do JAX-RS que responderia para essa requisição. O grande problema dessa abordagem é que o cliente fica o tempo inteiro disparando requisição contra o servidor, mas na maioria das vezes não tem nada de novo acontecendo! Basicamente, estamos metralhando o servidor com várias requisições inúteis.

15.2 API DE WEBSOCKETS E O NAVEGADOR

Na verdade, o que precisamos é que quando alguma promoção seja cadastrada, o servidor notifique o cliente com a novidade. Para contornar o problema de ficar fazendo requisição o tempo inteiro, ainda antigamente, os servidores começaram a suportar que certas conexões abertas pelo cliente não fossem fechadas, técnica conhecida como *Comet*.

Essa foi uma técnica criada justamente para os servidores começarem a suportar o chamado **Long Polling** e, dessa forma, pudessem enviar informações para o cliente sem a necessidade de uma nova requisição.

Pois bem, na última versão do HTML5, esse tipo de técnica virou uma especificação chamada *WebSocket*. Só que eles foram além de só especificar o que já tinha pronto, e adicionaram novos detalhes:

- Além de o servidor poder notificar o cliente, o cliente também pode enviar informações para o servidor;
- A comunicação é feita baseada em um novo protocolo, chamado justamente de *WebSocket*;

- API padrão para ser usada dentro do navegador em vez de ficar simulando requisição AJAX.

Usando a API de WebSockets no navegador

Vamos começar a implementar nossa nova funcionalidade. A primeira coisa que precisamos é abrir o WebSocket na página para podermos mandar e receber informações pela mesma conexão. Como queremos que, independente da página, o usuário seja notificado, vamos deixar esse código no `_template.xhtml`.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:o="http://omnifaces.org/ui"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <f:view locale="#{i18nBean.locale}"/>
  <head>
    <ui:include src="head.xhtml"/>

    <ui:insert name="extraCss"/>

  </head>
  <body class="#{bodyClass}">

    <ui:include src="header.xhtml"/>

    <ui:insert name="body"/>

    <ui:include src="footer.xhtml"/>
    <script>
      var salesChannel = new WebSocket(
        "ws://localhost:8080/#{request.contextPath}
        /channel/sales");
    </script>
  </body>
</html>
```

Criamos um novo objeto do tipo `WebSocket`, passando justamente o endereço que precisamos manter o link aberto com o cliente. Perceba, inclusive, que o protocolo usado não é o HTTP, e sim o WS, de `WebSocket`.

Agora que a conexão está aberta, precisamos receber as novas mensagens, que serão enviadas a partir do servidor.

```
salesChannel.onmessage = function (message) {  
    var newSale = JSON.parse(message.data);  
    if(confirm("Quer participar da seguinte promocao: "  
        +newSale.title+"?")){  
        document.location.href =  
            "#{request.contextPath}/site/  
                detalhe.xhtml?id="+newSale.bookId;  
    }  
};
```

A propriedade `onMessage` aceita um função que vai ser invocada sempre que o servidor enviar uma nova mensagem para o cliente. Perceba que a nossa função simplesmente leva o usuário para a tela de detalhes do livro com o `id` igual ao que foi recebido como argumento. A API de WebSockets é muito direta, a abstração criada pela especificação realmente deixou tudo muito simples.

Um outro detalhe que vale analisar com mais carinho é o formato da mensagem. Ela sempre chega como um JSON e, por meio da propriedade `data`, conseguimos acessar o texto que foi enviado pelo servidor.

Também é interessante notar o console do navegador. Quando acessamos uma página que abre um `WebSocket`, a requisição fica aberta com o status 101, que representa a conexão em andamento.



Fig. 15.1: Conexão aberta via WebSocket

Caso você use o serviço do *WhatsApp Web*, abra o console da página e perceba que ele usa um WebSocket para ficar enviando e recebendo novas mensagens.

15.3 ENVIANDO MENSAGENS A PARTIR DO SERVIDOR

Como foi comentado, precisamos de uma nova tela na aplicação para permitir que o administrador da loja consiga lançar novas promoções no sistema. Podemos criar esse formulário em `promocoões/form.xhtml`.

