



K9

TREINAMENTOS

# Desenvolvimento Web Avançado com JSF2, EJB3.1 e CDI

# Desenvolvimento Web Avançado com JSF2, EJB3.1 e CDI

29 de abril de 2011



# Sumário

<b>1</b>	<b>Introdução a EJB</b>	<b>1</b>
1.1	Por que utilizar EJB? . . . . .	1
1.2	EJB Container . . . . .	2
1.3	Exercícios . . . . .	2
<b>2</b>	<b>Stateless Session Beans</b>	<b>3</b>
2.1	Session Beans . . . . .	3
2.2	Caracterizando os SLSBs . . . . .	3
2.2.1	Serviço de Câmbio . . . . .	3
2.2.2	Dicionário . . . . .	4
2.2.3	Consulta de CEP . . . . .	4
2.3	SLSB - EJB 3.0 . . . . .	4
2.4	SLSB - EJB 3.1 . . . . .	5
2.5	Cliente Java Web Local - EJB 3.0 . . . . .	6
2.6	Exercícios . . . . .	7
2.7	Cliente Java Web Local - EJB 3.1 . . . . .	18
2.8	Exercícios . . . . .	18
2.9	Cliente Java SE Remoto . . . . .	24
2.10	Exercícios . . . . .	25
2.11	Ciclo de Vida . . . . .	29
2.11.1	Estados . . . . .	30
2.11.2	NÃO EXISTE -> PRONTO . . . . .	30
2.11.3	PRONTO -> PRONTO . . . . .	30
2.11.4	PRONTO -> NÃO EXISTE . . . . .	30
2.11.5	Escalabilidade e Pool . . . . .	31
2.11.6	Callbacks . . . . .	31
2.12	Exercícios . . . . .	32
<b>3</b>	<b>Stateful Session Beans</b>	<b>33</b>
3.1	Caracterizando os SFSBs . . . . .	33
3.1.1	Carrinho de Compras . . . . .	33
3.1.2	Prova Digital . . . . .	34
3.1.3	TrackList . . . . .	34
3.2	SFSB - EJB 3.0 . . . . .	35
3.3	SFSB - EJB 3.1 . . . . .	36
3.4	Exercícios . . . . .	36

3.5	Ciclo de Vida . . . . .	42
3.5.1	Estados . . . . .	42
3.5.2	NÃO EXISTE -> PRONTO . . . . .	43
3.5.3	PRONTO -> PASSIVADO . . . . .	43
3.5.4	PASSIVADA -> PRONTO . . . . .	43
3.5.5	PRONTO -> NÃO EXISTE . . . . .	43
3.5.6	PASSIVADO -> PRONTO -> NÃO EXISTE . . . . .	44
3.5.7	Callbacks . . . . .	44
3.6	Exercícios . . . . .	46
<b>4</b>	<b>Singleton Session Beans</b>	<b>47</b>
4.1	Caracterizando os Singleton Session Beans . . . . .	47
4.1.1	Número de usuários conectados . . . . .	47
4.1.2	Sistema de chat . . . . .	47
4.1.3	Trânsito Colaborativo . . . . .	48
4.2	Implementação . . . . .	48
4.2.1	Singleton Session Beans Locais . . . . .	49
4.3	Exercícios . . . . .	50
4.4	Ciclo de Vida . . . . .	54
4.4.1	Estados . . . . .	54
4.4.2	NÃO EXISTE -> PRONTO . . . . .	54
4.4.3	PRONTO -> NÃO EXISTE . . . . .	55
4.4.4	Callbacks . . . . .	56
<b>5</b>	<b>Persistência</b>	<b>57</b>
5.1	Data Sources . . . . .	57
5.2	Exercícios . . . . .	57
5.3	persistence.xml . . . . .	61
5.4	Entity Beans . . . . .	61
5.5	Entity Classes e Mapeamento . . . . .	62
5.6	Exercícios . . . . .	63
5.7	Entity Managers . . . . .	66
5.7.1	Obtendo Entity Managers . . . . .	66
5.8	Entity Manager Factories . . . . .	67
5.8.1	Obtendo Entity Manager Factories . . . . .	67
5.9	Exercícios . . . . .	68
<b>6</b>	<b>Transações</b>	<b>71</b>
6.1	ACID . . . . .	71
6.2	Transação Local ou Distribuída . . . . .	71
6.3	JTA e JTS . . . . .	72
6.4	Container Managed Transactions - CMT . . . . .	72
6.4.1	Atributo Transacional . . . . .	72
6.4.2	Rollback com SessionContext . . . . .	73
6.4.3	Rollback com Exceptions . . . . .	74
6.5	Bean Managed Transactions - BMT . . . . .	74

6.6 Exercícios . . . . .	75
<b>7 Segurança</b>	<b>81</b>
7.1 Realms . . . . .	81
7.2 Exercícios . . . . .	81
7.3 Autenticação - Aplicações Web . . . . .	84
7.4 Exercícios . . . . .	85
7.5 Autorização - Aplicações EJB . . . . .	89
7.5.1 @RolesAllowed . . . . .	89
7.5.2 @PermitAll . . . . .	90
7.5.3 @DenyAll . . . . .	90
7.5.4 @RunAs . . . . .	90
7.6 Exercícios . . . . .	91
<b>8 Interceptadores</b>	<b>93</b>
8.1 Interceptor Methods . . . . .	93
8.2 Internal Interceptors . . . . .	94
8.3 External Interceptors . . . . .	94
8.3.1 Method-Level Interceptors . . . . .	95
8.3.2 Class-Level Interceptors . . . . .	95
8.3.3 Default Interceptors . . . . .	96
8.4 Excluindo Interceptadores . . . . .	96
8.5 Invocation Context . . . . .	97
8.6 Ordem dos Interceptadores . . . . .	97
8.7 Exercícios . . . . .	97
<b>9 Scheduling</b>	<b>107</b>
9.1 Timers . . . . .	107
9.2 Métodos de Timeout . . . . .	108
9.3 Timers Automáticos . . . . .	108
9.4 Exercícios . . . . .	108
<b>10 Contexts and Dependency Injection - CDI</b>	<b>117</b>
10.1 Managed Beans . . . . .	117
10.2 Producer Methods and Fields . . . . .	118
10.3 EL Names . . . . .	118
10.4 beans.xml . . . . .	119
10.5 Exercícios . . . . .	119
10.6 Escopos e Contextos . . . . .	125
10.7 Injection Points . . . . .	126
10.7.1 Bean Constructors . . . . .	126
10.7.2 Field . . . . .	126
10.7.3 Initializer methods . . . . .	127
10.8 Exercícios . . . . .	127
<b>11 Projeto</b>	<b>129</b>
11.1 Exercícios . . . . .	129



# Capítulo 1

## Introdução a EJB

### 1.1 Por que utilizar EJB?

Muitos sistemas corporativos são construídos seguindo a arquitetura definida pelo padrão Enterprise JavaBeans (EJB). Ao utilizar essa arquitetura, as aplicações ganham certos benefícios.

**Transações:** A arquitetura EJB define um suporte sofisticado para utilização de transações. Esse suporte é integrado com a Java Transaction API (JTA) e oferece inclusive a possibilidade de realizar transações distribuídas.

**Segurança:** Suporte para realizar autenticação e autorização de maneira transparente. Os desenvolvedores das aplicações não precisam implementar a lógica de segurança pois ela faz parte da arquitetura EJB.

**Remotabilidade:** Aplicações EJB podem ser acessadas remotamente através de diversos protocolos de comunicação. Consequentemente, é possível desenvolver aplicações clientes de diversos tipos. Em particular, aplicações EJB podem ser expostas como Web Services.

**Multithreading e Concorrência:** A arquitetura EJB permite que os sistemas corporativos possam ser acessados por múltiplos usuários simultaneamente de maneira controlada para evitar problemas de concorrência.

**Persistência:** Facilidades para utilizar os serviços dos provedores de persistência que seguem a especificação JPA.

**Gerenciamento de Objetos:** Mecanismos de injecção de dependências e controle de ciclo de vida são oferecidos aos objetos que formam uma aplicação EJB. O mecanismo de controle de ciclo de vida pode garantir acesso a uma quantidade variável de usuários.

**Integração:** A arquitetura EJB é fortemente integrada com os componentes da plataforma Java EE. Podemos, por exemplo, facilmente integrar uma aplicação JSF com uma aplicação EJB.

## 1.2 EJB Container

As aplicações EJB executam dentro de um EJB Container pois ele é o responsável pela implementação de vários recursos oferecidos a essas aplicações. Há diversas implementações de EJB Container disponíveis no mercado que podemos utilizar. Neste curso, utilizaremos a implementação do Glassfish V3.

A seguir, vamos instalar e configurar o Glassfish V3.

## 1.3 Exercícios

1. Na Área de Trabalho, entre na pasta **K19-Arquivos** e copie **glassfish-3.0.1-with-hibernate.zip** para o seu Desktop. Descompacte este arquivo na própria Área de Trabalho.
2. Ainda na Área de Trabalho, entre na pasta **glassfishv3/glassfish/bin** e execute o script **startserv** para executar o glassfish.
3. Verifique se o glassfish está executando através de um navegador acessando a url:  
**<http://localhost:8080>**.
4. Pare o glassfish executando o script **stopserv** que está na mesma pasta do script **startserv**.
5. No eclipse, abra a view **servers** e clique com o botão direito no corpo dela. Escolha a opção **new** e configure o glassfish.
6. Execute o glassfish pela view **servers** e verifique se ele está funcionando acessando através de um navegador a url:  
**<http://localhost:8080>**.
7. Pare o glassfish pela view **servers**.

# Capítulo 2

## Stateless Session Beans

### 2.1 Session Beans

Um sistema corporativo é composto por muitos processos ou tarefas. Por exemplo, um sistema bancário possui processos específicos para realizar transferências, depósitos, saques, empréstimos, cobranças, entre outros. Esses procedimentos são chamados de regras de negócio. Cada aplicação possui as suas próprias regras de negócio já que elas são consequência imediata do contexto da aplicação.

Utilizando a arquitetura EJB, as regras de negócio são implementadas em componentes específicos que são chamados de **Session Beans**. O EJB Container administra esses componentes oferecendo diversos recursos a eles.

### 2.2 Caracterizando os SLSBs

Stateless Session Bean é o primeiro tipo de Session Bean. Muitas vezes, utilizaremos a sigla SLSB para fazer referência a esse tipo de componente. A ideia fundamental por trás dos SLSBs é a não necessidade de manter estado entre as execuções das regras de negócio que eles implementam.

#### 2.2.1 Serviço de Câmbio

Para exemplificar, suponha uma regra de negócio para converter valores monetários entre moedas diferentes. Por exemplo, converter 100 reais para o valor correspondente em dólar americano. Poderíamos, implementar essa regra de negócio em um único método.

```
1 public double converte(double valor, String moeda1, String moeda2) {  
2     // lógica da conversão monetária  
3 }
```

Devemos perceber que uma execução do método CONVERTE() não depende das execuções anteriores. Em outras palavras, o método CONVERTE() não precisa manter estado entre as chamadas.

## 2.2.2 Dicionário

Outro exemplo, suponha a implementação de um dicionário de português. Dado uma palavra, o dicionário deve devolver a definição dela. Podemos criar um método para implementar essa regra de negócio.

```
1 public String getDefinicao(String palavra) {  
2     // lógica do dicionario  
3 }
```

Novamente, perceba que as chamadas ao método **getDefinicao()** são totalmente independentes. Dessa forma, não é necessário guardar informações referentes às chamadas anteriores.

## 2.2.3 Consulta de CEP

Mais um exemplo, dado uma localidade podemos consultar o CEP correspondente. Podemos criar um método para implementar essa regra de negócio.

```
1 public String consultaCEP(String estado, String cidade, String logradouro, int numero) ←  
2 {  
3     // lógica da consulta do CEP  
4 }
```

Como cada consulta de CEP independe das consultas anteriores, não é necessário manter dados entre uma consulta e outra. Em outras palavras, não é necessário manter estado.

## 2.3 SLSB - EJB 3.0

O primeiro passo para implementar um SLSB é definir a sua interface de utilização através de uma interface Java. Por exemplo, suponha um SLSB que implementa algumas operações matemáticas básicas. Uma possível interface de utilização para esse SLSB seria:

```
1 public interface Calculadora {  
2     double soma(double a, double b);  
3     double subtrai(double a, double b);  
4     double multiplica(double a, double b);  
5     double divide(double a, double b);  
6 }
```

Após definir a interface de utilização, o segundo passo seria implementar as operações do SLSB através de uma classe Java.

## Stateless Session Beans

---

```
1 public class CalculadoraBean implements Calculadora {  
2  
3     public double soma(double a, double b){  
4         return a + b;  
5     }  
6  
7     public double subtrai(double a, double b) {  
8         return a - b;  
9     }  
10  
11    public double multiplica(double a, double b) {  
12        return a * b;  
13    }  
14  
15    public double divide(double a, double b) {  
16        return a / b;  
17    }  
18}
```

O terceiro passo é especificar o tipo de Session Bean que queremos utilizar. No caso da calculadora, o tipo seria SLSB. Essa definição é realizada através da anotação **@Stateless**.

```
1 @Stateless  
2 public class CalculadoraBean implements Calculadora {  
3     ...  
4 }
```

Por fim, é necessário definir se o SLSB poderá ser acessado remotamente ou apenas localmente. Quando o acesso a um SLSB é local, ele só pode ser acessado por aplicações que estejam no mesmo servidor de aplicação que ele. Caso contrário, quando o acesso a um SLSB é remoto, ele pode ser acessado tanto por aplicações que estejam no mesmo servidor de aplicação quanto aplicações que não estejam.

A definição do tipo de acesso é realizada através das anotações: **@Local** e **@Remote**.

```
1 @Stateless  
2 @Remote(Calculadora.class)  
3 public class CalculadoraBean implements Calculadora {  
4     ...  
5 }
```

```
1 @Stateless  
2 @Local(Calculadora.class)  
3 public class CalculadoraBean implements Calculadora {  
4     ...  
5 }
```

## 2.4 SLSB - EJB 3.1

Na versão 3.1, quando o acesso a um SLSB é local, não é mais necessário definir uma interface Java nem utilizar a anotação **@LOCAL**. Então, bastaria implementar uma classe Java com a anotação **@STATELESS**.

```

1  @Stateless
2  public class CalculadoraBean {
3
4      public double soma(double a, double b) {
5          return a + b;
6      }
7
8      public double subtrai(double a, double b) {
9          return a - b;
10     }
11
12     public double multiplica(double a, double b) {
13         return a * b;
14     }
15
16     public double divide(double a, double b) {
17         return a / b;
18     }
19 }
```

## 2.5 Cliente Java Web Local - EJB 3.0

Como vimos os session beans são utilizados para implementar as regras negócios das nossas aplicações. Em particular, os stateless session beans são utilizados para implementar as regras de negócio que não necessitam manter estado. Contudo, além das regras de negócio, devemos nos preocupar com a interface dos usuários. Hoje em dia, na maioria dos casos, essa interface é web.

A seguir, vamos implementar um sistema que realiza operações matemáticas básicas para exemplificar a utilização da arquitetura EJB em conjunto com as tecnologias Java para desenvolvimento de interfaces web. Suponha que todo o sistema (session beans e a camada web) esteja no mesmo servidor de aplicação.

A seguinte interface Java será utilizada para definir as operações de um SLSB.

```

1  public interface Calculadora {
2      double soma(double a, double b);
3  }
```

Depois de definir a interface, devemos implementar as operações do SLSB através de uma classe Java com as anotações apropriadas.

```

1  @Stateless
2  @Local(Calculadora.class)
3  public class CalculadoraBean implements Calculadora {
4      public double soma(double a, double b) {
5          return a + b;
6      }
7  }
```

Perceba que o acesso local foi definido para esse SLSB pois ele será acessado por uma camada web no mesmo servidor de aplicação. Nesse momento, o SLSB está pronto.

O próximo passo é implementar a camada web. A interface que define as operações do SLSB precisa estar no classpath da camada web. Já a classe que implementa as operações não

## Stateless Session Beans

---

precisa estar no classpath da camada web.

Suponha que a camada web da nossa aplicação utiliza apenas Servlets. Podemos injetar, através da anotação **@EJB**, o SLSB em uma Servlet.

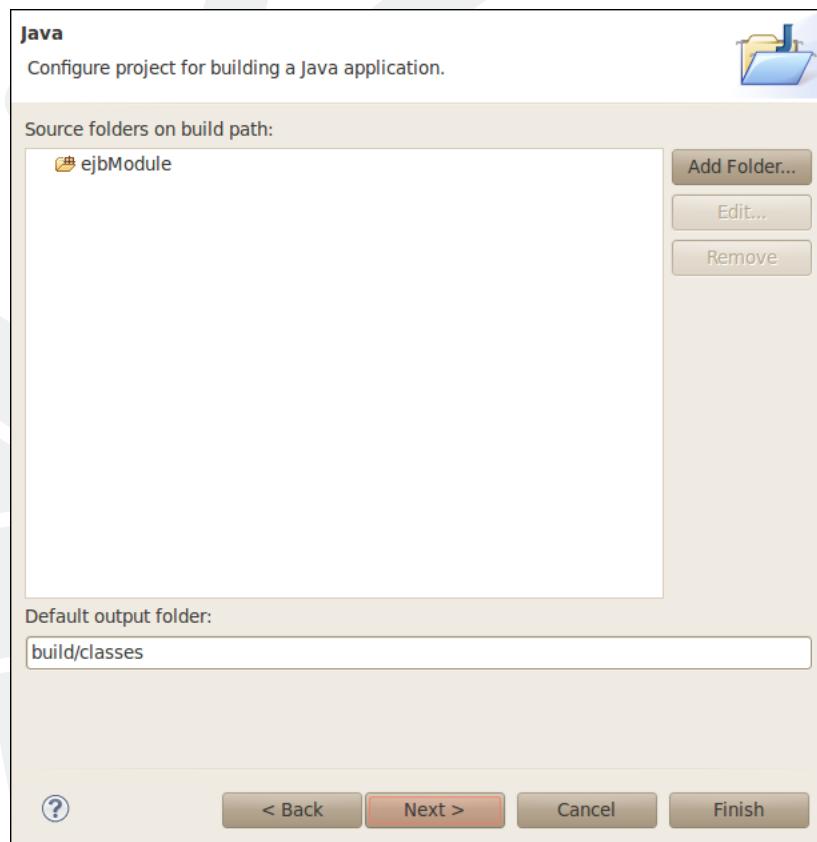
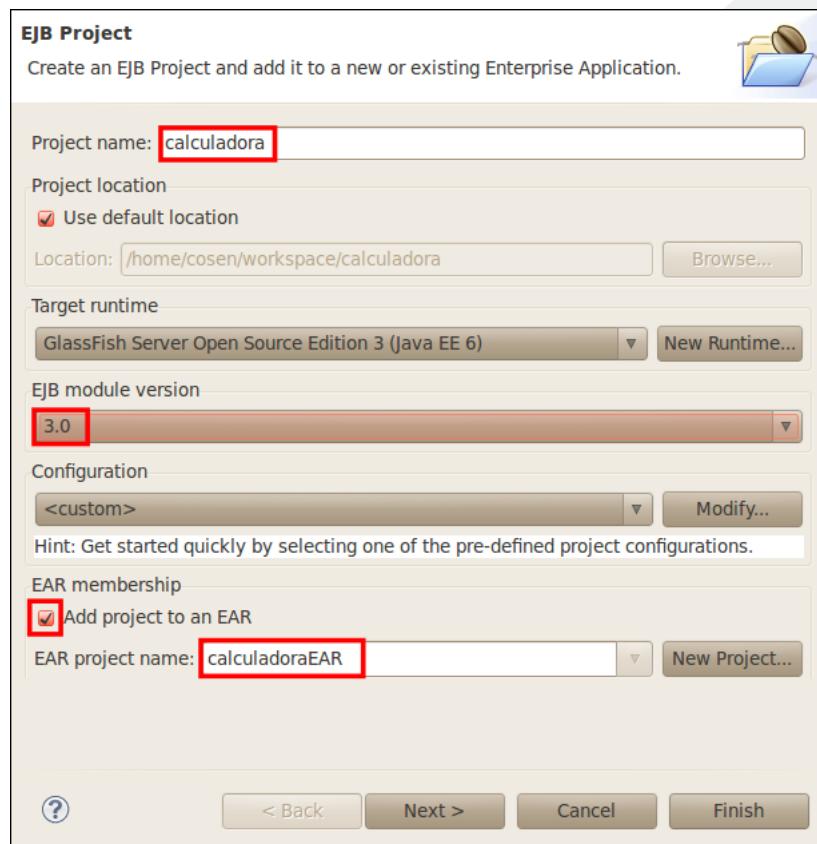
```
1 @WebServlet("/soma")
2 public class SomaServlet extends HttpServlet {
3     @EJB
4     private Calculadora calculadora;
5
6     protected void service(HttpServletRequest req, HttpServletResponse res) {
7         double a = Double.parseDouble(req.getParameter("a"));
8         double b = Double.parseDouble(req.getParameter("b"));
9
10        double resultado = this.calculadora.soma(a, b);
11
12        PrintWriter out = response.getWriter();
13        out.println("<html><body><p>");
14        out.println("Soma: " + resultado);
15        out.println("</p></body></html>");
16    }
17 }
```

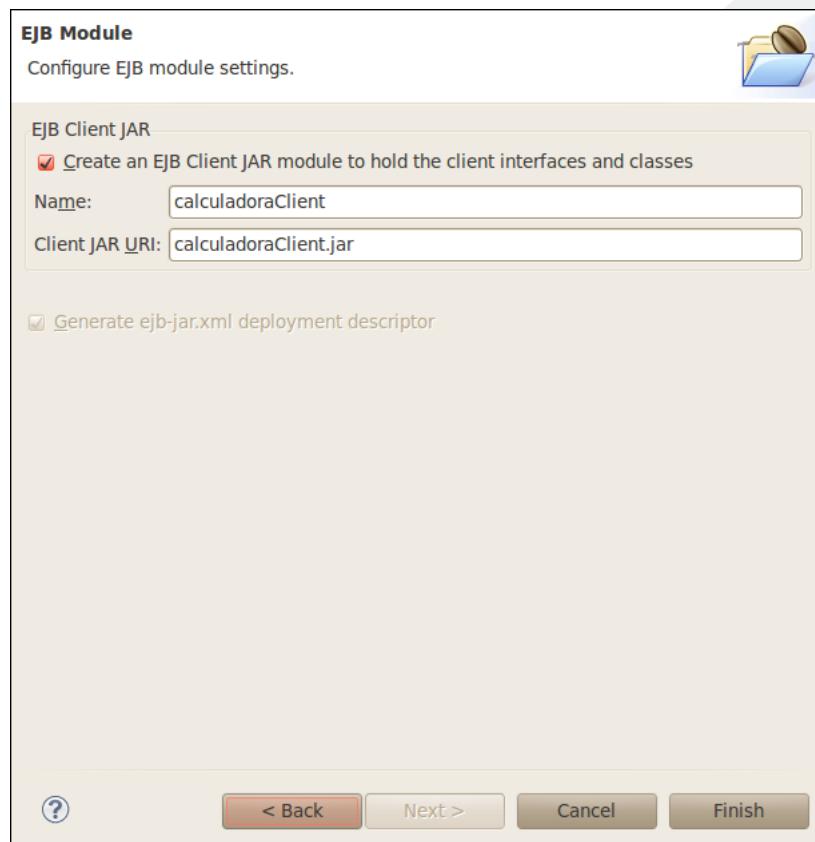
Agora, suponha que a camada web utilize JSF. Podemos injetar o SLSB em um Managed Bean também através da anotação **@EJB**.

```
1 @ManagedBean
2 public class CalculadoraMB {
3     @EJB
4     private Calculadora calculadora;
5
6     private double a;
7
8     private double b;
9
10    private double resultado;
11
12    public void soma() {
13        this.resultado = this.calculadora.soma(a,b);
14    }
15
16    // GETTERS AND SETTERSS
17 }
```

## 2.6 Exercícios

1. Crie um EJB project no eclipse. Você pode digitar “CTRL+3” em seguida “new EJB project” e “ENTER”. Depois, siga exatamente as imagens abaixo.





**OBS:** Três projetos serão criados:

**calculadora:** As classes que implementam os SLSB devem ser colocadas nesse projeto.

**calculadoraClient:** As interfaces que definem as operações dos SLSB devem ser colocadas nesse projeto.

**calculadoraEAR:** Esse projeto empacota todos os módulos do nosso sistema.

2. Crie um Dynamic Web Project no eclipse para implementar a camada web. Você pode digitar “CTRL+3” em seguida “new Dynamic Web Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.

**Dynamic Web Project**

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: **calculadoraWeb**

Project location  
 Use default location

Location: /home/cosen/workspace/calculadoraWeb

Target runtime  
GlassFish Server Open Source Edition 3 (Java EE 6)

Dynamic web module version  
3.0

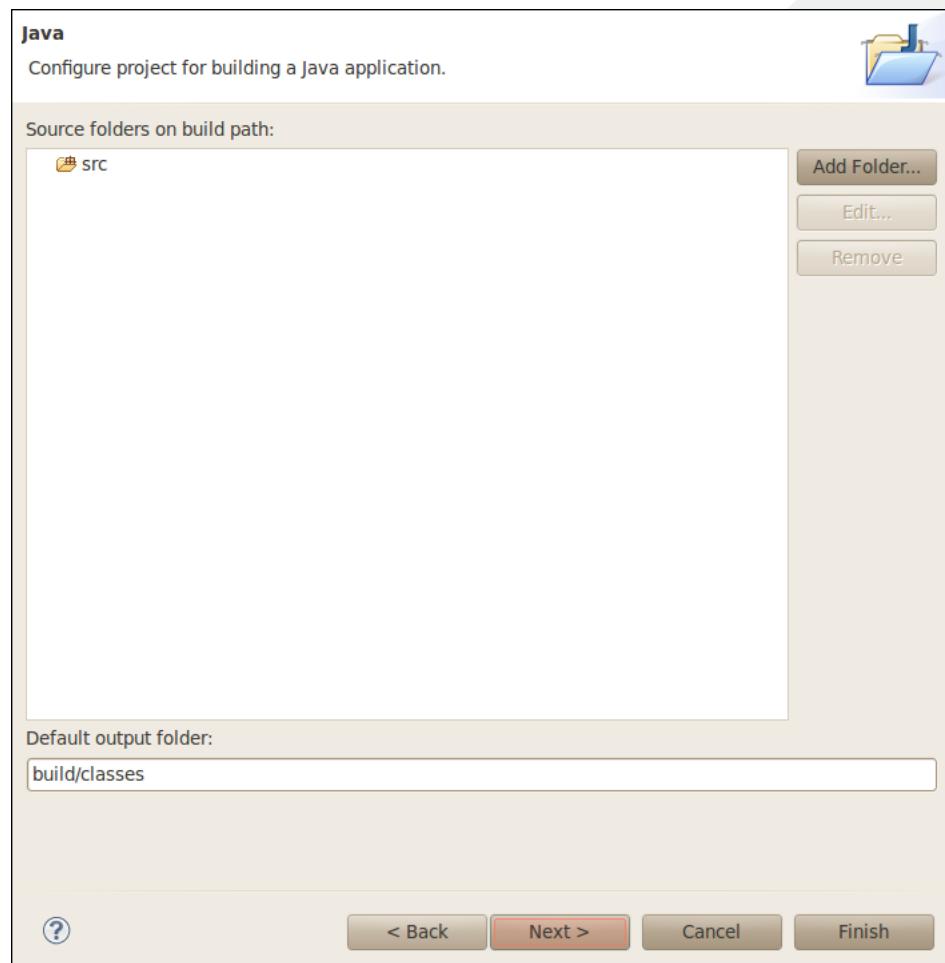
Configuration  
**JavaServer Faces v2.0 Project**   
Configures a Dynamic Web application to use JSF v2.0

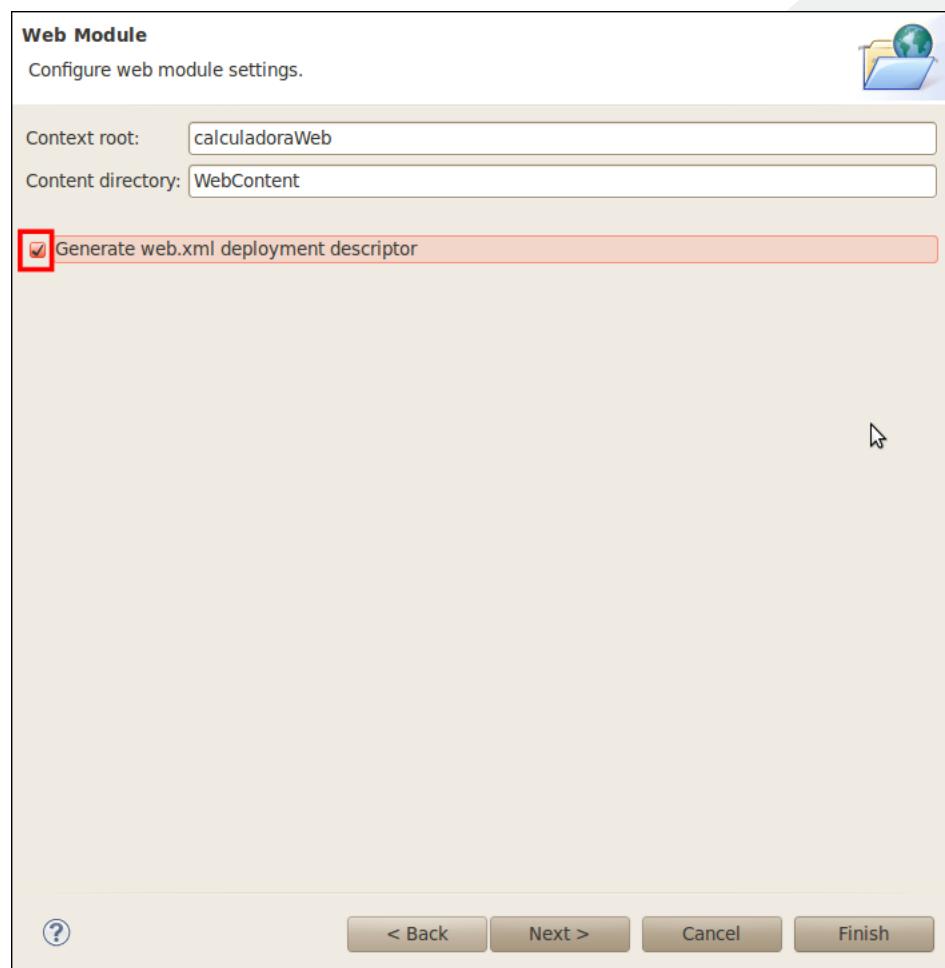
EAR membership  
 Add project to an EAR

EAR project name: **calculadoraEAR**

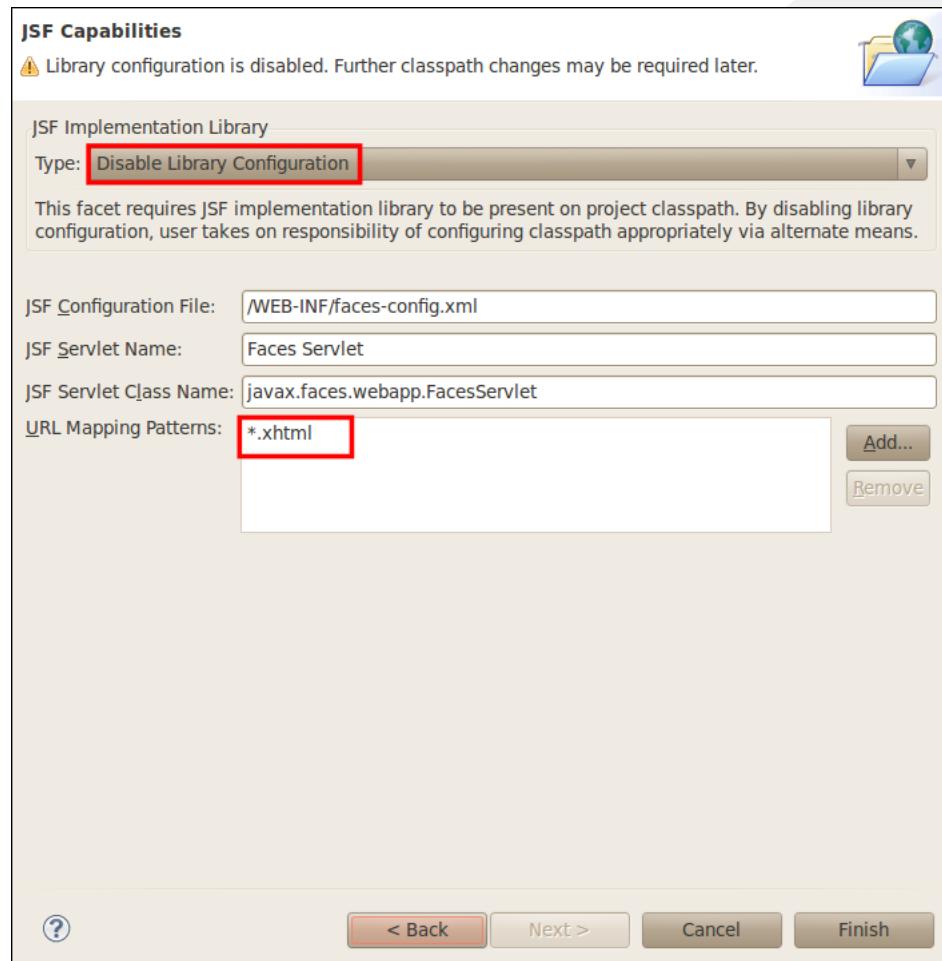
## *Stateless Session Beans*

---



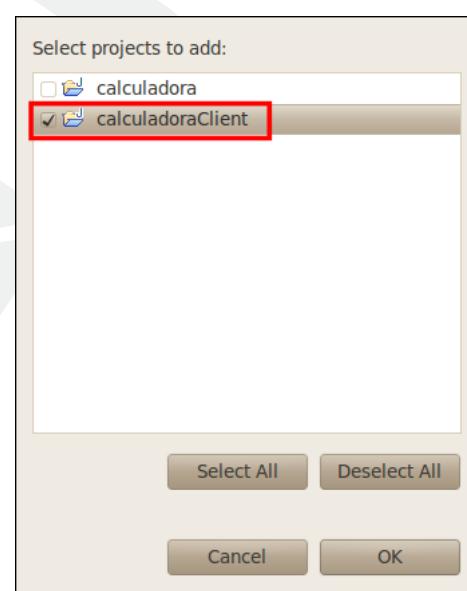
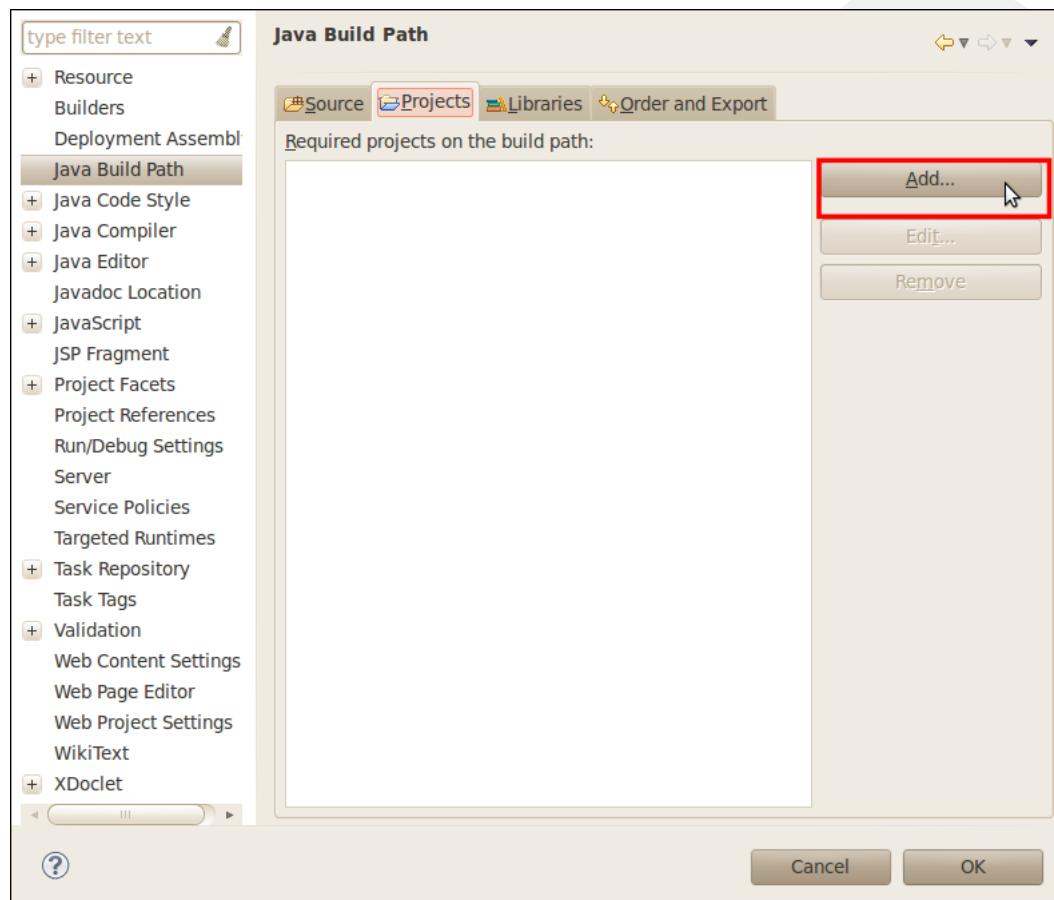


## Stateless Session Beans

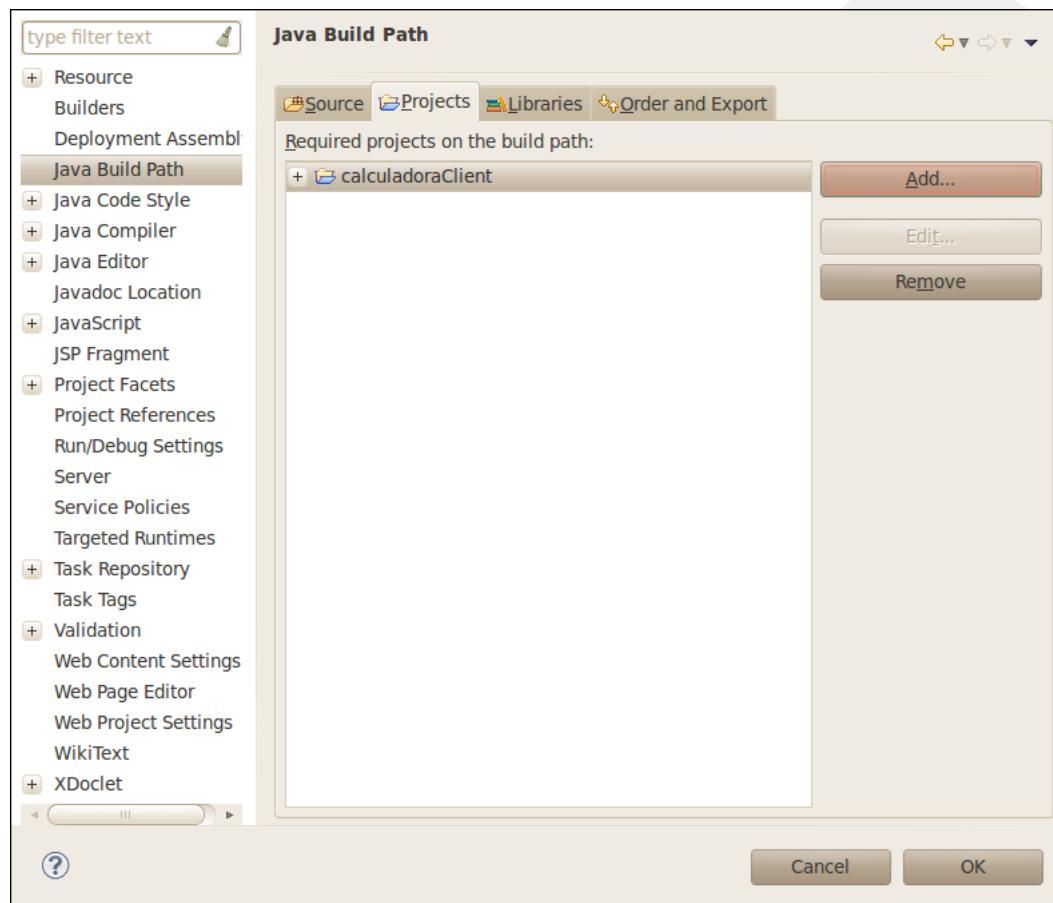


**OBS:** O módulo web já está vinculado ao projeto **calculadoraEAR**.

3. Adicione o projeto **calculadoraClient** como dependência do projeto **calculadoraWeb**. Abra as propriedades do projeto **calculadoraWeb**. Você pode selecionar o projeto **calculadoraWeb** e digitar “ALT+ENTER”. Depois, siga as imagens abaixo.