```
<html xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:pt="http://xmlns.jcp.org/jsf/passthrough">
  <h:body>
    <h:messages />
    <h:form>
      <div>
        <h:outputLabel>Titulo</h:outputLabel>
        <h:inputText
          value="#{adminSalesBean.sale.title}"
          label="titulo"/>
      </div>
      <div>
        <h:outputLabel for="livro">
          Livro
        </h:outputLabel>
      </div>
    </h:form>
  </h:body>
</html>
```

```

        <h:selectOneMenu
            value="#{adminSalesBean.sale.book.id}"
            id="livro">
            <f:selectItems
                value="#{adminListBooksBean.books}"
                var="book"
                itemLabel="#{book.title}"
                itemValue="#{book.id}"/>
            </h:selectOneMenu>
        </div>

        <h:commandButton value="Gravar"
            action="#{adminSalesBean.save}"/>
    </h:form>
</h:body>
</html>

```

Em relação ao JSF, não temos nada de novo. O detalhe interessante está apenas no uso da tag `selectItems`. Precisamos da lista de livros cadastrados. Em vez de fazer uma nova lógica para isso, aproveitamos o *bean* que já existe.

Lembre-se sempre de que não existe nenhuma regra que impõe a necessidade de criar um *bean* para cada tela. Divida suas classes de forma a reaproveitar as responsabilidades, sem perder a coesão.

Além de reaproveitar a listagem de livros, precisamos de um novo *bean* que será responsável por tratar a solicitação de uma nova promoção.

```

package br.com.casadocodigo.loja.managedbeans.admin;

//imports

@Model
public class AdminSalesBean {

    private Sale sale = new Sale();

    @PostConstruct
    private void configure() {

```

```
        this.sale.setBook(new Book());
    }

    public String save() {
        //precisamos notificar os usuários sobre a promoção
        return "/admin/promocoes/form.xhtml?faces-redirect=true";
    }

    public Sale getSale() {
        return sale;
    }

    public void setSale(Sale sale) {
        this.sale = sale;
    }
}
```

De novo, apenas códigos que já estudamos! Para que nosso código compile, também é necessário que seja criada a classe `Sale`, a abstração de uma nova promoção no sistema.

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

//imports

public class Sale {

    private String title;
    private Book book;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

```
public Book getBook() {  
    return book;  
}  
  
public void setBook(Book book) {  
    this.book = book;  
}  
}
```

Agora, só falta resolvermos justamente a parte de enviar a mensagem via WebSocket. Um último detalhe: não se esqueça de adicionar o novo endereço na configuração do JAAS no seu `web.xml`. Não queremos que os usuários que não são administradores lancem promoções no sistema.

WebSockets no Java EE

Entendendo a nova tendência das aplicações web, a especificação Java EE não perdeu tempo e adicionou o suporte à comunicação via WebSocket. O mais legal de tudo isso é que realmente ficou muito simples suportar esse modelo dentro da nossa aplicação.

A nossa primeira tarefa é a de aceitar uma nova conexão via WebSocket.

```
package br.com.casadocodigo.loja.websockets;  
  
import javax.websocket.server.ServerEndpoint;  
  
@ServerEndpoint("/channel/sales")  
public class SalesEndpoint {  
  
}
```

É uma configuração bem parecida com a que fazemos quando precisamos de um *Servlet*. Só que, em vez de usarmos a annotation `@WebServlet`, usamos a nova annotation `@ServerEndpoint`. Também somos obrigados a passar o endereço em que a conexão via WebSocket será aberta. Perceba que a URL é a mesma utilizada dentro da nossa página.

Agora que a conexão está aberta, é necessário que o nosso *bean* de cadastro de promoções seja capaz de mandar uma mensagem para cada cliente que

abriu um novo WebSocket nesse `endpoint`. Para saber que uma nova conexão foi solicitada, basta que um método seja criado e anotado com `@OnOpen`.