## Stateless Session Beans



4. No projeto **calculadoraClient**, adicione um pacote chamado **sessionbeans** e acrescente nele uma interface Java chamada **Calculadora**.

```
1 package sessionbeans;
2
3 public interface Calculadora {
4     double soma(double a, double b);
5 }
```

5. No projeto **calculadora**, adicione um pacote chamado **sessionbeans** e acrescente nele uma classe Java chamada **CalculadoraBean**.

```
1 package sessionbeans;
2
3 @Stateless
4 @Local(Calculadora.class)
5 public class CalculadoraBean implements Calculadora {
6     public double soma(double a, double b) {
7         return a + b;
8     }
9 }
```

6. No projeto **calculadoraWeb**, adicione um pacote chamado **managedbeans** acrescenta nele uma classe Java chamada **CalculadoraMB**.

```

1 package managedbeans;
2
3 import javax.ejb.EJB;
4 import javax.faces.bean.ManagedBean;
5
6 import sessionbeans.Calculadora;
7
8 @ManagedBean
9 public class CalculadoraMB {
10
11     @EJB
12     private Calculadora calculadora;
13
14     private double a;
15
16     private double b;
17
18     private double resultado;
19
20     public void soma() {
21         this.resultado = this.calculadora.soma(a, b);
22     }
23
24     // GETTERS AND SETTERS
25 }
```

7. Crie uma simples tela na aplicação web para utilizar o Manager Bean. Adicione o arquivo **soma.xhtml** na pasta **WebContent** do projeto **calculadoraWeb** com o seguinte conteúdo.

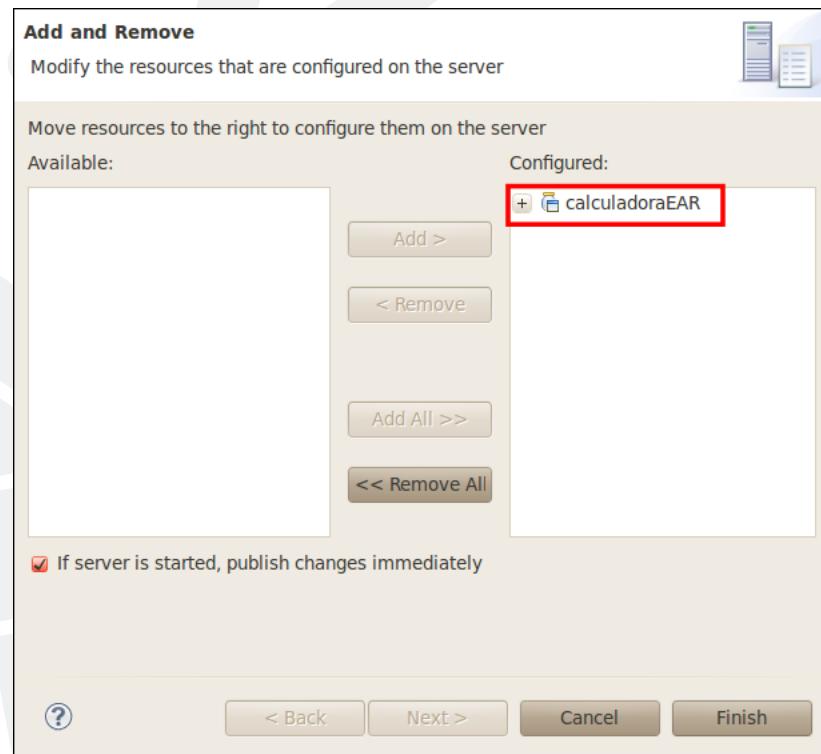
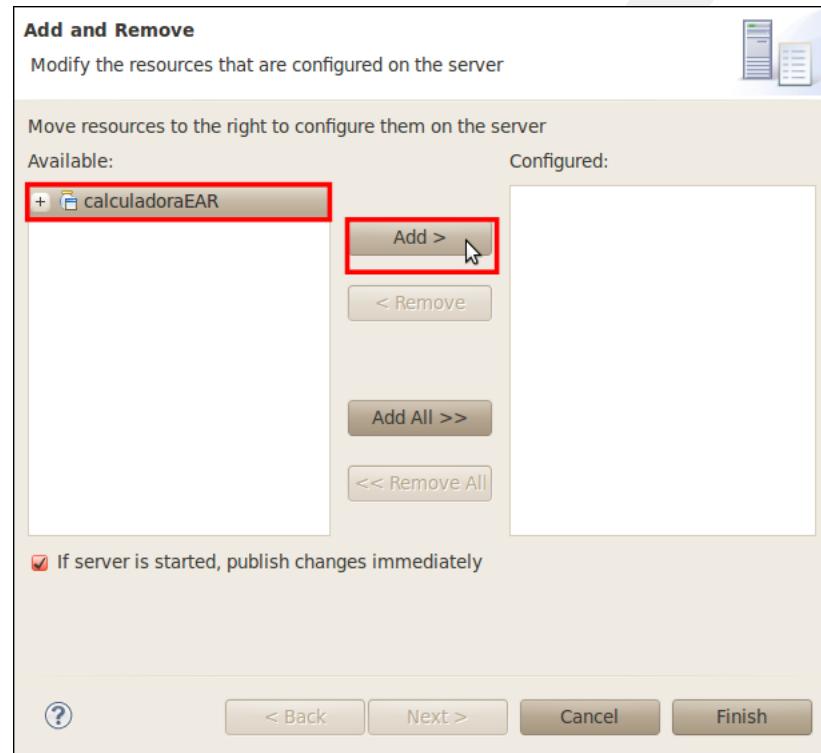
```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5      xmlns:h="http://java.sun.com/jsf/html">
6
7 <h:head>
8     <title>Calculadora - Soma</title>
9 </h:head>
10
11 <h:body>
12     <h:form>
13         <h:outputLabel value="Valor A: "/>
14         <h:inputText value="#{calculadoraMB.a}" ></h:inputText>
15
16         <h:outputLabel value="Valor B: "/>
17         <h:inputText value="#{calculadoraMB.b}" ></h:inputText>
18
19         <h:commandButton action="#{calculadoraMB.soma}" value="soma"/>
20
21         <h:outputLabel value="Resultado: "/>
22         <h:outputText value="#{calculadoraMB.resultado}" />
23     </h:form>
24 </h:body>
25 </html>
```

8. Adicione o projeto **calculadoraEAR** no glassfish. Clique com o botão direito no glas-

## Stateless Session Beans

sfish da view Servers e escolha a opção “Add and Remove”. Depois, siga as imagens abaixo.



9. Acesse a url <http://localhost:8080/calculadoraWeb/soma.xhtml> e teste o funcionamento da aplicação.

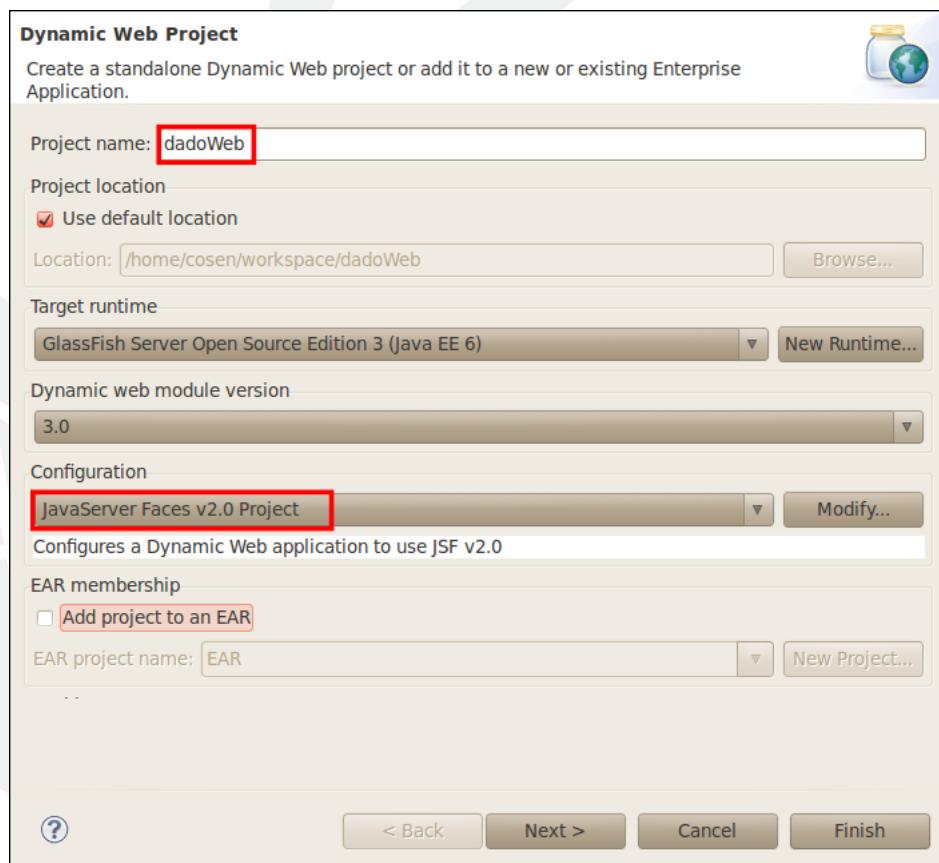
## 2.7 Cliente Java Web Local - EJB 3.1

Como dito anteriormente, na versão 3.1, quando o acesso a um SLSB é local, não é mais necessário definir uma interface Java nem utilizar a anotação @LOCAL.

Além disso, as regras de empacotamento foram simplificadas. Os Session Beans podem ser empacotados no módulo web. Isso simplifica bastante o funcionamento das IDEs como o eclipse. Perceberemos essa diferença o exercício seguinte.

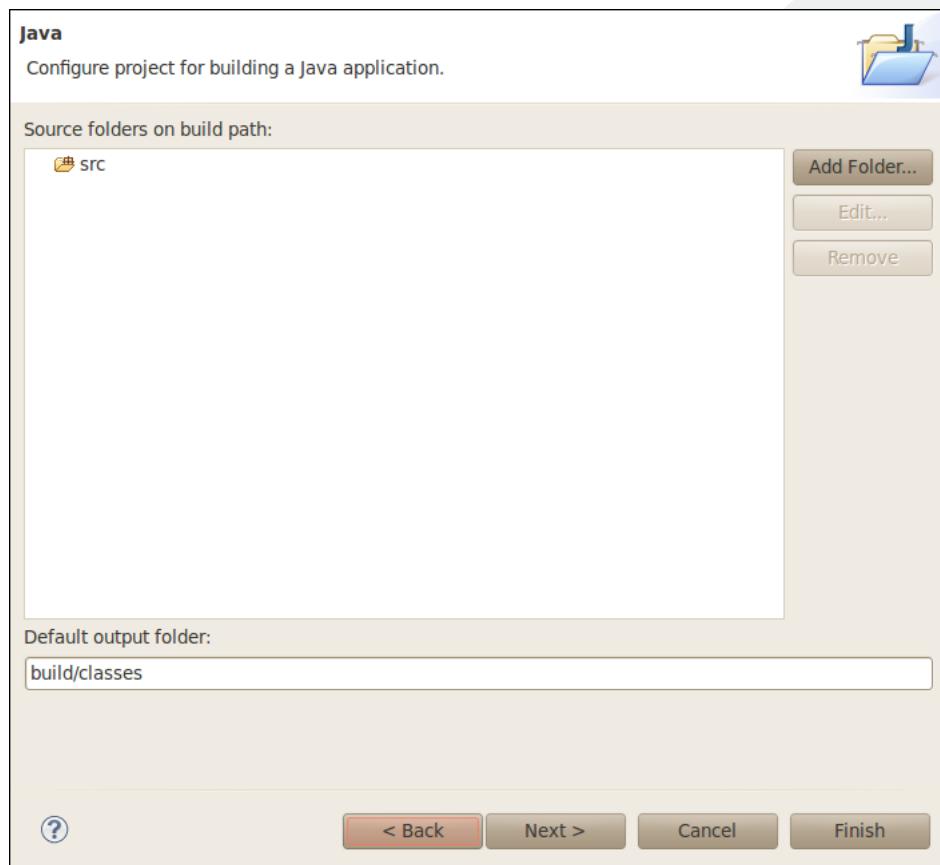
## 2.8 Exercícios

10. Crie um Dynamic Web Project no eclipse para implementar a camada web. Você pode digitar “CTRL+3” em seguida “new Dynamic Web Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.



## Stateless Session Beans

---



**Web Module**  
Configure web module settings.

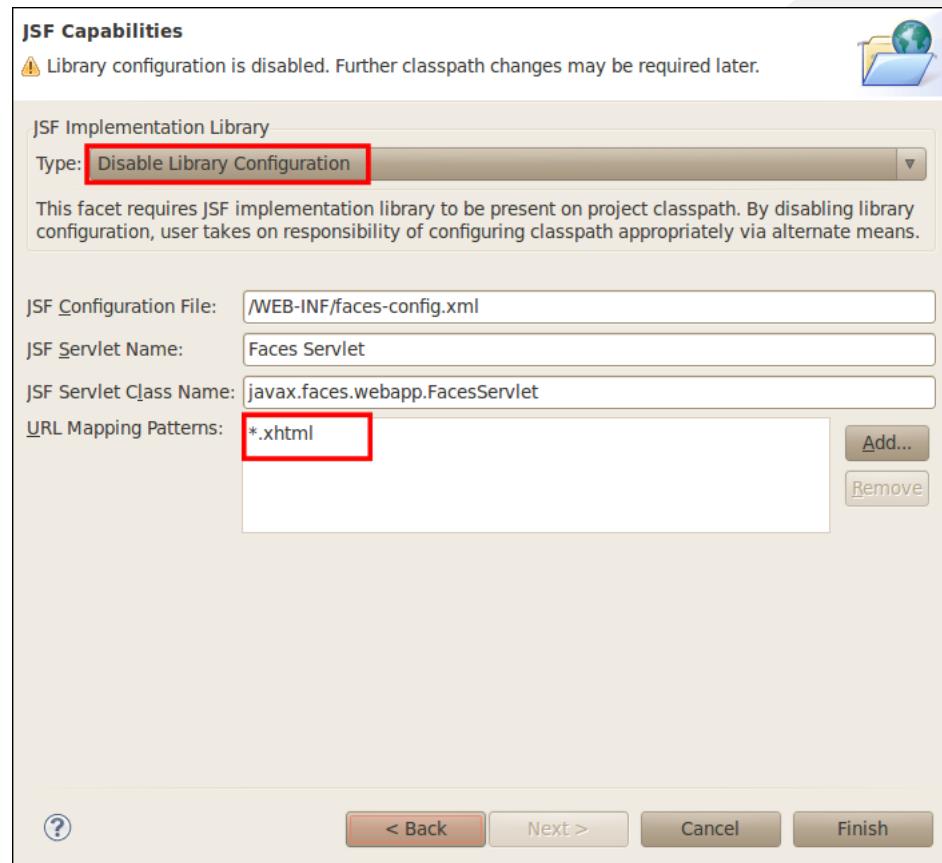
Context root:  Content directory:

Generate web.xml deployment descriptor

?

< Back    Next >    Cancel    Finish

## Stateless Session Beans



11. No projeto **dadoWeb**, adicione um pacote chamado **sessionbeans** e acrescente nele uma classe Java chamada **LancadorDeDadoBean**.

```
1 package sessionbeans;
2
3 import java.util.Random;
4
5 import javax.ejb.Stateless;
6
7 @Stateless
8 public class LancadorDeDadoBean {
9     private Random gerador = new Random();
10
11     public int lanca(){
12         return this.gerador.nextInt(5) + 1;
13     }
14 }
```

12. No projeto **dadoWeb**, adicione um pacote chamado **managedbeans** acrescenta nele uma classe Java chamada **DadoMB**.

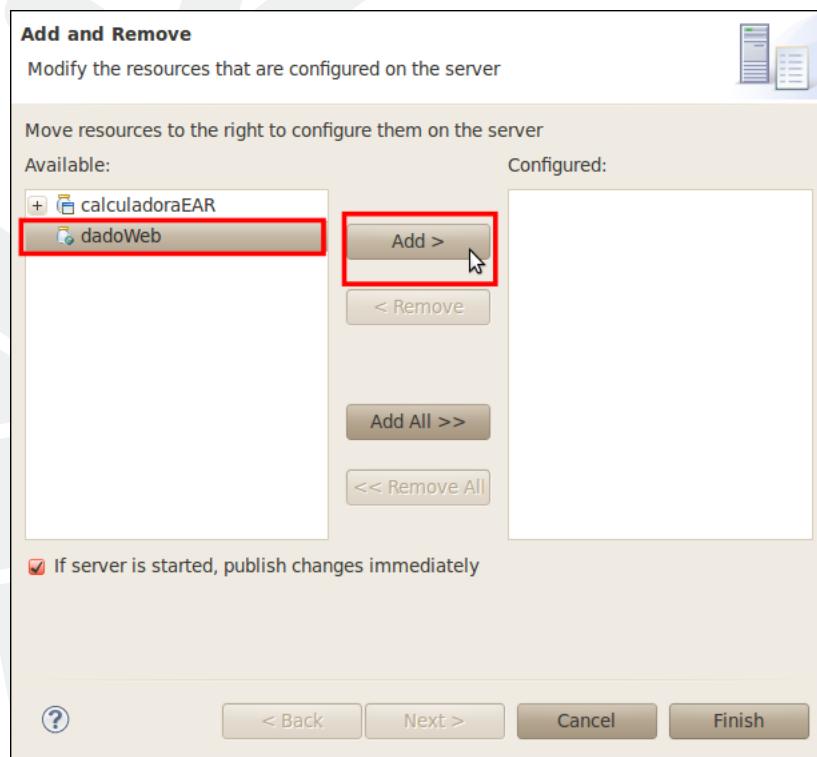
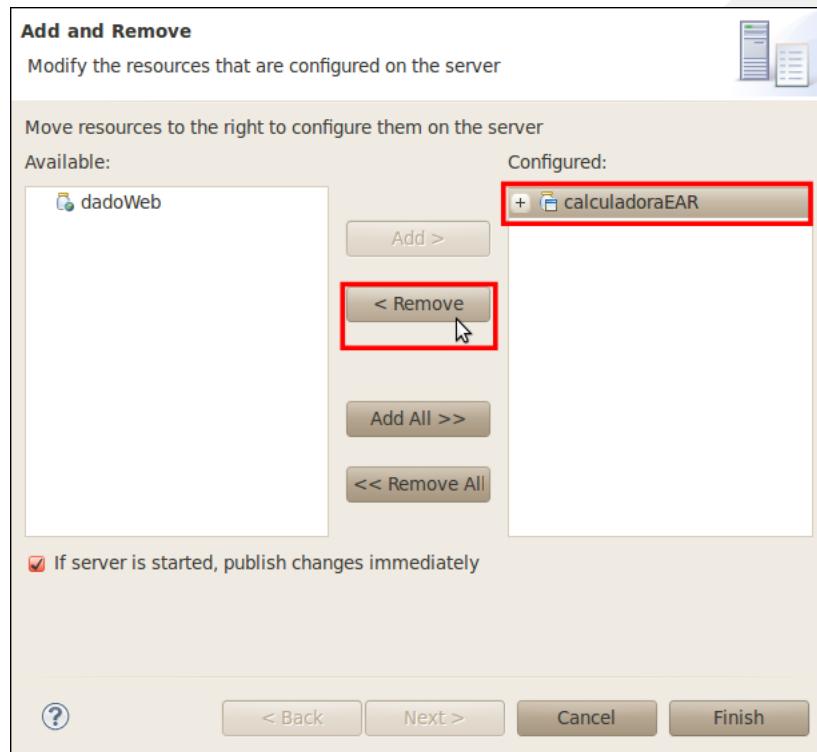
```
1 package managedbeans;
2
3 import javax.ejb.EJB;
4 import javax.faces.bean.ManagedBean;
5
6 import sessionbeans.LancadorDeDadoBean;
7
8 @ManagedBean
9 public class DadoMB {
10     @EJB
11     private LancadorDeDadoBean lancadorDeDadoBean;
12
13     private int resultado;
14
15     public void lancaDado() {
16         this.resultado = this.lancadorDeDadoBean.lanca();
17     }
18
19     public int getResultado() {
20         return resultado;
21     }
22
23     public void setResultado(int resultado) {
24         this.resultado = resultado;
25     }
26 }
```