```
...
import javax.inject.Inject;
import javax.websocket.OnOpen;
import javax.websocket.Session;

@ServerEndpoint("/channel/sales")
public class SalesEndpoint {

    @Inject
    private ConnectedUsers connectedUser;

    @OnOpen
    public void onNewUser(Session session){
        connectedUser.add(session);
    }
}
```

Podemos dar qualquer nome para o método, o importante é que ele esteja anotado com `OnOpen`. Além disso, podemos receber um parâmetro do tipo `Session`, que representa a conexão estabelecida entre o cliente e o servidor. Quando uma nova promoção for lançada, vamos precisar de todos objetos do tipo `Session` para que possamos notificar cada um deles do novo evento. Uma primeira solução poderia ser guardar esse conjunto dentro da própria classe `SalesEndpoint`.

```
@ServerEndpoint("/channel/sales")
public class SalesEndpoint {

    private Set<Session> remoteUsers = new HashSet<>();

    @OnOpen
    public void onNewUser(Session session){
        remoteUsers.add(session);
    }
}
```

O problema é que a especificação diz que, para cada nova conexão, um novo objeto da classe anotada com `@ServerEndpoint` será criado e, por conta disso, sempre vamos perder os clientes que já estavam adicionados no nosso `Set`.

Para resolver essa situação, podemos criar uma outra classe que será responsável por manter as referências aos usuários que estão conectados naquele momento.

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

//outros imports
import javax.websocket.Session;

@ApplicationScoped
public class ConnectedUsers {

    private Set<Session> remoteUsers = new HashSet<>();
    private Logger logger =
        LoggerFactory.getLogger(ConnectedUsers.class);

    public void add(Session remoteUser) {
        this.remoteUsers.add(remoteUser);
    }
}
```

O código é basicamente o mesmo que tínhamos escrito, só que agora mantemos a lógica dentro de uma classe anotada com `@ApplicationScoped`. Graças ao CDI, podemos injetar um objeto deste tipo dentro da nossa classe `SalesEndpoint`.

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

import javax.inject.Inject;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
```

```
@ServerEndpoint("/channel/sales")
public class SalesEndpoint {

    @Inject
    private ConnectedUsers connectedUsers;

    @OnOpen
    public void onNewUser(Session session){
        connectedUsers.add(session);
    }
}
```

Como o escopo da classe `ConnectedUsers` é de aplicação, conseguimos manter uma lista única com todos usuários conectados naquele momento. Agora que já temos quase tudo pronto, podemos voltar para o nosso *bean*, e finalmente disparar a mensagem!

```
package br.com.casadocodigo.loja.managedbeans.admin;

//imports

@Model
public class AdminSalesBean {

    private Sale sale = new Sale();
    @Inject
    private ConnectedUsers connectedUsers;

    @PostConstruct
    private void configure() {
        this.sale.setBook(new Book());
    }

    public String save() {
        connectedUsers.send(sale.toJson());
        return "/admin/promocoes
            /form.xhtml?faces-redirect=true";
    }
}
```

```
public Sale getSale() {  
    return sale;  
}  
  
public void setSale(Sale sale) {  
    this.sale = sale;  
}  
}
```

Apenas recebemos injetado o objeto do tipo `ConnectedUsers`, e invocamos o método `send` passando um JSON como argumento. O método `send` vai ser o responsável por enviar as mensagens para o cliente, também via WebSocket.

```
package br.com.casadocodigo.loja.websockets;  
  
import javax.inject.Inject;  
import javax.websocket.OnOpen;  
import javax.websocket.Session;  
import javax.websocket.server.ServerEndpoint;  
  
@ServerEndpoint("/channel/sales")  
public class SalesEndpoint {  
  
    @Inject  
    private ConnectedUsers connectedUsers;  
  
    @OnOpen  
    public void onNewUser(Session session){  
        connectedUsers.add(session);  
    }  
  
    public void send(String message) {  
        for (Session user : remoteUsers) {  
            if (user.isOpen()) {  
                try {  
                    user.getBasicRemote().sendText(message);  
                }  
            }  
        }  
    }  
}
```



```
        } catch (IOException e) {  
            logger.error("Não foi possível enviar  
                mensagem para um cliente, {}", e);  
        }  
    }  
}  
}
```

O método `getBasicRemote` retorna o objeto capaz de mandar mensagens para o cliente conectado. Por fim, o método `sendText` é o responsável por enviar uma mensagem no formato texto.

É tudo tão simples que parece até estranho, mas como já mostramos durante o livro, o Java EE caminhou para um alto nível de simplicidade. A maioria das especificações já está baseada em APIs de uso fácil, e as mais antigas, como o JAAS, estão sendo atualizadas para virem mais simples no Java EE 8.

Para não ficarmos com um código sem explicação, vamos apenas olhar o método `toJson` na classe `Sale`.

```
package br.com.casadocodigo.loja.models;  
  
import javax.json.Json;  
import javax.json.JsonObjectBuilder;  
  
import br.com.casadocodigo.loja.models.Book;  
  
public class Sale {  
  
    private String title;  
    private Book book;  
  
    //outros métodos  
  
    public String toJson() {  
        JsonObjectBuilder sale = Json.createObjectBuilder();  
        sale.add("title", this.title)  
            .add("bookId", this.book.getId());  
    }  
}
```

```
        return sale.build().toString();
    }

}
```

Apenas usamos um pouco mais da nova especificação de geração de JSON, presente no Java EE 7.

15.4 OUTROS DETALHES DA ESPECIFICAÇÃO DE WEBSOCKETS

Nossa funcionalidade já está implementada, vamos apenas tentar deixar nosso código um pouco melhor. Por exemplo, caso um usuário feche o navegador, a sua conexão WebSocket será perdida, mas mesmo assim ainda mantemos a referência para ele na classe `ConnectedUsers`. O melhor é que retiremos esse cliente, para não termos objetos inválidos.