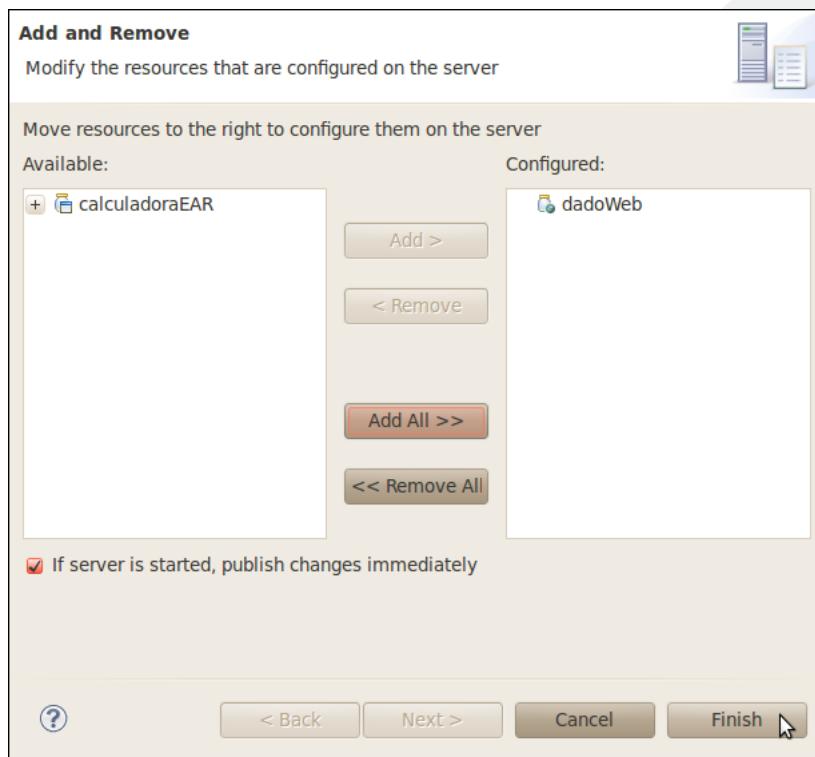
13. Crie uma simples tela na aplicação web para utilizar o Manager Bean. Adicione o arquivo **dado.xhtml** na pasta **WebContent** do projeto **dadoWeb** com o seguinte conteúdo.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5      xmlns:h="http://java.sun.com/jsf/html">
6
7 <h:head>
8     <title>Lançador de dado</title>
9 </h:head>
10
11 <h:body>
12     <h:form>
13         <h:commandButton action="#{dadoMB.lancaDado}" value="Lança o Dado"/>
14
15         <h:outputLabel value="Resultado: "/>
16         <h:outputText value="#{dadoMB.resultado}"/>
17     </h:form>
18 </h:body>
19 </html>
```

14. Adicione o projeto **dadoWeb** no glassfish. Clique com o botão direito no glassfish da view Servers e escolha a opção “Add and Remove”. Depois, siga as imagens abaixo.

## Stateless Session Beans





15. Acesse a url <http://localhost:8080/dadoWeb/dado.xhtml> e teste o funcionamento da aplicação.

## 2.9 Cliente Java SE Remoto

Vimos os SLSBs sendo acessados localmente por aplicações web implantadas no mesmo servidor de aplicação. Contudo, eles podem ser acessados remotamente, ou seja, podem ser acessados por aplicações fora do mesmo servidor de aplicação. Inclusive, um SLSB pode ser acessado por aplicações Java SE.

Quando o acesso é local, podemos injetar um SLSB através da anotação @EJB no componente que necessita dos serviços implementados pelo SLSB. Agora, quando o acesso é remoto, não temos o recurso de injeção de dependência. Dessa forma, os SLSBs devem ser obtidos de outra maneira.

Todo SLSB implantado em um servidor de aplicação recebe um “nome”. Toda aplicação fora desse servidor de aplicação pode utilizar esse nome para obter a referência remota do SLSB.

Antes da versão 3.1, os nomes dos Session Beans não eram padronizados. Consequentemente, cada servidor de aplicação possuía uma regra diferente para nomear os Session Beans. A partir da versão 3.1, os nomes foram padronizados e portanto são portáveis (iguais em todos os servidores de aplicação). Consulte a especificação para conhecer as regras de nomenclatura <http://jcp.org/en/jsr/detail?id=318>.

Uma aplicação Java remota deve acessar o serviço de nomes (JNDI) do servidor de aplicação no qual o SLSB que ela deseja utilizar está implantado. O trecho de código Java para fazer uma consulta por um SLSB no JNDI teria o seguinte padrão.

## Stateless Session Beans

---

```
1 InitialContext ic = new InitialContext();
2 StatelessSessionBean statelessSessionBean =
3     (StatelessSessionBean) ic.lookup("java:global/aplicacaoWeb/StatelessSessionBean");
```

Uma vez com a referência do SLSB, a aplicação pode chamar as operações normalmente como se o Session Bean estivesse local. Contudo, é importante ressaltar que as chamadas são remotas e portanto mais demoradas.

## 2.10 Exercícios

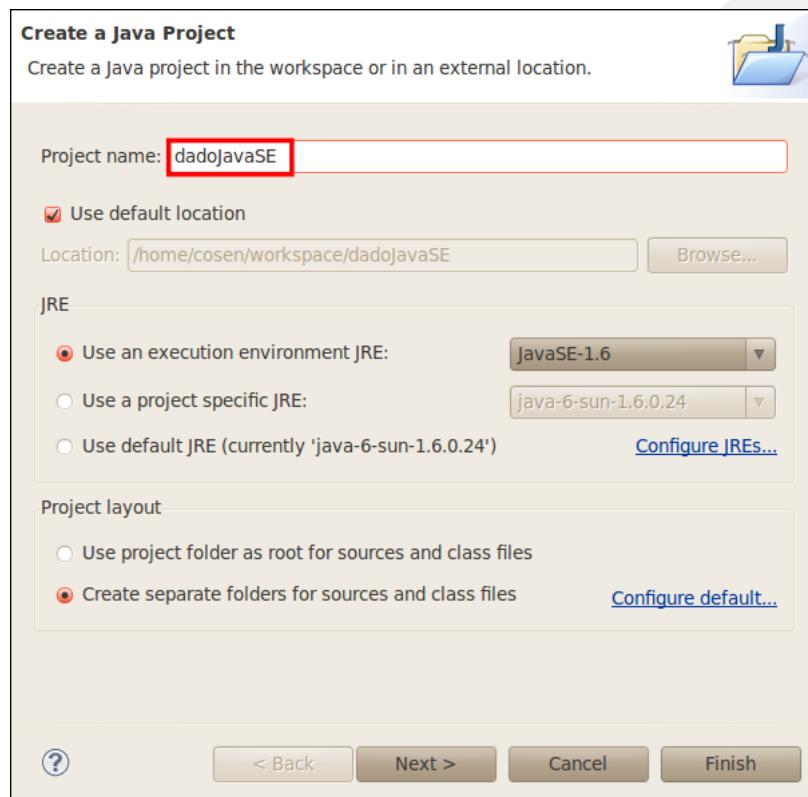
16. Adicione uma interface Java no pacote **sessionbeans** chamada **LancadorDeDados**.

```
1 package sessionbeans;
2
3 public interface LancadorDeDados {
4     public int lanca();
5 }
```

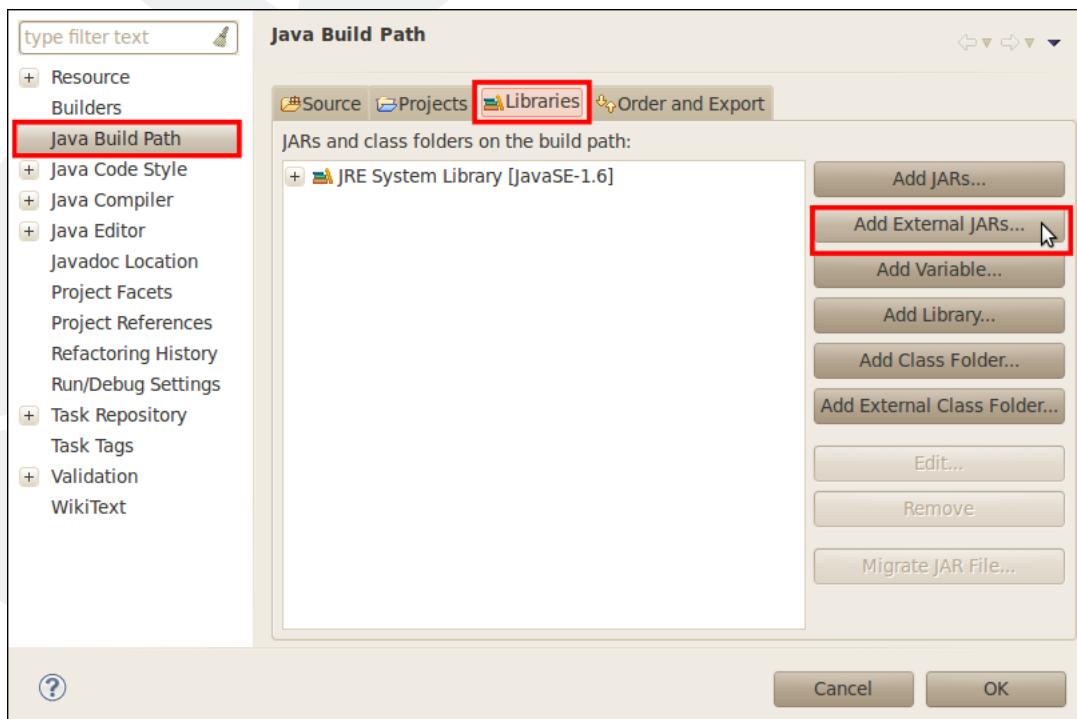
17. Altere a classe **LancadorDeDadosBean** do projeto **dadoWeb**.

```
1 @Stateless
2 @Remote(LancadorDeDados.class)
3 public class LancadorDeDadosBean implements LancadorDeDados{
4     private Random gerador = new Random();
5
6     public int lanca(){
7         return this.gerador.nextInt(5) + 1;
8     }
9 }
```

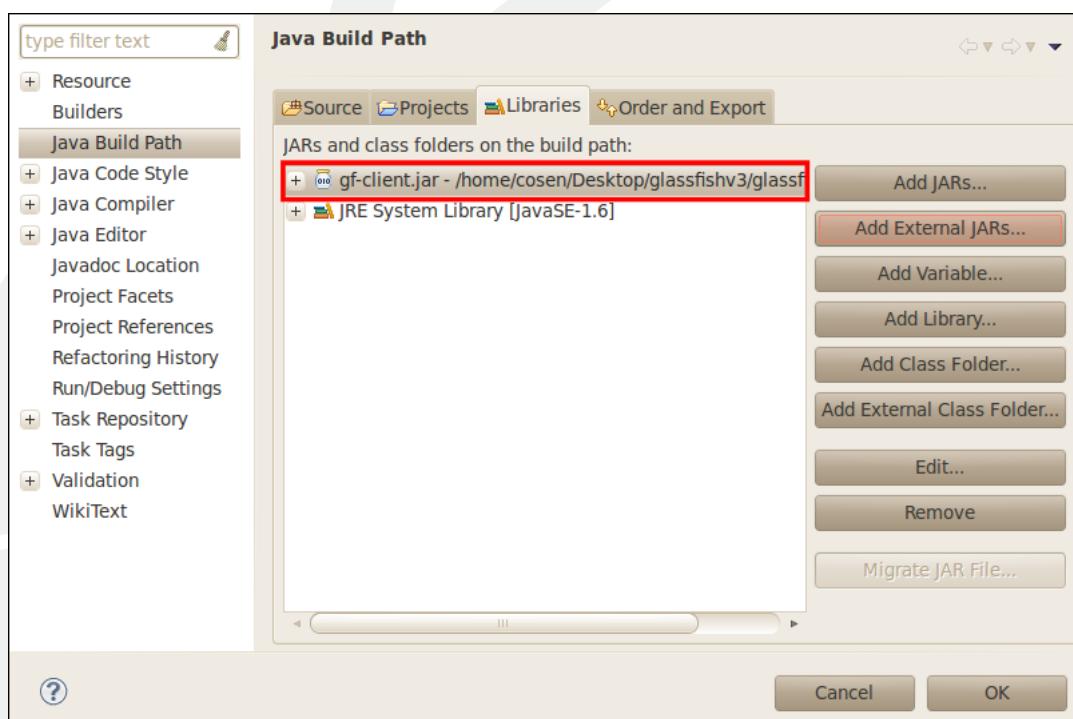
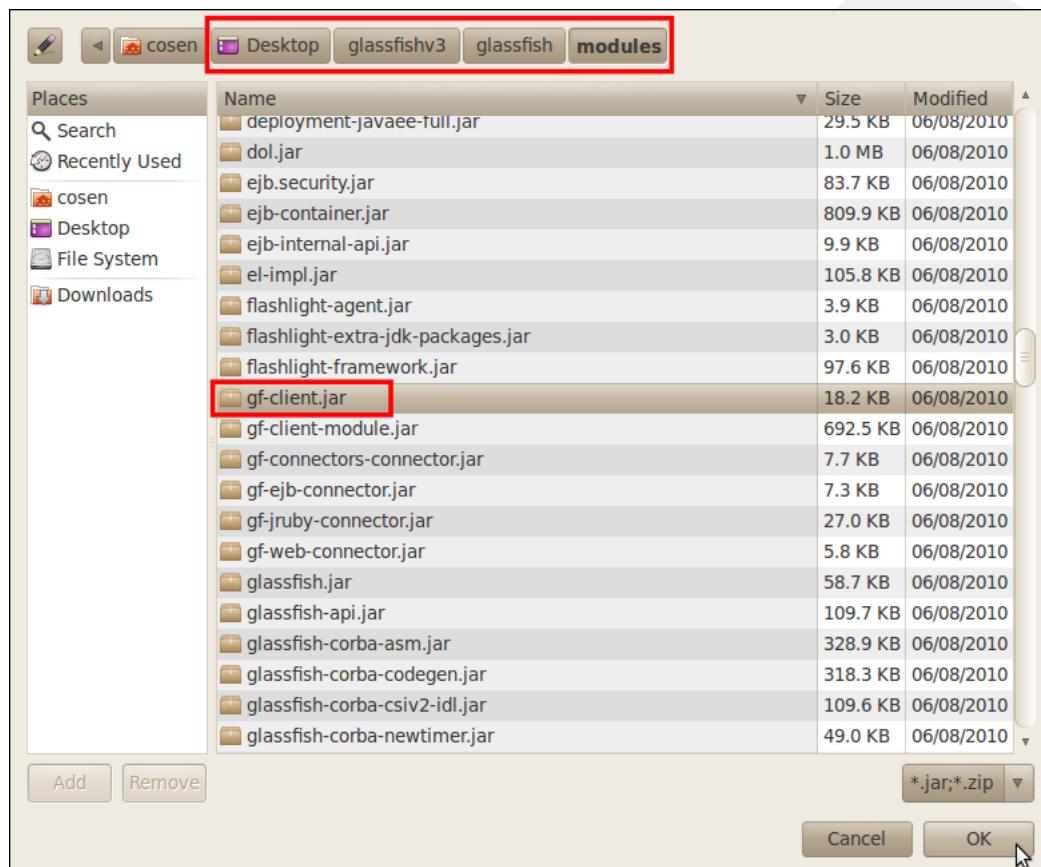
18. Crie um Java project no eclipse. Você pode digitar “CTRL+3” em seguida “new Java project” e “ENTER”. Depois, siga exatamente as imagens abaixo.



19. Adicione as bibliotecas do Glassfish necessárias para a consulta ao serviço de nomes. Abra as propriedades do projeto **dadoJavaSE**. Você pode selecionar o projeto **dadoJavaSE** e digitar “ALT+ENTER”. Depois, siga as imagens abaixo.

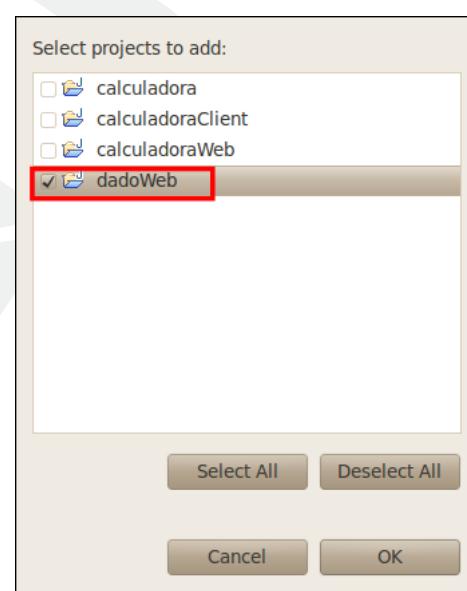
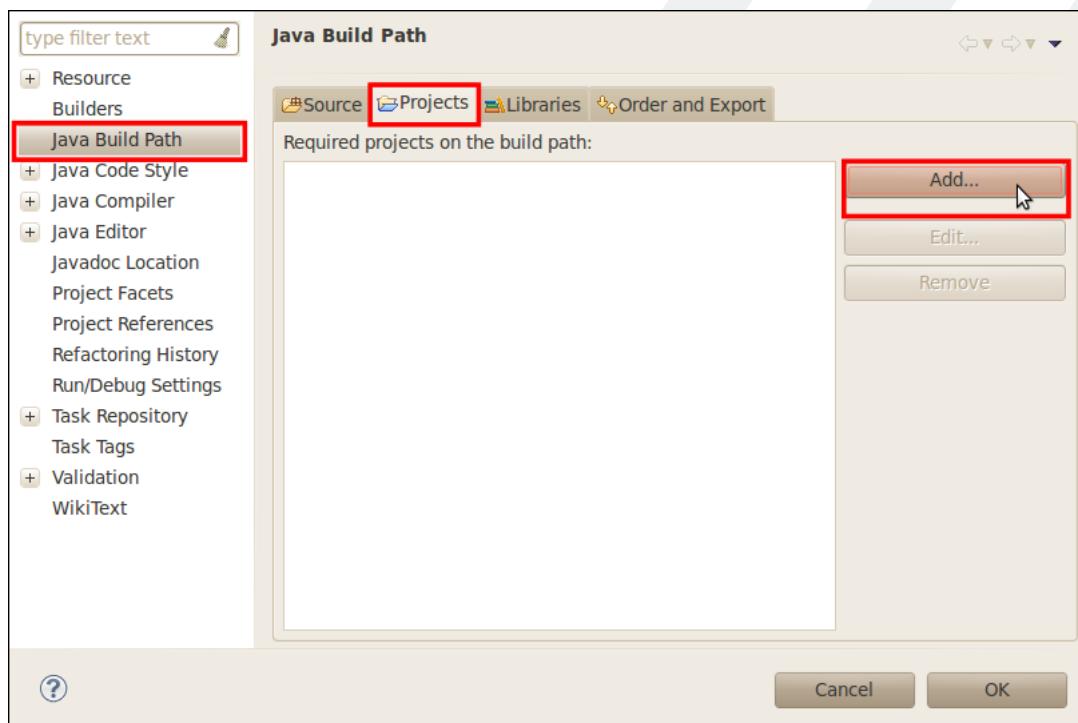


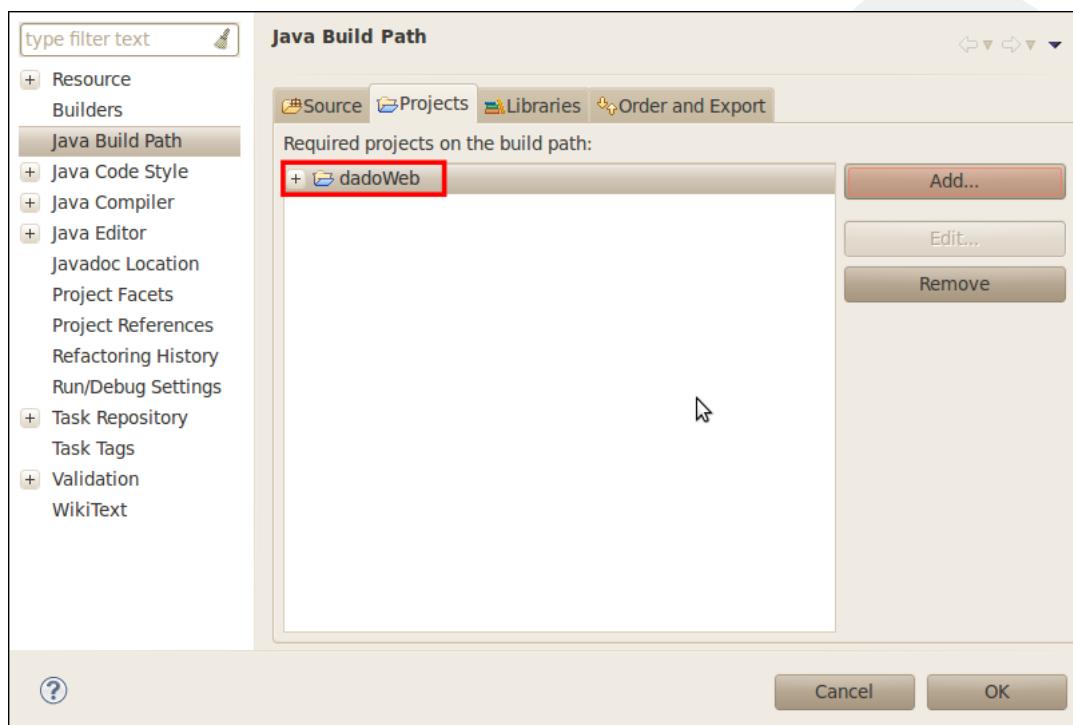
## Stateless Session Beans



20. Adicione o projeto **dadoWeb** como dependência do projeto **dadoJavaSE**. Abra as pro-

priedades do projeto **dadoJavaSE**. Você pode selecionar o projeto **dadoJavaSE** e digitar “ALT+ENTER”. Depois, siga as imagens abaixo





21. No projeto **dadoJavaSE**, adicione um pacote chamado **testes** acrescenta nele uma classe Java chamada **TesteDeAcesso**.