```
@ServerEndpoint("/channel/sales")
public class SalesEndpoint {

    //outros métodos

    @OnClose
    public void onClose(Session session, CloseReason closeReason){
        connectedUsers.remove(session);
        System.out.println(closeReason.getCloseCode());
    }
}
```

Continuando na linha da simplicidade, basta que anotemos o nosso método com `@OnClose`. Recebemos um objeto do tipo `Session` que, como já vimos, representa a conexão com o cliente. Além disso, ainda podemos receber um objeto do `CloseReason` que, como o próprio nome diz, retorna o motivo da quebra da conexão.

Para fechar, caso sua aplicação necessite receber mensagens do cliente, basta que seja criado um método anotado com `OnMessage` e, assim como fizemos para enviar, podemos receber uma `String` como argumento.

15.5 CONCLUSÃO

O uso de WebSockets já é uma realidade nas aplicações. Muitas das que usamos (como WhatsApp Web, Slack etc.) usam essa API para poder enviar e receber mensagens de um servidor. O exemplo mais comum encontrado pelo mundo é a criação de um chat, mas mostramos que a aplicação do conceito vale para outras situações também.

Caso você tenha a necessidade de usar vários WebSockets na sua aplicação, nada impede que seja criada várias classes anotadas com `@ServerEndpoint`, e que todas compartilhem do mesmo objeto do tipo `ConnectedUsers`. A única preocupação necessária é a de evoluir a classe `ConnectedUsers`, para que ela suporte clientes vindos de `endpoints` diferentes.

Você já está quase no fim do livro! No próximo capítulo, será comentado detalhes importantes que você deve levar em consideração quando for realizar o deploy da sua aplicação, além de algumas opiniões importantes sobre algumas partes que poderiam ser melhores dentro da especificação.

Minha dica é que você já finalize o livro, não deixe para amanhã o que você pode fazer **agora**.

CAPÍTULO 16

Últimas considerações técnicas

16.1 DEPLOY

Uma parte muito importante na vida de qualquer aplicação é a sua instalação no ambiente de produção, momento também conhecido como *deploy*. Para que o deploy seja o mais fácil possível, idealmente todas as configurações deveriam ficar dentro da própria aplicação, como fizemos no momento da configuração do tópico do JMS. Entretanto, as nossas configurações de acesso a banco de dados e envio de e-mail e segurança ainda ficaram no arquivo do próprio WildFly.

Dado esse cenário, não vamos ter muita escapatória. Será preciso pegar uma máquina limpa e instalar tudo o que for necessário para a nossa aplicação rodar.

Opções de servidores

Estamos vivendo a época dos *clouds*. Criar e destruir máquinas são operações que estão a um clique de um botão em uma interface web. Vamos tirar proveito disso e conhecer três boas opções do mercado.

Vamos começar pela **Amazon** e pela **DigitalOcean**. Ambas oferecem toda a infraestrutura necessária para você criar seus servidores e instalar tudo o que for necessário para que sua aplicação execute.

Como a responsabilidade de instalar e cuidar dos softwares necessários para a execução do sistema é do programador, esse tipo de *cloud* é conhecido como *IAAS*, acrônimo para *Infrastructure as a Service*. Este autor gosta muito da DigitalOcean, por conta da maior simplicidade.

Uma terceira opção, especificamente para o mundo Java EE, é o **OpenShift**. Além de fornecer a infraestrutura, eles também possuem painéis de configuração em que é possível adicionar servidores de aplicação, bancos de dados e mais uma outra grande variedade de softwares.

Nesse tipo de *cloud*, o desenvolvedor fica responsável apenas por enviar o código, todo o resto deveria ser de responsabilidade do serviço de cloud. O problema, como já comentamos, é que temos algumas configurações que ficam nos arquivos de configurações. Por exemplo, o OpenShift até resolve a configuração do banco de dados, mas ainda nos deixa a ver navios em relação ao e-mail e a parte de segurança.

Por conta desses problemas, este autor prefere ainda não usar o OpenShift como opção de servidor. De qualquer forma, esse de tipo de *cloud*, que oferece além da automatização da infraestrutura de máquinas, também a automatização da infraestrutura de software, é conhecido como *PAAS*, acrônimo para *Platform as a Service*. Outra opção nessa linha é o **CloudBees**.

Lembre-se de que realizar o deploy não muda muito em relação ao que você já vem fazendo localmente. Você precisará ter um banco de dados, um WildFly instalado na máquina de deploy e o `war` da sua aplicação, para que você possa adicionar na pasta `stadalone/deployments` do próprio Wildfly. Lembre-se apenas de criar o `DataSource` de produção com o mesmo nome que você criou o de desenvolvimento.

Dependências necessárias

Para gerar o arquivo `war` de deploy usando o Maven, é necessário que rodemos o comando `caminhoParaMaven/bin/mvn package`. O problema é que, quando executamos essa linha, recebemos os seguintes erros no console:

```
[ERROR] COMPILATION ERROR :
[INFO]
[ERROR] /Users/alberto/ambiente/desenvolvimento/jboss-developer-tools/casadocodigo/src/main/java/br/com/casadocodigo/models/SystemUser.java:[17,43] package org.hibernate.validator.constraints does not exist
[ERROR] /Users/alberto/ambiente/desenvolvimento/jboss-developer-tools/casadocodigo/src/main/java/br/com/casadocodigo/models/SystemUser.java:[18,43] package org.hibernate.validator.constraints does not exist
[ERROR] /Users/alberto/ambiente/desenvolvimento/jboss-developer-tools/casadocodigo/src/main/java/br/com/casadocodigo/models/ShoppingCart.java:[12,18] package javax.json does not exist
[ERROR] /Users/alberto/ambiente/desenvolvimento/jboss-developer-tools/casadocodigo/src/main/java/br/com/casadocodigo/models/ShoppingCart.java:[13,18] package javax.json does not exist
[ERROR] /Users/alberto/ambiente/desenvolvimento/jboss-developer-tools/casadocodigo/src/main/java/br/com/casadocodigo/models/Book.java:[14,36] package javax.validation.constraints does not exist
[ERROR] /Users/alberto/ambiente/desenvolvimento/jboss-developer-tools/casadocodigo/src/main/java/br/com/casadocodigo/models/Book.java:[15,36] package javax.validation.constraints does not exist
[ERROR] /Users/alberto/ambiente/desenvolvimento/jboss-developer-tools/casadocodigo/src/main/java/br/com/casadocodigo/models/Book.java:[16,36] package javax.validation.constraints does not exist
[ERROR] /Users/alberto/ambiente/desenvolvimento/jboss-developer-tools/casadocodigo/src/main/java/br/com/casadocodigo/models/Book.java:[17,36] package javax.validation.constraints does not exist
```

Fig. 16.1: Problema de compilação

Aconteceram muito mais erros do que o exibido, mas esse trecho já nos passa uma ideia. O Maven não está conseguindo encontrar as dependências necessárias para compilar nosso código.

Como adicionamos o WildFly no nosso projeto, dentro do Eclipse, esse problema ficou mascarado. Só que quando vamos compilar direto pelo terminal, é necessário ter todos os `jars` necessários para que tudo funcione. Para resolver essa questão, basta que as dependências necessárias sejam adicionadas no `pom.xml`.

```
<!-- javaee7 api -->
```

```
<dependency>
  <groupId>javax</groupId>
  <artifactId>javaee-api</artifactId>
  <version>7.0</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.picketbox</groupId>
  <artifactId>picketbox</artifactId>
```