```
1 public class TesteDeAcesso {
2     public static void main(String[] args) throws Exception{
3         InitialContext ic = new InitialContext();
4
5         LancadorDeDado lancadorDeDado = (LancadorDeDado) ic.lookup("java:global/dadoWeb←
6             /LancadorDeDadoBean");
7         System.out.println(lancadorDeDado.lanca());
8     }
}
```

**Execute e confira o resultado no console**

## 2.11 Ciclo de Vida

As instâncias dos SLSBs são administradas pelo EJB Container. Devemos entender o de ciclo de vida desses objetos para utilizar corretamente a tecnologia EJB. Três aspectos fundamentais dos SLSBs nos ajudam a entender o ciclo de vida das instâncias.

1. Uma única instância de um SLSB pode atender chamadas de diversos clientes.
2. Uma instância de um SLSB não atende duas chamadas ao mesmo tempo. Em outras palavras, ela processa uma chamada de cada vez.
3. O EJB Container pode criar várias instâncias do mesmo SLSB para atender mais rapidamente as chamadas dos clientes.

### 2.11.1 Estados

O ciclo de vida das instâncias de um SLSB possui apenas dois estados.

1. NÃO EXISTE
2. PRONTO

### 2.11.2 NÃO EXISTE -> PRONTO

Antes de ser criada, dizemos que uma instância de um SLSB se encontra no estado NÃO EXISTE. Obviamente, nesse estado, uma instância não pode atender chamadas dos clientes.

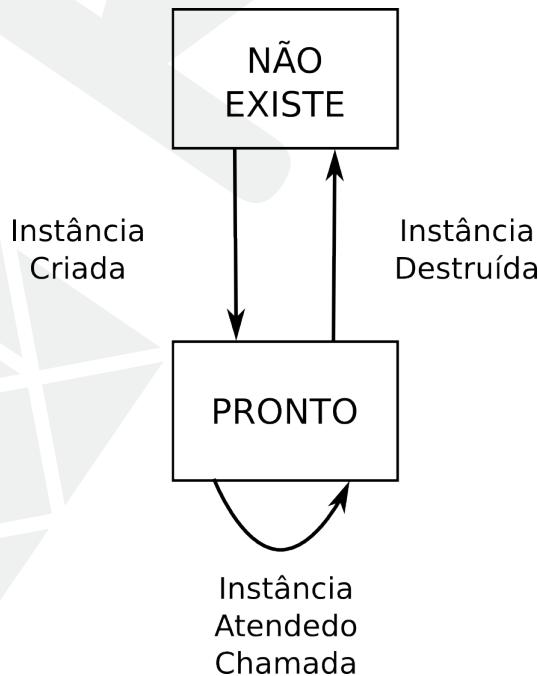
De acordo com a quantidade de chamadas e critérios de cada servidor de aplicação, o EJB Container pode criar novas instâncias de um SLSB. Cada instância criada passa para o estado PRONTO. No estado PRONTO, uma instância está apta a receber uma chamada.

### 2.11.3 PRONTO -> PRONTO

Quando uma chamada é realizada, o EJB Container seleciona uma instância entre as que estejam no estado PRONTO para realizar o atendimento. Enquanto, uma instância está atendendo uma chamada ela não pode atender outras chamadas. Depois de finalizar o atendimento, a instância volta para o estado PRONTO podendo receber outra chamada.

### 2.11.4 PRONTO -> NÃO EXISTE

Novamente, de acordo com a quantidade de chamadas e critérios de cada servidor de aplicação, o EJB Container pode destruir instâncias que estejam no estado PRONTO. Essas instâncias voltam para o estado NÃO EXISTE.



## 2.11.5 Escalabilidade e Pool

As características dos SLSBs favorecem a escalabilidade da aplicação pois, de acordo com a demanda, o EJB Container cria novas instâncias e cada instância pode atender vários clientes.

O EJB Container administra as instâncias criadas através de um Pool. Cada servidor de aplicações oferece configurações específicas para melhorar a eficiência no atendimento das chamadas. Por exemplo, o Glassfish permite que uma quantidade máxima de instâncias de um determinado SLSB seja definido pela aplicação.

## 2.11.6 Callbacks

Podemos associar lógicas específicas nas transições de estado no ciclo de vida dos SLSBs.

### @PostConstruct

Podemos registrar um método de instância no EJB Container para que ele o execute em cada instância logo após ela ser criada. Esse registro é realizado através da anotação **@PostConstruct**.

```
1 @Stateless
2 public class CalculadoraBean {
3
4     @PostConstruct
5     public void inicializando() {
6         System.out.println("Mais uma calculadora criada...")
7     }
8
9     // METODOS DE NEGOCIO
10 }
```

O EJB Container utiliza o construtor sem argumentos para criar uma instância de um SLSB. Depois de chamar o construtor sem argumentos, o EJB Container injeta eventuais dependências na instância criada. Por fim, os métodos anotados com **@POSTCONSTRUCT** são executados.

### @PreDestroy

Também podemos registrar um método de instância no EJB Container para que ele o execute em cada instância imediatamente antes dela ser destruída. Esse registro é realizado através da anotação **@PreDestroy**.

```
1 @Stateless
2 public class CalculadoraBean {
3
4     @PreDestroy
5     public void destruindo() {
6         System.out.println("Mais uma calculadora será destruída...")
7     }
8
9     // METODOS DE NEGOCIO
10 }
```

## 2.12 Exercícios

22. Adicione métodos de callbacks na classe **LancadorDeDadoBean** do projeto **dadoWeb**.

```

1  @Stateless
2  @Remote(LancadorDeDado.class)
3  public class LancadorDeDadoBean implements LancadorDeDado {
4      private Random gerador = new Random();
5
6      @PostConstruct
7      public void inicializando() {
8          System.out.println("Mais um lançador de dado criado...");
9      }
10
11     @PreDestroy
12     public void destruindo() {
13         System.out.println("Mais um lançador de dado será destruído...");
14     }
15
16     public int lanca(){
17         return this.gerador.nextInt(5) + 1;
18     }
19 }
```

23. Adicione um teste no pacote **testes** do projeto **dadoJavaSE** para fazer consultas em paralelo ao SLSB que lança moedas.

```

1  public class TesteCicloDeVidaSLSB {
2      public static void main(String[] args) throws Exception {
3          InitialContext ic = new InitialContext();
4
5          for (int i = 0; i < 100; i++) {
6              final LancadorDeDado lancadorDeDado = (LancadorDeDado) ic
7                  .lookup("java:global/dadoWeb/LancadorDeDadoBean");
8
9              Thread thread = new Thread(new Runnable() {
10
11                  @Override
12                  public void run() {
13                      for(int i = 0; i < 100; i++) {
14                          System.out.println(lancadorDeDado.lanca());
15                      }
16                  }
17              });
18              thread.start();
19          }
20      }
21 }
```

**Reinic peace o Glassfish depois execute o teste e observe o console para conferir as mensagens dos métodos de callback**

24. Pare o Glassfish e observe o console para conferir as mensagens dos métodos de callback.

# Capítulo 3

## Stateful Session Beans

### 3.1 Caracterizando os SFSBs

Stateful Session Bean é o segundo tipo de Session Bean. Muitas vezes, utilizaremos a sigla SFSB para fazer referência a esse tipo de componente. A ideia fundamental por trás dos SFSBs é a necessidade de manter estado entre as execuções das regras de negócio que eles implementam.

#### 3.1.1 Carrinho de Compras

Para exemplificar, suponha o funcionamento de um carrinho de compras de uma loja virtual. As regras de negócio do carrinho podem ser implementadas através de alguns métodos.

```
1 class CarrinhoBean {  
2       
3     public void adiciona(Produto produto) {  
4         // lógica para adicionar produto  
5     }  
6       
7     public void remove(Produto produto) {  
8         // lógica para remover produto  
9     }  
10      
11    public void finalizaCompra(){  
12        // lógica para finalizar a compra  
13    }  
14}
```

Há duas necessidades fundamentais no exemplo do carrinho de compras que devemos observar. Primeiro, uma instância da classe CARRINHOBEAN não deve atender vários clientes para não misturar produtos escolhidos por clientes diferentes. Segundo, os produtos adicionados devem ser mantidos entre as chamadas dos métodos da classe CARRINHOBEAN. Em outras palavras, é necessário manter o estado do carrinho.

Provavelmente, o estado do carrinho, ou seja, os produtos adicionados seria mantido em uma lista ou em um conjunto.

```

1 class CarrinhoBean {
2
3     private Set<Produto> produtos = new HashSet<Produto>();
4
5     public void adiciona(Produto produto) {
6         this.produtos.add(produto);
7     }
8
9     public void remove(Produto produto) {
10        this.produtos.remove(produto);
11    }
12
13    public void finalizaCompra(){
14        // lógica para finalizar a compra
15    }
16}

```

### 3.1.2 Prova Digital

Outro exemplo, suponha o funcionamento de um sistema para aplicar provas que permita que os usuários respondam as questões em qualquer ordem ou modifiquem respostas já realizadas antes de finalizar a prova. As resposta poderiam ser mantidas em um mapa.

```

1 class ProvaBean {
2     private Map<Integer, Character> respostas = new HashMap<Integer, Character>();
3
4     public void responde(Integer questao, Character resposta) {
5         this.respostas.put(questao, resposta);
6     }
7
8     public void finaliza(){
9         // lógica para finalizar a prova
10    }
11}

```

Uma instância da classe PROVABEAN não pode atender dois clientes para não misturar as respostas de dois usuários diferentes. Além disso, as respostas já realizadas devem ser mantidas entre as chamadas.

### 3.1.3 TrackList

Mais um exemplo, suponha o funcionamento de um player de vídeo que permite que os usuários selecionem um conjunto de vídeos para assistir.

```

1 class ListaDeVideos {
2     private List<Video> videos = new ArrayList<Video>();
3
4     public void adiciona(Video video) {
5         this.videos.add(video);
6     }
7
8     public void embaralha() {
9         Collections.shuffle(this.videos);
10    }
11}

```

Novamente, cada instância da classe LISTADEVIDEOS deve ser exclusiva para um cliente e os vídeos adicionados devem ser mantidos entre as chamadas dos métodos.

## 3.2 SFSB - EJB 3.0

O primeiro passo para implementar um SFSB é definir a sua interface de utilização através de uma interface Java. Por exemplo, suponha um SFSB que o funcionamento do carrinho de compras. Uma possível interface de utilização para esse Session Bean seria:

```
1 public interface Carrinho {  
2     void adiciona(Produto produto);  
3     void remove(Produto produto);  
4 }
```

Após definir a interface de utilização, o segundo passo seria implementar as operações do SFSB através de uma classe Java.

```
1 public class CarrinhoBean implements Carrinho {  
2  
3     private Set<Produto> produtos = new HashSet<Produto>();  
4  
5     public void adiciona(Produto produto) {  
6         this.produtos.add(produto);  
7     }  
8  
9     public void remove(Produto produto) {  
10        this.produtos.remove(produto);  
11    }  
12 }
```

O terceiro passo é especificar o tipo de Session Bean que queremos utilizar. No caso do carrinho, o tipo seria SFSB. Essa definição é realizada através da anotação **@Stateful**.

```
1 @Stateful  
2 public class CarrinhoBean implements Carrinho {  
3     ...  
4 }
```

Por fim, é necessário definir se o SFSB poderá ser acessado remotamente ou apenas localmente. Quando o acesso a um SLSB é local, ele só pode ser acessado por aplicações que estejam no mesmo servidor de aplicação que ele. Caso contrário, quando o acesso a um SLSB é remoto, ele pode ser acessado tanto por aplicações que estejam no mesmo servidor de aplicação quanto aplicações que não estejam.

A definição do tipo de acesso é realizada através das anotações: **@Local** e **@Remote**.

```
1 @Stateful  
2 @Remote(Carrinho.class)  
3 public class CarrinhoBean implements Carrinho {  
4     ...  
5 }
```

```

1 @Stateful
2 @Local(Carrinho.class)
3 public class CarrinhoBean implements Carrinho {
4     ...
5 }
```

### 3.3 SFSB - EJB 3.1

Na versão 3.1, quando o acesso a um SFSB é local, não é mais necessário definir uma interface Java nem utilizar a anotação @LOCAL. Então, bastaria implementar uma classe Java com a anotação @STATEFUL.

```

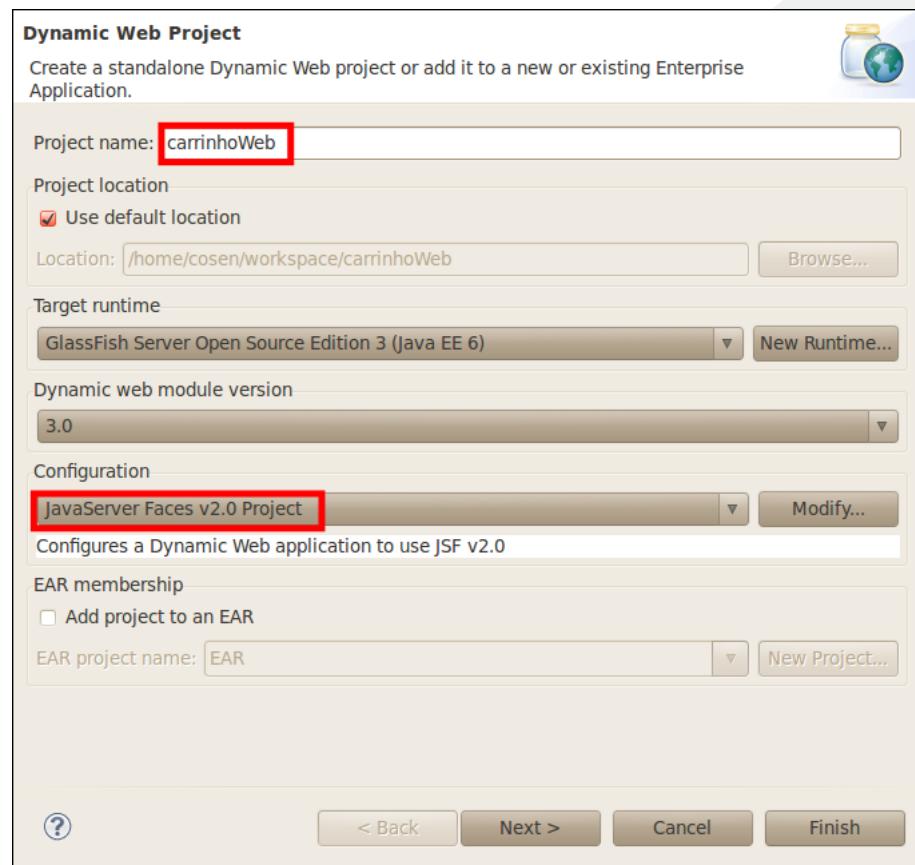
1 @Stateful
2 public class CarrinhoBean {
3
4     private Set<Produto> produtos = new HashSet<Produto>();
5
6     public void adiciona(Produto produto) {
7         this.produtos.add(produto);
8     }
9
10    public void remove(Produto produto) {
11        this.produtos.remove(produto);
12    }
13 }
```

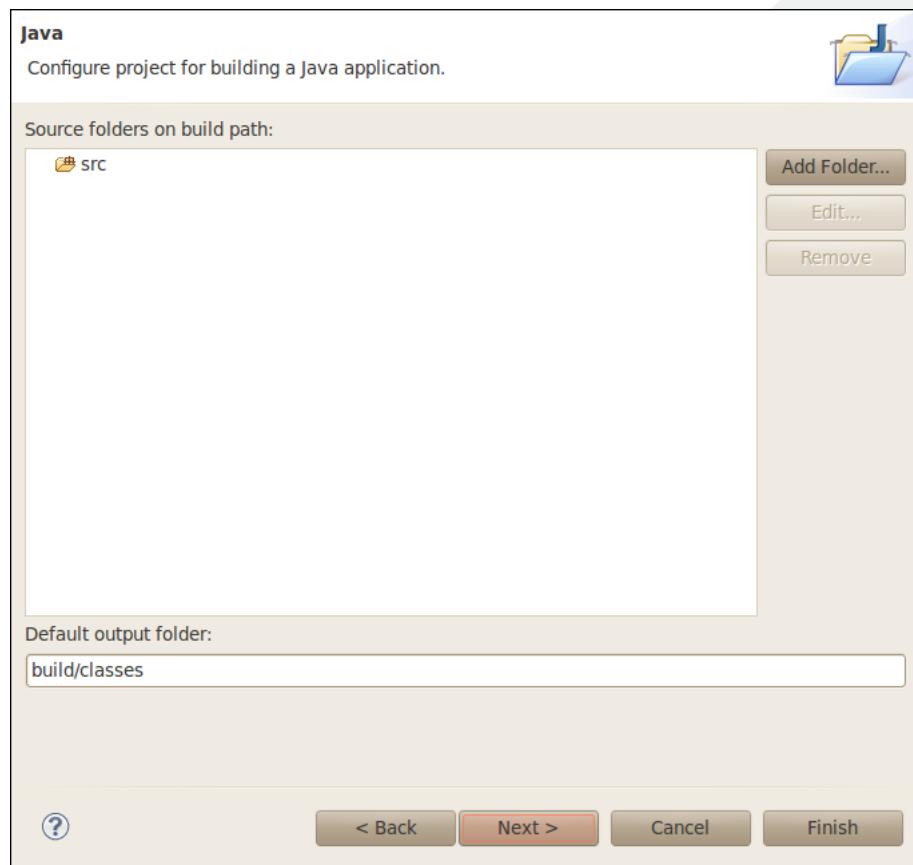
### 3.4 Exercícios

1. Crie um Dynamic Web Project no eclipse para implementar o funcionamento do carrinho de compras. Você pode digitar “CTRL+3” em seguida “new Dynamic Web Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.

## *Stateful Session Beans*

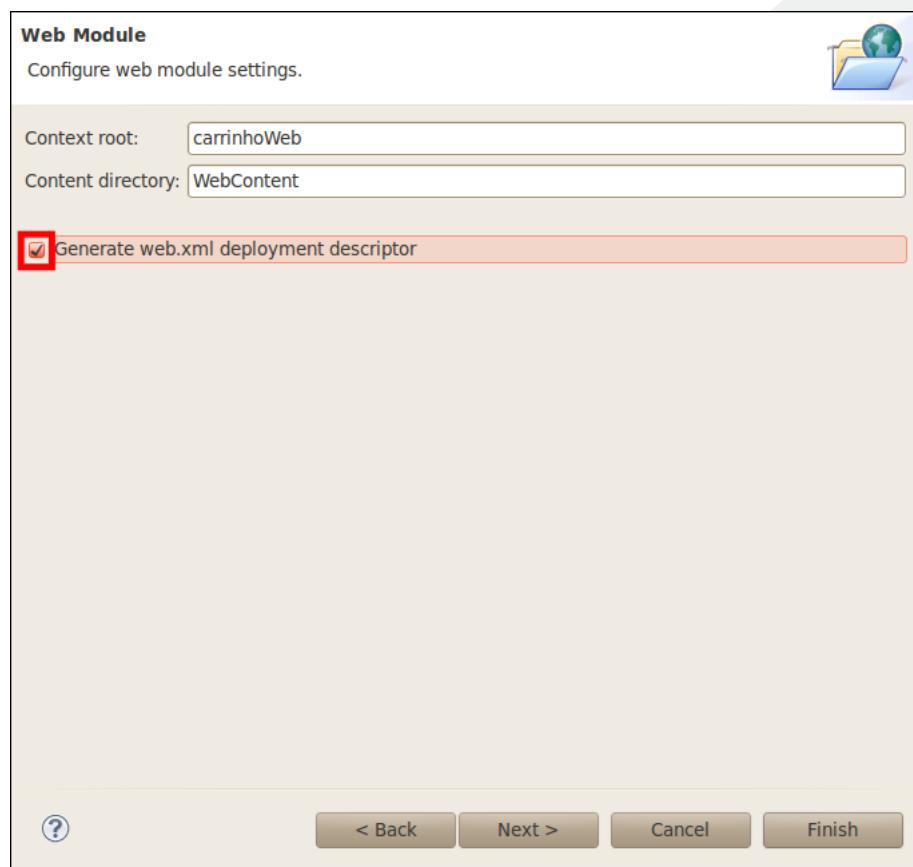
---

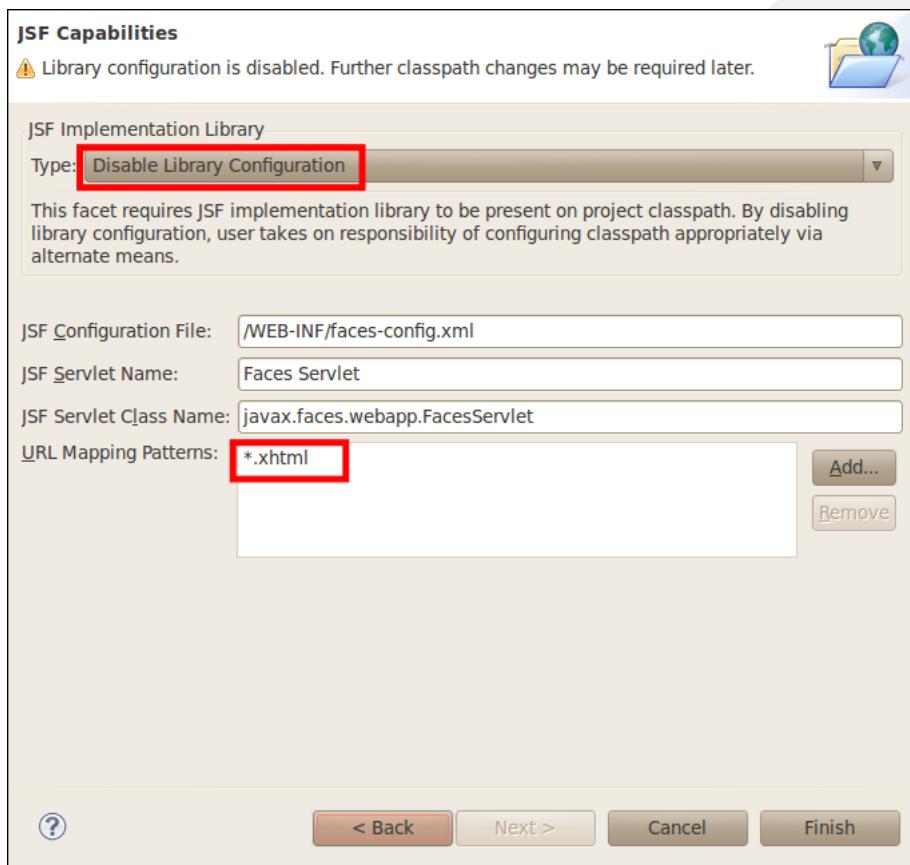




## *Stateful Session Beans*

---





2. Crie um pacote chamado **sessionbeans** e adicione a seguinte classe.

```

1 package sessionbeans;
2
3 import java.util.HashSet;
4 import java.util.Set;
5
6 import javax.ejb.Stateful;
7
8 @Stateful
9 public class CarrinhoBean {
10
11     private Set<String> produtos = new HashSet<String>();
12
13     public void adiciona(String produto) {
14         this.produtos.add(produto);
15     }
16
17     public void remove(String produto) {
18         this.produtos.remove(produto);
19     }
20
21     public Set<String> getProdutos() {
22         return produtos;
23     }
24 }
```

3. Crie um pacote chamado **managedbeans** e adicione a seguinte classe.

## Stateful Session Beans

---

```
1 package managedbeans;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 import javax.ejb.EJB;
7 import javax.faces.bean.ManagedBean;
8 import javax.faces.bean.SessionScoped;
9
10 import sessionbeans.CarrinhoBean;
11
12 @ManagedBean
13 @SessionScoped
14 public class CarrinhoMB {
15     @EJB
16     private CarrinhoBean carrinhoBean;
17
18     private String produto;
19
20     public List<String> getProdutos(){
21         return new ArrayList<String>(this.carrinhoBean.getProdutos());
22     }
23
24     public void adiciona(){
25         this.carrinhoBean.adiciona(this.produto);
26     }
27
28     public void remove(String produto){
29         this.carrinhoBean.remove(produto);
30     }
31
32     public void setProduto(String produto) {
33         this.produto = produto;
34     }
35
36     public String getProduto() {
37         return produto;
38     }
39 }
```

- 
4. Adicione o arquivo **produtos.xhtml** na pasta **WebContent** do projeto **carrinhoWeb** com o seguinte conteúdo.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html">
6
7  <h:head>
8   <title>Carrinho de Compras</title>
9  </h:head>
10
11 <h:body>
12   <h:form>
13     <h:outputLabel value="Produto: "/>
14     <h:inputText value="#{carrinhoMB.produto}" />
15     <h:commandButton action="#{carrinhoMB.adiciona}" value="Adiciona no carrinho"/>
16
17     <hr/>
18
19     <h:outputLabel value="Produtos no carrinho:>
20     <h:dataTable value="#{carrinhoMB.produtos}" var="p">
21       <h:column>
22         <h:outputText value="#{p}"/>
23       </h:column>
24       <h:column>
25         <h:commandLink action="#{carrinhoMB.remove(p)}" value="remove" />
26       </h:column>
27     </h:dataTable>
28   </h:form>
29 </h:body>
30 </html>
```

5. Adicione o projeto **carrinhoWeb** no glassfish e teste a aplicação acessando a url <http://localhost:8080/carrinhoWeb/produtos.xhtml>.
6. (Opcional) Faça o carrinho de compras permitir que os produtos sejam cadastrados com quantidade. Por exemplo, o usuário pode adicionar 5 canetas no seu carrinho.

## 3.5 Ciclo de Vida

As instâncias dos SFSBs são administradas pelo EJB Container. Devemos entender o de ciclo de vida desses objetos para utilizar corretamente a tecnologia EJB. Para entender mais facilmente o ciclo de vida das instâncias dos SFSBs, devemos sempre ter em mente que cada instância atende apenas um cliente.

### 3.5.1 Estados

O ciclo de vida das instâncias de um SFSB possui três estados.

1. NÃO EXISTE
2. PRONTO
3. PASSIVADO

### 3.5.2 NÃO EXISTE -> PRONTO

Antes de ser criada, dizemos que uma instância de um SFSB se encontra no estado NÃO EXISTE. Obviamente, nesse estado, uma instância não pode atender as chamadas do seu cliente.

Quando um cliente recebe por injeção ou recupera por lookup um SFSB, o EJB Container cria uma nova instância desse SFSB para atender exclusivamente esse cliente. Nesse instante, logo após ser criada, a instância se encontra no estado PRONTO e pode atender as chamadas do seu respectivo cliente.

### 3.5.3 PRONTO -> PASSIVADO

Uma instância de um SFSB fica ociosa enquanto não estiver processando uma chamada do seu cliente. Para não consumir a memória do computador, depois de um certo tempo de ociosidade, o EJB Container pode transferir o conteúdo de uma instância ociosa para dispositivos secundários de armazenamento (hard disk). Esse processo de transferência é chamado de **passivação**. Após ser passivada, a instância ociosa se encontrará no estado PASSIVADO.

Outros fatores além da ociosidade podem levar o EJB Container decidir passivar instâncias dos SFSBs. Por exemplo, quando um certo limite de instâncias no estado PRONTO (ocupando memória) for atingido.

### 3.5.4 PASSIVADA -> PRONTO

Se o cliente de uma instância passivada realizar uma chamada a ela, o EJB Container realizará automaticamente o processo de ativação. Esse processo consiste na transferência do conteúdo da instância passivada para a memória principal novamente. Após ser ativada, a instância se encontrará no estado PRONTO e apta a atender a chamada realizada.

### 3.5.5 PRONTO -> NÃO EXISTE

Em determinadas situações, uma instância de um SFSB pode não ser mais útil. Em outras palavras, o cliente correspondente pode não precisar mais dela. Por exemplo, quando o cliente de um carrinho de compras finaliza a compra, a instância que representa o carrinho pode ser destruída (não compensa reaproveitar o mesmo carrinho para outro cliente).

O EJB Container é o responsável por destruir uma instância de um SFSB que não é mais útil. Por outro lado, a aplicação é responsável por determinar quando uma instância se torna inútil. Adicionando um método de negócio anotado com **@Remove** a aplicação declara que após a execução desse método a instância não é mais necessária.

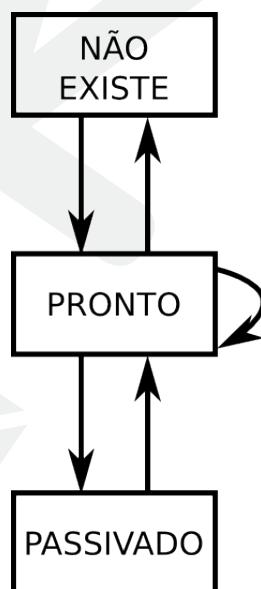
No exemplo do carrinho de compras, poderíamos definir um método para implementar a lógica de finalizar compra e anotá-lo com **@REMOVE**. Dessa forma, logo após a execução desse método a instância seria destruída pelo EJB Container.

```

1  @Stateful
2  public class CarrinhoBean {
3
4      private Set<String> produtos = new HashSet<String>();
5
6      public void adiciona(String produto) {
7          this.produtos.add(produto);
8      }
9
10     public void remove(String produto) {
11         this.produtos.remove(produto);
12     }
13
14     public Set<String> getProdutos() {
15         return produtos;
16     }
17
18     @Remove
19     public void finalizaCompra(){
20         // lógica para finalizar compra
21     }
22 }
```

### 3.5.6 PASSIVADO -> PRONTO -> NÃO EXISTE

Uma instância pode ser passivada porque ficou ociosa quando estava no estado PRONTO. Ou seja, o respectivo cliente não realizou nenhuma chamada durante um “grande” período de tempo. Da mesma maneira, quando passivada, uma instância pode não receber uma chamada do seu cliente durante um “grande” período de tempo. Nesses casos, o EJB Container pode assumir que o cliente não chamará mais a instância passivada e portanto ela não é mais útil. Quando uma instância passivada não é mais útil, o EJB Container ativa e depois a destrói.



### 3.5.7 Callbacks

Podemos associar lógicas específicas nas transições de estado no ciclo de vida dos SFSBs.

## Stateful Session Beans

---

### @PostConstruct

Podemos registrar um método de instância no EJB Container para que ele o execute em cada instância logo após ela ser criada. Esse registro é realizado através da anotação **@PostConstruct**.

```
1 @Stateless
2 public class CarrinhoBean {
3
4     @PostConstruct
5     public void inicializando() {
6         System.out.println("Mais um carrinho criado...");  

7     }
8
9     // METODOS DE NEGOCIO
10 }
```

O EJB Container utiliza o construtor sem argumentos para criar uma instância de um SLSB. Depois de chamar o construtor sem argumentos, o EJB Container injeta eventuais dependências na instância criada. Por fim, os métodos anotados com **@POSTCONSTRUCT** são executados.

### @PreDestroy

Também podemos registrar um método de instância no EJB Container para que ele o execute em cada instância imediatamente antes dela ser destruída. Esse registro é realizado através da anotação **@PreDestroy**.

```
1 @Stateless
2 public class CarrinhoBean {
3
4     @PreDestroy
5     public void destruindo() {
6         System.out.println("Mais um carrinho será destruído...");  

7     }
8
9     // METODOS DE NEGOCIO
10 }
```

### @PrePassivate

Também podemos registrar um método de instância no EJB Container para que ele o execute em cada instância imediatamente antes dela ser passivada. Esse registro é realizado através da anotação **@PrePassivate**.

```
1 @Stateless
2 public class CarrinhoBean {
3
4     @PrePassivate
5     public void passivando() {
6         System.out.println("Mais um carrinho será passivado...");  

7     }
8
9     // METODOS DE NEGOCIO
10 }
```

### @PostActivate

Também podemos registrar um método de instância no EJB Container para que ele o execute em cada instância imediatamente depois dela ser ativada. Esse registro é realizado através da anotação **@PostActivate**.

```
1 @Stateless
2 public class CarrinhoBean {
3
4     @PostActivate
5     public void ativando() {
6         System.out.println("Mais um carrinho foi ativado...");
7     }
8
9     // METODOS DE NEGOCIO
10 }
```

## 3.6 Exercícios

7. Implemente uma aplicação com interface web para aplicar provas.

# Capítulo 4

## Singleton Session Beans

### 4.1 Caracterizando os Singleton Session Beans

Singleton Session Bean é o terceiro tipo de Session Bean. Este tipo de Session Bean surgiu na versão 3.1 da especificação Enterprise Java Beans. A ideia fundamental por trás desse tipo de Session Bean é a necessidade de compartilhar dados transientes entre todos os usuários de uma aplicação EJB.

#### 4.1.1 Número de usuários conectados

Para exemplificar, suponha que seja necessário contabilizar a número de usuários conectados à aplicação. Esse serviço pode ser implementado através de uma classe.

```
1 class ContadorDeUsuariosBean {  
2     private int contador = 0;  
3  
4     public void adiciona() {  
5         this.contador++;  
6     }  
7  
8     public int getContador() {  
9         return this.contador;  
10    }  
11 }  
12 }
```

Uma única instância da classe CONTADORDEUSUARIOSBEAN deve ser criada para contabilizar corretamente o número de usuários conectados. Além disso, o contador de usuários conectados não precisa ser persistido entre duas execuções da aplicação.

#### 4.1.2 Sistema de chat

Outro exemplo, suponha o funcionamento de um sistema de chat no qual as salas são criadas dinamicamente pelos usuários durante a execução. Podemos definir alguns métodos para implementar esse sistema.

```

1 class ChatBean {
2     private Set<String> salas = new HashSet<String>();
3
4     public void criaSala(String sala) {
5         this.salas.add(sala);
6     }
7
8     public List<String> listaSalas(){
9         return new ArrayList<String>(this.salas);
10    }
11 }
```

As salas são criadas dinamicamente e todos os usuários compartilham todas as salas. Se o sistema “cair” por qualquer que seja o motivo não é necessário guardar as salas pois na próxima execução novas salas serão criadas pelos usuários. Uma única instância da classe CHATBEAN deve ser criada.

### 4.1.3 Trânsito Colaborativo

Mais um exemplo, suponha um sistema colaborativo para informar o grau de congestionamento nas vias de uma cidade. As regras desse sistema poderiam ser implementadas através de alguns métodos.

```

1 class TransitoBean {
2     private Map<String, List<Integer>> vias = new HashMap<String, List<Integer>>();
3
4     public void registra(String via, Integer velocidade) {
5         if(this.vias.containsKey(via)) {
6             this.vias.get(via).add(velocidade);
7         }
8     }
9
10    public List<Integer> getVelocidadesRegistradas(String via) {
11        return this.vias.get(via);
12    }
13 }
```

Os dados sobre o trânsito são fornecidos pelos usuários e todos podem consultar as mesmas informações. A princípio, não é necessário manter esses dados persistidos.

## 4.2 Implementação

Para implementar um Singleton Session Bean podemos definir uma interface Java com as assinaturas dos métodos desejados. Por exemplo, suponha que um Singleton Session Bean será utilizado para implementar um sistema de chat.

```

1 public interface Chat {
2     void criaSala(String sala);
3     List<String> listaSalas();
4 }
```

Após definir a interface de utilização, o segundo passo seria implementar as operações do Session Bean através de uma classe Java.

## Singleton Session Beans

---

```
1 public class ChatBean implements Chat {  
2  
3     private Set<String> salas = new HashSet<String>();  
4  
5     public void criaSala(String sala) {  
6         this.salas.add(sala);  
7     }  
8  
9     public List<String> listaSalas(){  
10        return new ArrayList<String>(this.salas);  
11    }  
12}
```

O terceiro passo é especificar o tipo de Session Bean que queremos utilizar. No caso do chat, o tipo seria Singleton. Essa definição é realizada através da anotação **@Singleton**.

```
1 @Singleton  
2 public class ChatBean implements Chat {  
3     ...  
4 }
```

Por fim, é necessário definir se o Session Bean poderá ser acessado remotamente ou apenas localmente. Quando o acesso a um Session Bean é local, ele só pode ser acessado por aplicações que estejam no mesmo servidor de aplicação que ele. Caso contrário, quando o acesso a um Session Bean é remoto, ele pode ser acessado tanto por aplicações que estejam no mesmo servidor de aplicação quanto aplicações que não estejam.

A definição do tipo de acesso é realizada através das anotações: **@Local** e **@Remote**.

```
1 @Singleton  
2 @Remote(Chat.class)  
3 public class ChatBean implements Chat {  
4     ...  
5 }
```

```
1 @Singleton  
2 @Local(Chat.class)  
3 public class ChatBean implements Chat {  
4     ...  
5 }
```

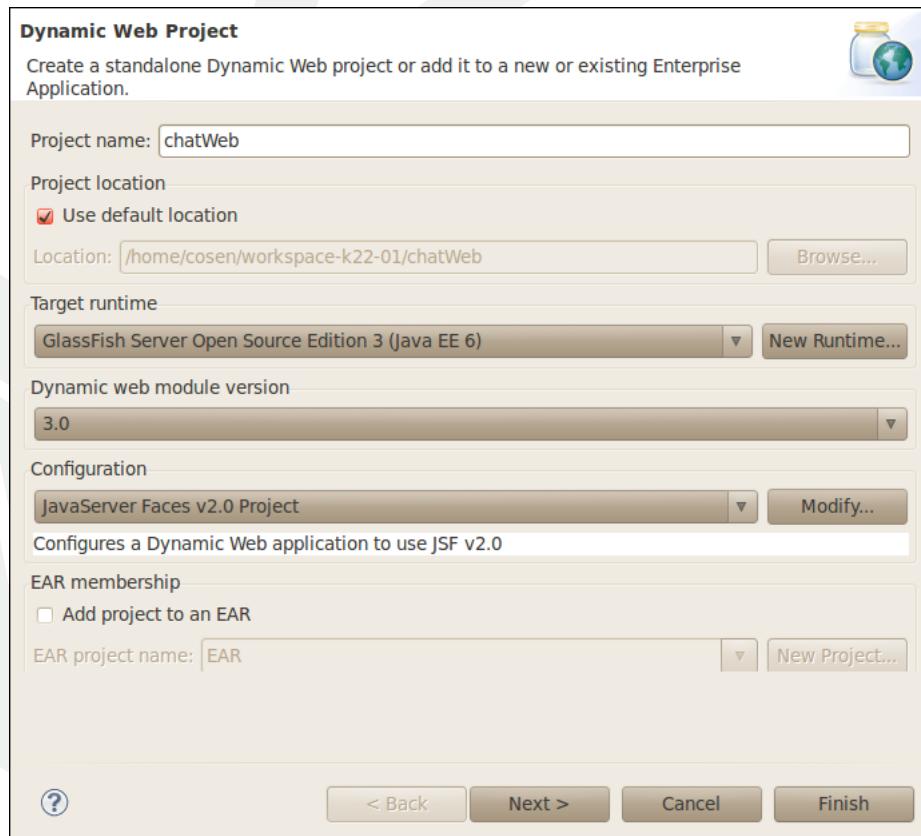
### 4.2.1 Singleton Session Beans Locais

Quando o acesso a um Singleton Session Bean é local, não é necessário definir uma interface Java nem utilizar a anotação **@LOCAL**. Então, bastaria implementar uma classe Java com a anotação **@SINGLETON**.

```
1 @Singleton
2 public class ChatBean {
3
4     private Set<String> salas = new HashSet<String>();
5
6     public void criaSala(String sala) {
7         this.salas.add(sala);
8     }
9
10    public List<String> listaSalas(){
11        return new ArrayList<String>(this.salas);
12    }
13 }
```