```
<version>4.9.2.Final</version>
<scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>5.2.0.Final</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-jaxb-provider</artifactId>
  <version>3.0.11.Final</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>4.3.10.Final</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>4.3.10.Final</version>
  <scope>provided</scope>
</dependency>
```

Perceba que deixamos o escopo de todas essas novas dependências como `provided`, justamente para explicar ao Maven que elas são úteis na compilação, mas que não devem ser empacotadas no `war` final.

16.2 PROFILES

Não é anormal termos informações que variam entre os ambientes que executamos o nosso código: configurações de banco de dados, classes que precisam ser carregadas etc. O mundo Java EE ainda carece de uma especificação que adicione a parte de *environment* à nossa aplicação.

Os *alternatives* do CDI ainda são configurados via XML, o que nos deixa com pouca flexibilidade para indicar qual o ambiente que está sendo executado no momento. Até existe uma iniciativa da Apache com o projeto Tamaya (<http://tamaya.incubator.apache.org/>) , mas ainda estamos longe de ter todas as especificações ligadas a uma especificação de ambiente, como elas já estão ligadas ao CDI.

16.3 TESTES

Outro ponto que ainda deixa a desejar dentro da especificação Java EE é o apoio para o ciclo de testes da nossa aplicação. Talvez, não seja nem responsabilidade da especificação, mas como todos os componentes do nosso código são, de alguma forma, ligados ao Java EE, era mais do que justo que tivéssemos alguma ajuda nessa área.

Precisamos iniciar contextos do CDI, criar e destruir bases de dados, testar invocação de métodos nos beans do JSF ou recursos do JAX-RS, enfim, uma área que ainda pode ser bem explorada.

16.4 PROJETOS PARALELOS QUE PODEM NOS AJUDAR

Na tentativa de fechar alguns dos buracos que citamos neste capítulo, alguns projetos foram criados pela comunidade:

- **DeltaSpike:** um conjunto de bibliotecas que podem ajudar em várias partes do projeto. Por exemplo, eles têm uma implementação de perfis.
- **Arquillian:** a ideia é facilitar os testes integrados com o container. Por mais que tenha seu valor, ainda é muito complexo e, na opinião deste autor, mais atrapalha do que ajuda.

Esses dois projetos já estão bem famosos. O DeltaSpike realmente pode ter seu valor, já que ele se aventura em áreas que ainda não estão tão fortes na especificação. Como comentado, ele traz uma implementação para a parte de **profiles**, assim como tenta ajudar, por exemplo, na parte de testes.

16.5 CONCLUSÃO

É sempre importante ter um olhar crítico sobre todas as soluções que usamos. O conjunto de tecnologias providas pelo Java EE ainda carece de melhorias em alguns pontos. Mesmo assim, a versão 7 da especificação trouxe muitas facilidades, como vimos no decorrer do livro. Vale muito a pena utilizá-la, desde projetos simples até complexos, já que as APIs estão muito fáceis de serem aproveitadas.

Este foi o último capítulo técnico do livro! No próximo, aproveito para agradecer a você pela companhia e também deixo meus contatos, para que não percamos a amizade!

CAPÍTULO 17

Hora de praticar

Muito obrigado por ter ficado comigo durante toda a jornada do livro! É muito importante que você tente praticar tudo o que foi estudado.

Hoje em dia, no mundo Java, as duas melhores plataformas de desenvolvimento são o Java EE e o Spring. Seja pragmático, tente conhecer bem dos dois mundos e tenha as informações necessárias para tomar a melhor decisão no seu projeto.

Apenas para dar um panorama, a disputa está muito acirrada. Enquanto o Spring tem componentes mais bem conectados e traz coisas como Profile e facilidades para testes; o Java EE 7 fornece APIs para trabalhos complexos, mas que são expostas de maneira muito simples para o programador.

Tudo dependerá das necessidades da aplicação e das restrições, que muitas vezes são impostas pelos clientes. Não seja apaixonado, crie uma tabela de decisão e veja quais são os pontos necessários, providos pelos dois lados, que podem lhe auxiliar para seguir com o melhor caminho técnico.

Para concluir, é muito importante que você continue estudando! Faça mais projetos, leia as especificações, ajude as pessoas nos fóruns e tudo mais. Toda fonte de aprendizado vai ser importante no momento em que a situação complicada chegar.

17.1 ESTUDAMOS MUITOS ASSUNTOS!

Conseguimos um feito quase raro, usamos diversas especificações sem forçar a barra em nenhum lugar. Tudo que foi utilizado teve um motivo e poderia ser aplicado caso você estivesse construindo qualquer outra aplicação.

Por sinal, apenas reforçando, este autor recomenda fortemente que você refaça a aplicação inteira, ou que pegue uma outra aplicação de que você gosta e tente construir uma similar.

Que tal tentar implementar várias funcionalidades do facebook? Aqueles atualizações dos comentários dos posts parecem ser um ótimo caso para a utilização de WebSockets. Existem várias aplicações que podem servir de inspiração!

17.2 MANTENHA CONTATO

Estou completamente disponível para ajudá-lo onde for possível. Você pode me achar das seguintes formas:

- Por e-mail, basta enviar uma mensagem para livrojavaee@gmail.com;
- Caso use o Twitter, é só acessar https://twitter.com/alberto_souza;
- Ainda há a possibilidade de acompanhar meus projetos pelo GitHub, basta acessar <https://github.com/asouza>.
- Caso queira postar uma dúvida e ainda ajudar outras pessoas, use o fórum criado especialmente para o livro, disponível em <https://groups.google.com/forum/#!forum/livro-javaee>.

Por último, todo e qualquer feedback sobre o livro é bem-vindo. Use o fórum ou mande diretamente para meu e-mail. Escrevê-lo foi um grande desafio, espero que ele tenha sido de alguma valia para você.