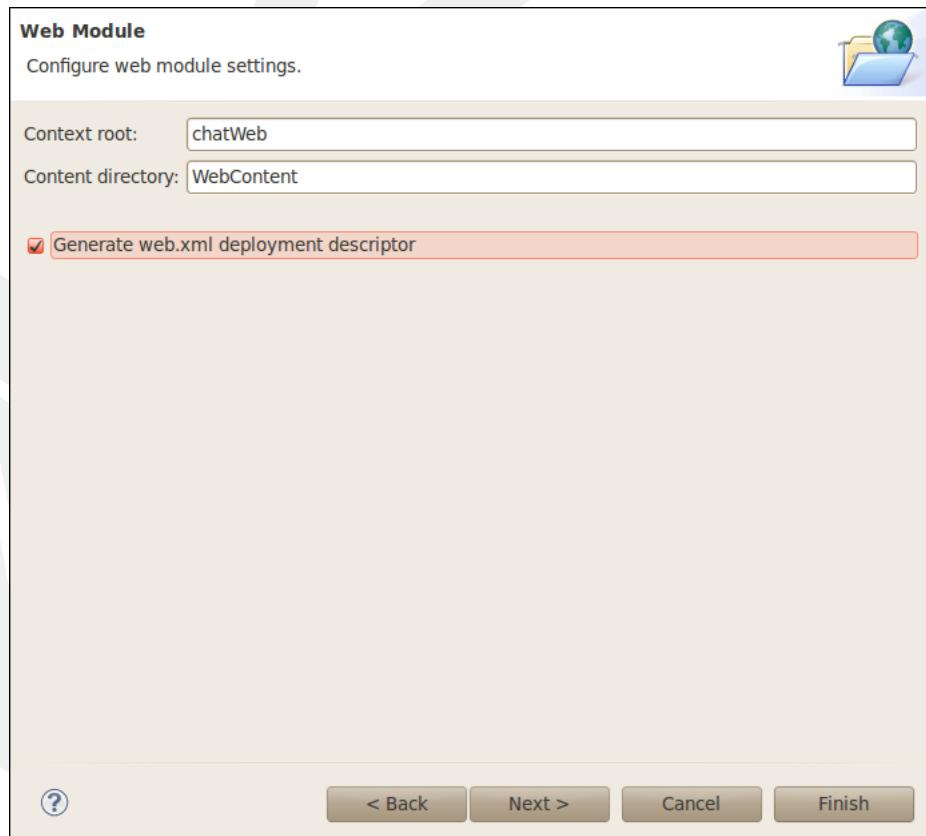
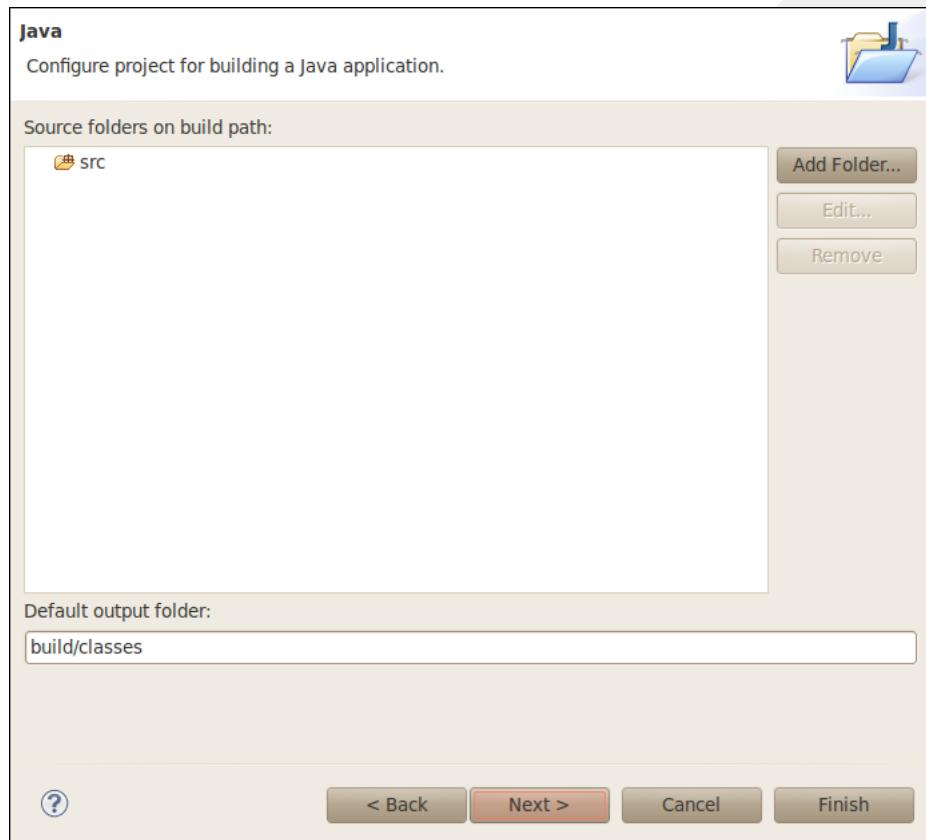
### 4.3 Exercícios

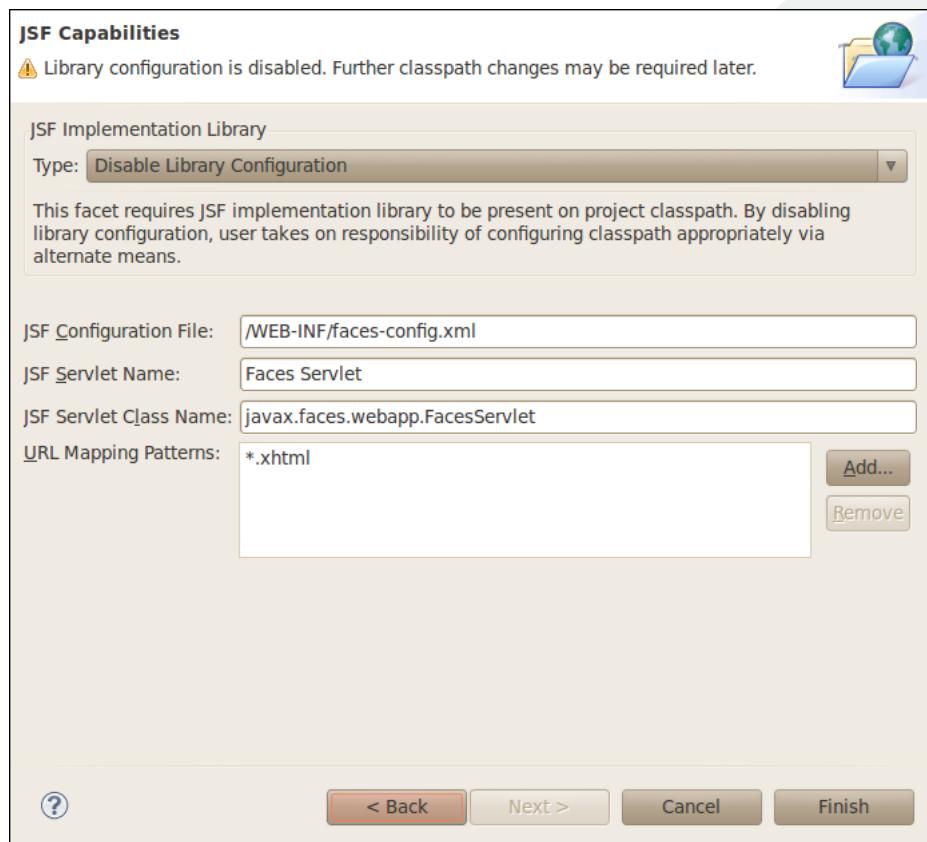
1. Crie um Dynamic Web Project no eclipse para implementar o funcionamento de um sistema de chat. Você pode digitar “CTRL+3” em seguida “new Dynamic Web Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.



## Singleton Session Beans

---





2. Crie um pacote chamado **sessionbeans** e adicione a seguinte classe.

```
1 package sessionbeans;
2
3 import java.util.HashSet;
4 import java.util.Set;
5
6 import javax.ejb.Singleton;
7
8 @Singleton
9 public class ChatBean {
10
11     private Set<String> salas = new HashSet<String>();
12
13     public void criaSala(String sala) {
14         this.salas.add(sala);
15     }
16
17     public List<String> listaSalas() {
18         return new ArrayList<String>(this.salas);
19     }
20 }
```

3. Crie um pacote chamado **managedbeans** e adicione a seguinte classe.

## *Singleton Session Beans*

---

```
1 package managedbeans;
2
3 import java.util.List;
4
5 import javax.ejb.EJB;
6 import javax.faces.bean.ManagedBean;
7
8 import sessionbeans.ChatBean;
9
10 @ManagedBean
11 public class ChatMB {
12
13     @EJB
14     private ChatBean chatBean;
15
16     private String sala;
17
18     public void adicionaSala() {
19         this.chatBean.criaSala(this.sala);
20     }
21
22     public List<String> getSalas() {
23         return this.chatBean.listaSalas();
24     }
25
26     public void setSala(String sala) {
27         this.sala = sala;
28     }
29
30     public String getSala() {
31         return sala;
32     }
33 }
```

- 
4. Adicione o arquivo **chat.xhtml** na pasta **WebContent** do projeto **chatWeb** com o seguinte conteúdo.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9  <h:head>
10 <title>Chat</title>
11 </h:head>
12
13 <h:body>
14 <h:form>
15   <h:outputLabel value="Nova Sala: "/>
16   <h:inputText value="#{chatMB.sala}" />
17   <h:commandButton value="Criar" action="#{chatMB.adicionaSala}" />
18
19   <hr/>
20
21   <h:dataTable value="#{chatMB.salas}" var="sala">
22     <h:column>
23       <h:outputText value="#{sala}" />
24     </h:column>
25   </h:dataTable>
26 </h:form>
27 </h:body>
28
29 </html>
```

5. Adicione o projeto **chatWeb** no glassfish e teste a aplicação acessando a url <http://localhost:8080/chatWeb/chat.xhtml> através de dois navegadores.
6. (Opcional) Implemente a remoção de salas.

## 4.4 Ciclo de Vida

As instâncias dos Singleton Session Beans são administradas pelo EJB Container. Devemos entender o de ciclo de vida desses objetos para utilizar corretamente a tecnologia EJB. Para entender mais facilmente o ciclo de vida das instâncias dos Singleton Session Beans, devemos sempre ter em mente que o EJB Container cria apenas uma instância de cada Session Bean desse tipo.

### 4.4.1 Estados

O ciclo de vida das instâncias dos Singleton Session Beans possui dois estados.

1. NÃO EXISTE
2. PRONTO

### 4.4.2 NÃO EXISTE -> PRONTO

Antes de ser criada, dizemos que uma instância de um Singleton Session Bean se encontra no estado NÃO EXISTE. Obviamente, nesse estado, uma instância não pode atender as chamadas dos clientes da aplicação.

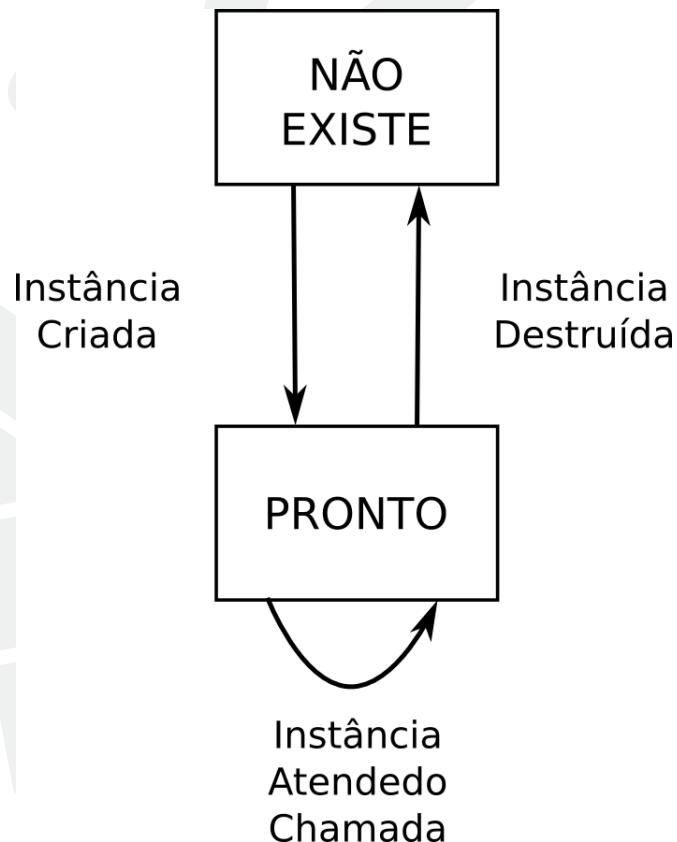
O EJB Container cria apenas uma instância para cada Singleton Session Bean. Por padrão, o EJB Container é quem decide quando a criação da instância de um Singleton Session Bean deve ser realizada. Contudo, é possível determinar que essa criação seja realizada na inicialização da aplicação através da anotação **@Startup**.

```
1 @Singleton
2 @Startup
3 class ContadorDeUsuariosBean {
4
5     private int contador = 0;
6
7     public void adiciona() {
8         this.contador++;
9     }
10
11    public int getContador() {
12        return this.contador;
13    }
14 }
```

Quando a instância de um Singleton Session Bean é criada, ela passa para do estado NÃO EXISTE para o estado PRONTO e pode atender as chamadas dos clientes da aplicação.

#### 4.4.3 PRONTO -> NÃO EXISTE

Quando a aplicação é finalizada, o EJB Container destrói as instâncias dos Singleton Session Beans. Dessa forma, elas passam do estado PRONTO para o NÃO EXISTE.



#### 4.4.4 Callbacks

Podemos associar lógicas específicas nas transições de estado no ciclo de vida dos Singleton Session Beans.

##### @PostConstruct

Podemos registrar um método de instância no EJB Container para que ele o execute em cada instância logo após ela ser criada. Esse registro é realizado através da anotação **@PostConstruct**.

```
1 @Singleton
2 class ContadorDeUsuariosBean {
3
4     @PostConstruct
5     public void inicializando() {
6         System.out.println("Contador de usuários criado...");
7     }
8
9     // METODOS DE NEGOCIO
10 }
```

O EJB Container utiliza o construtor sem argumentos para criar a instância de um Singleton Session Bean. Depois de chamar o construtor sem argumentos, o EJB Container injeta eventuais dependências na instância criada. Por fim, os métodos anotados com **@POSTCONSTRUCT** são executados.

##### @PreDestroy

Também podemos registrar um método de instância no EJB Container para que ele o execute em cada instância imediatamente antes dela ser destruída. Esse registro é realizado através da anotação **@PreDestroy**.

```
1 @Singleton
2 class ContadorDeUsuariosBean {
3
4     @PreDestroy
5     public void destruindo() {
6         System.out.println("Contador de usuários será destruído...");
7     }
8
9     // METODOS DE NEGOCIO
10 }
```

# Capítulo 5

## Persistência

### 5.1 Data Sources

Aplicações Java se comunicam com banco de dados através de conexões JDBC. Para estabelecer uma conexão JDBC, algumas informações como usuário, senha e base de dados são necessárias.

As configurações relativas às conexões JDBC podem ser definidas nas aplicações ou nos servidores de aplicação. Quando definidas em uma aplicação serão utilizadas somente por essa aplicação. Quando definidas em um servidor de aplicação podem ser utilizadas em diversas aplicações.

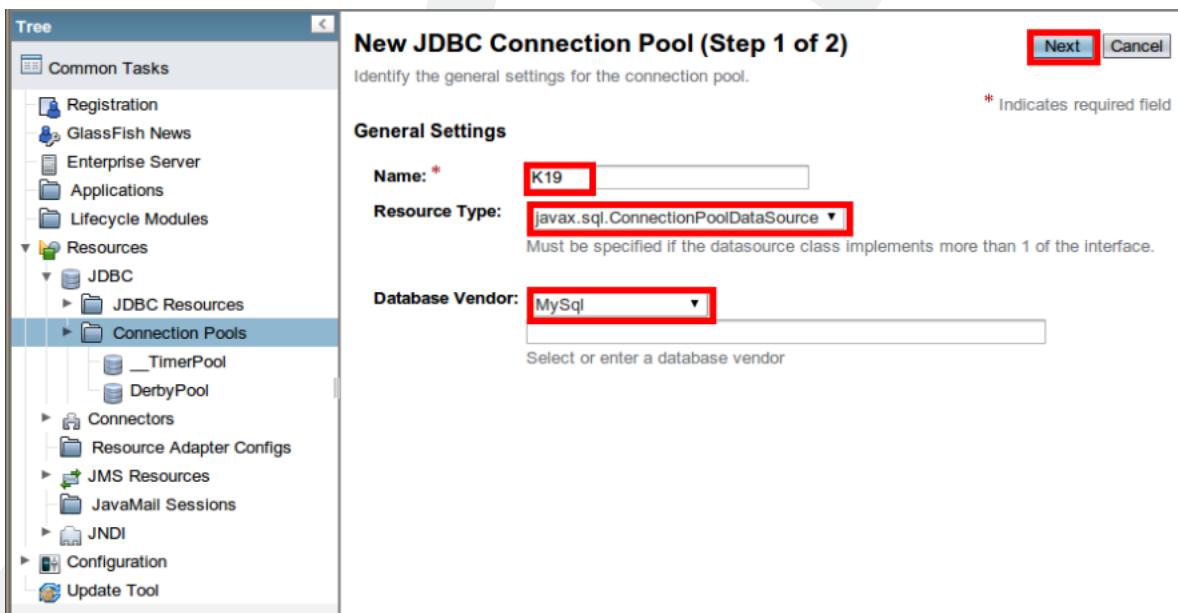
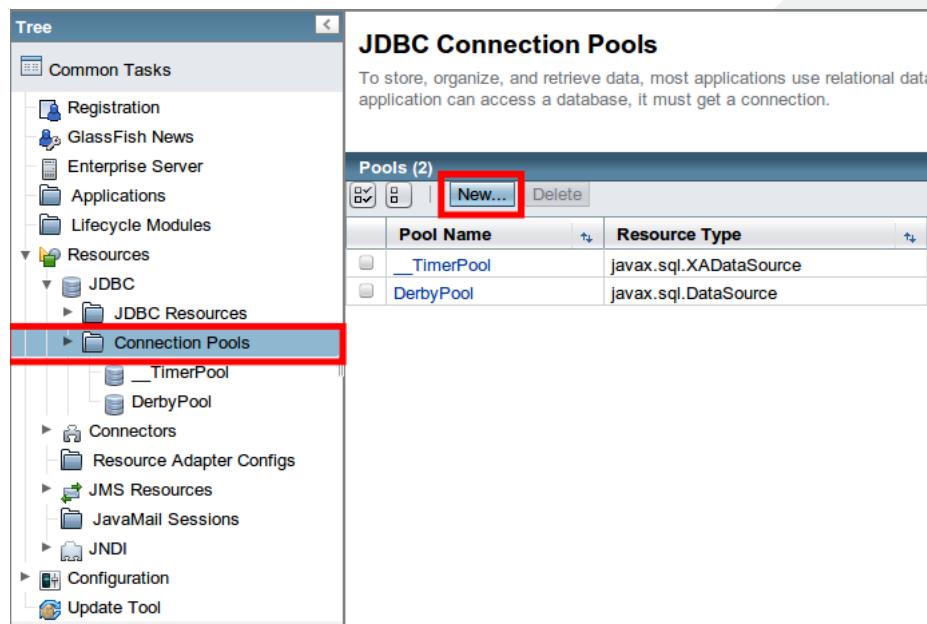
Em um servidor de aplicação, as configurações JDBC são definidas em componentes chamados **Data Sources**. A criação de Data Sources depende do servidor de aplicação utilizado. Em particular, no Glassfish, os Data Sources podem ser criados através da interface de administração.

Os Data Sources permitem que uma única configuração JDBC seja utilizada por diversas aplicações. Eles também permitem que outros tipos de configurações sejam compartilhadas. Por exemplo, a configuração de um Connection Pool.

Além disso, através de Data Sources podemos utilizar o serviço de transações dos servidores de aplicação. Esse serviço é definido pela especificação Java Transaction API (JTA).

### 5.2 Exercícios

1. Copie o arquivo **mysql-connector-java-5.1.13-bin.jar** que se encontra na pasta **K19-Arquivos/MySQL-Connector-JDBC** da Área de Trabalho para a pasta **glassfishv3/glassfish/lib** também da Área de Trabalho.
2. Através do MySQL Query Browser, apague a base de dados **k22** caso ela exista. Depois, crie uma base de dados chamada **k22**.
3. Com o Glassfish inicializado, abra a interface de administração acessando a url **localhost:4848**. Siga os passos abaixo:



4. Defina os seguintes valores para as seguintes propriedades:

**DatabaseName:** k22

**Password:** root

**ServerName:** localhost

**URL:** jdbc:mysql://localhost:3306/k22

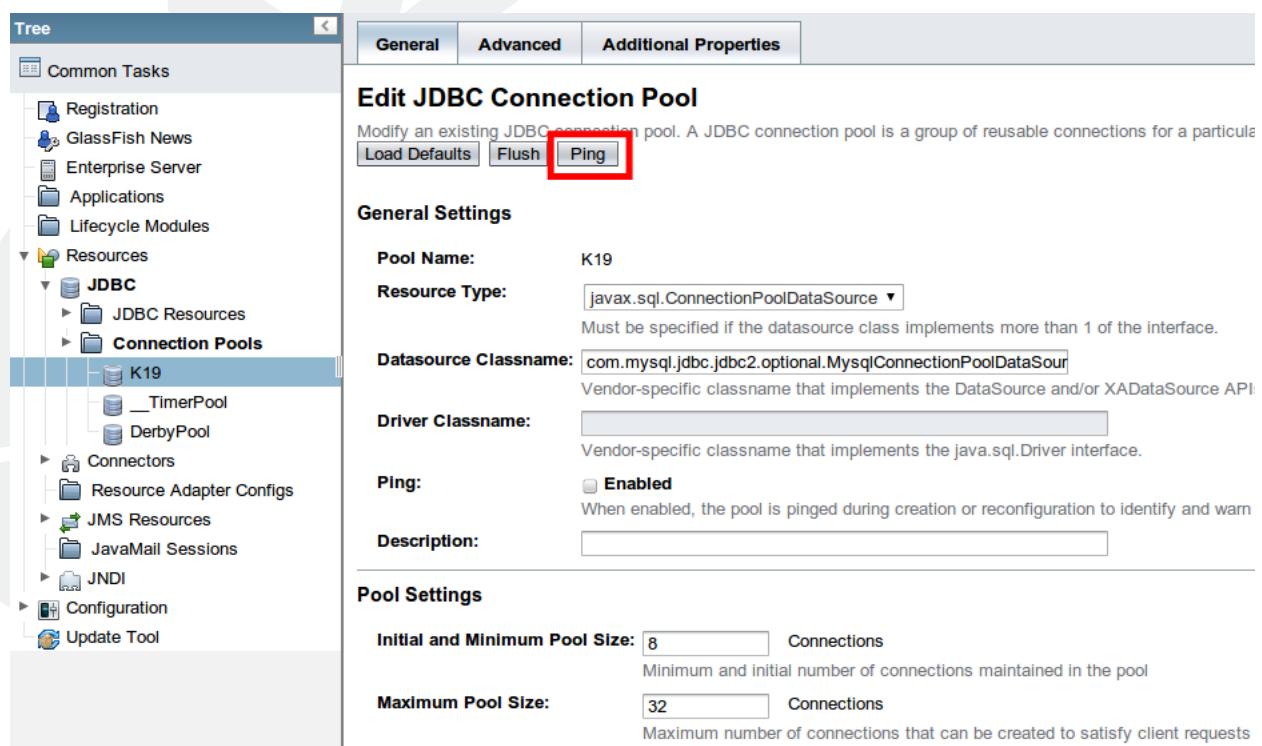
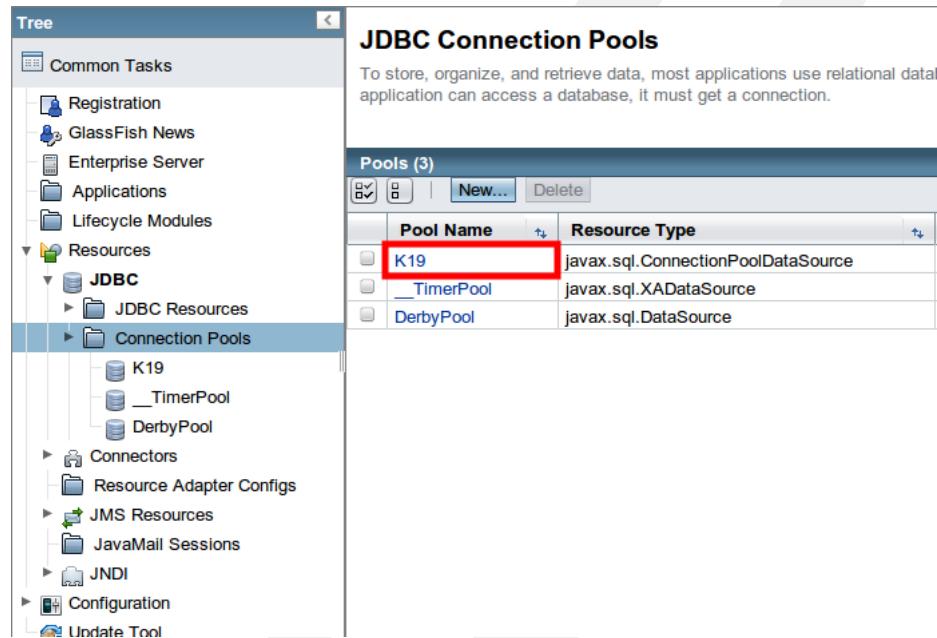
**url:** jdbc:mysql://localhost:3306/k22

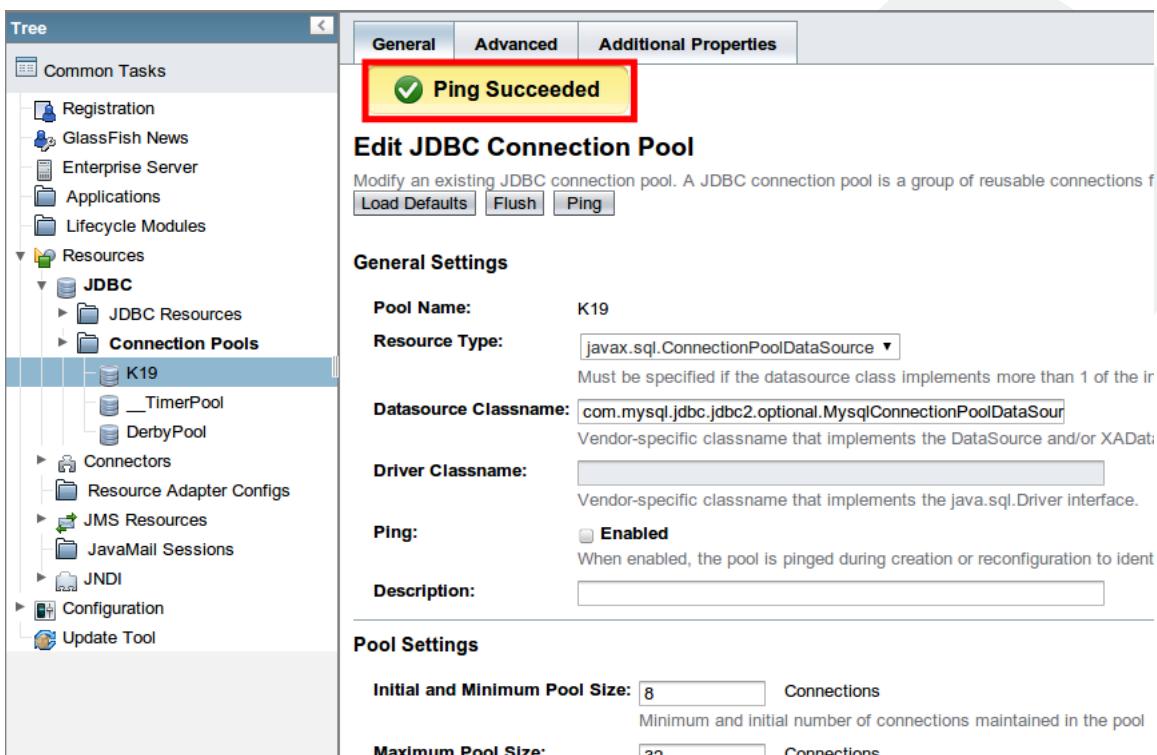
**user:** root

## Persistência

Depois, clique em **finish**.

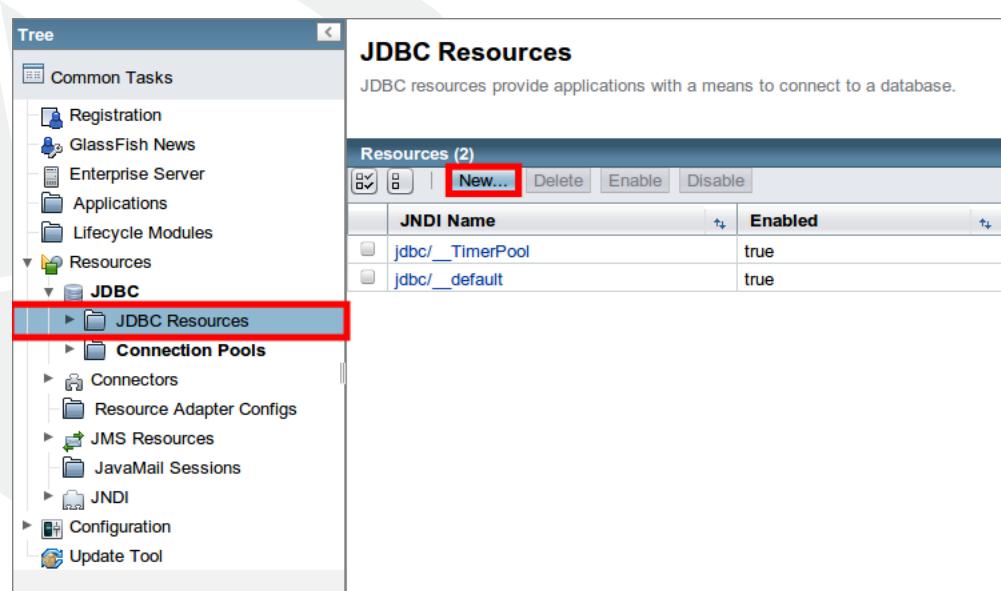
5. Para testar o Connection Pool K19, siga os passos abaixo:

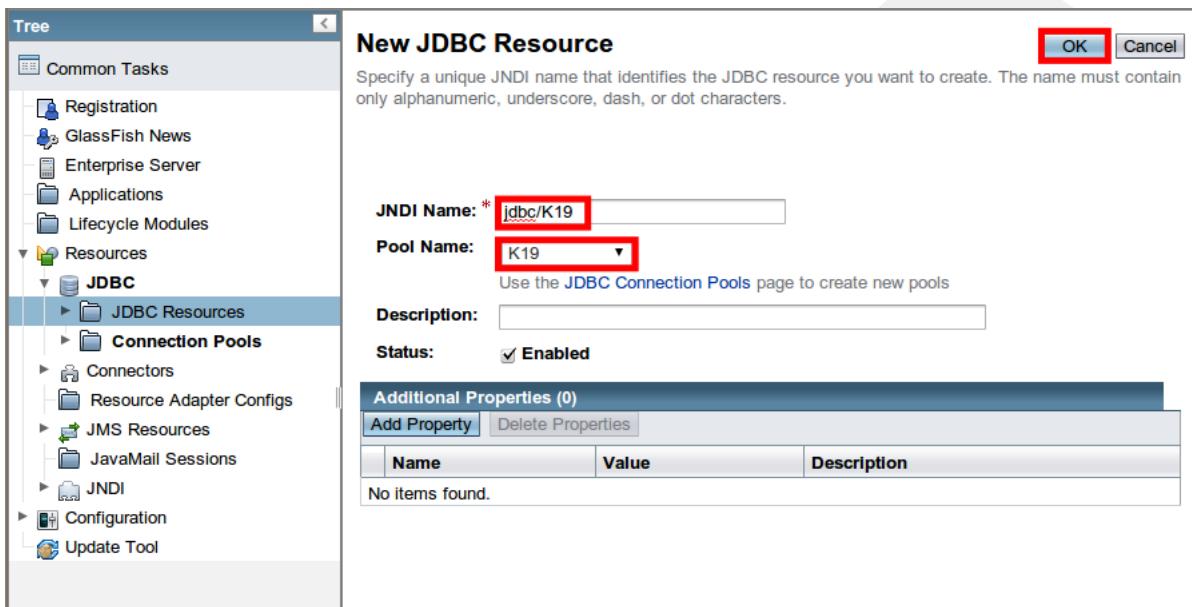




**OBS:** A mensagem “Ping Succeeded” deve aparecer.

6. Agora vamos criar o Data Source. Siga os passos abaixo:





## 5.3 persistence.xml

Em um ambiente Java EE, diversas configurações relativas à persistência são realizadas nos Data Sources. Contudo, algumas configurações ainda devem ser realizadas pelas aplicações. A especificação JPA determina que cada aplicação contenha um arquivo de configurações chamado **persistence.xml** dentro de uma pasta chamada **META-INF** no classpath da aplicação.

No arquivo `persistence.xml` podemos definir qual Data Source será utilizado pela aplicação.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/←
5   ns/persistence/persistence_1_0.xsd"
6   version="1.0">
7
8   <persistence-unit name="K19" transaction-type="JTA">
9     <provider>org.hibernate.ejb.HibernatePersistence</provider>
10    <jta-data-source>jdbc/K19</jta-data-source>
11
12    <properties>
13      <property name="hibernate.show_sql" value="true" />
14      <property name="hibernate.format_sql" value="true" />
15      <property name="hibernate.hbm2ddl.auto" value="update" />
16      <property name="hibernate.dialect" value="org.hibernate.dialect.←
17        MySQL5InnoDBDialect"/>
18    </properties>
19  </persistence-unit>
20</persistence>

```

## 5.4 Entity Beans

As regras de negócio de uma aplicação EJB são implementadas nos Session Beans. Por outro lado, os dados da aplicação que devem ser persistidos são armazenados em objetos cha-

mados **Entity Beans**. São exemplos de Entity Beans que poderiam formar uma aplicação:

- clientes
- produtos
- pedidos
- funcionários
- fornecedores

## 5.5 Entity Classes e Mapeamento

Os Entity Beans são definidos por classes java (Entity Classes). As Entity Classes devem ser mapeadas para tabelas no banco de dados através de anotações ou XML. As principais anotações de mapeamento são:

**@Entity** É a principal anotação do JPA. Ela que deve aparecer antes do nome de uma classe. E deve ser definida em todas as classes que terão objetos persistidos no banco de dados.

As classes anotadas com **@ENTITY** são mapeadas para tabelas. Por convenção, as tabelas possuem os mesmos nomes das classes. Mas, podemos alterar esse comportamento utilizando a anotação **@TABLE**.

Os atributos declarados em uma classe anotada com **@ENTITY** são mapeados para colunas na tabela correspondente à classe. Outra vez, por convenção, as colunas possuem os mesmos nomes dos atributos. E novamente, podemos alterar esse padrão utilizando para isso a anotação **@COLUMN**.

**@Id** Utilizada para indicar qual atributo de uma classe anotada com **@ENTITY** será mapeado para a chave primária da tabela correspondente à classe. Geralmente o atributo anotado com **@ID** é do tipo **LONG**.

**@GeneratedValue** Geralmente vem acompanhado da anotação **@ID**. Serve para indicar que o valor de um atributo que compõe uma chave primária deve ser gerado pelo banco no momento em que um novo registro é inserido.

Supondo uma aplicação que administra livros e autores. As seguintes classes são exemplos de Entity Classes mapeadas com anotações que poderiam ser utilizadas no contexto dessa aplicação:

```

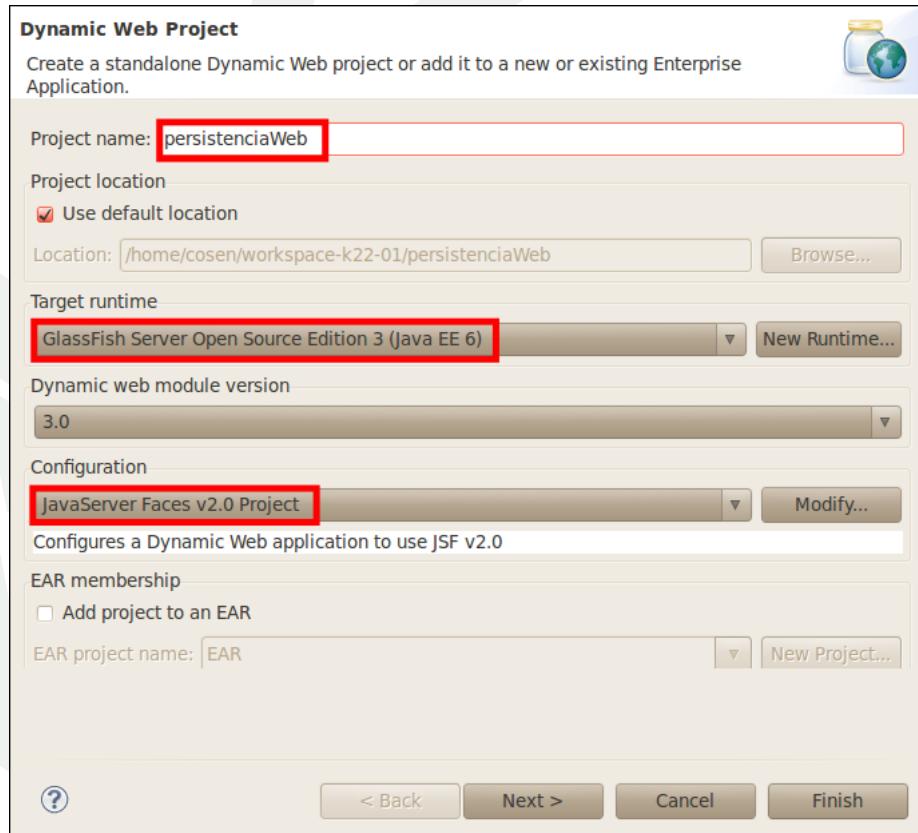
1  @Entity
2  public class Livro {
3
4      @Id @GeneratedValue
5      private Long id;
6
7      private String nome;
8
9      private Double preco;
10
11     // GETTERS AND SETTERS
12 }
```

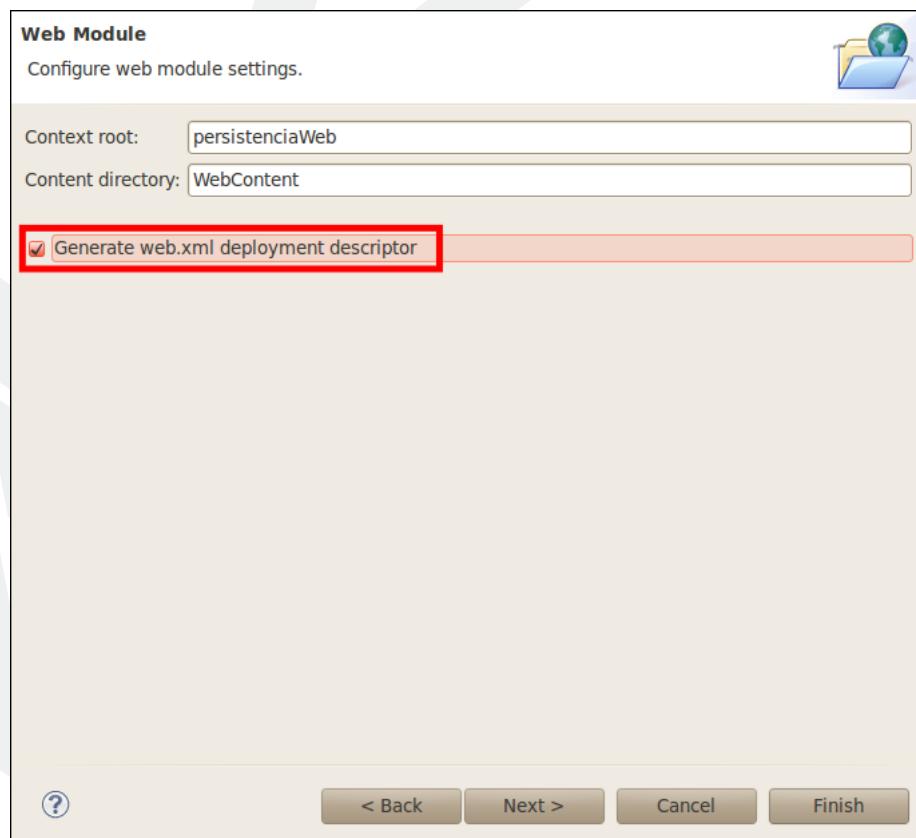
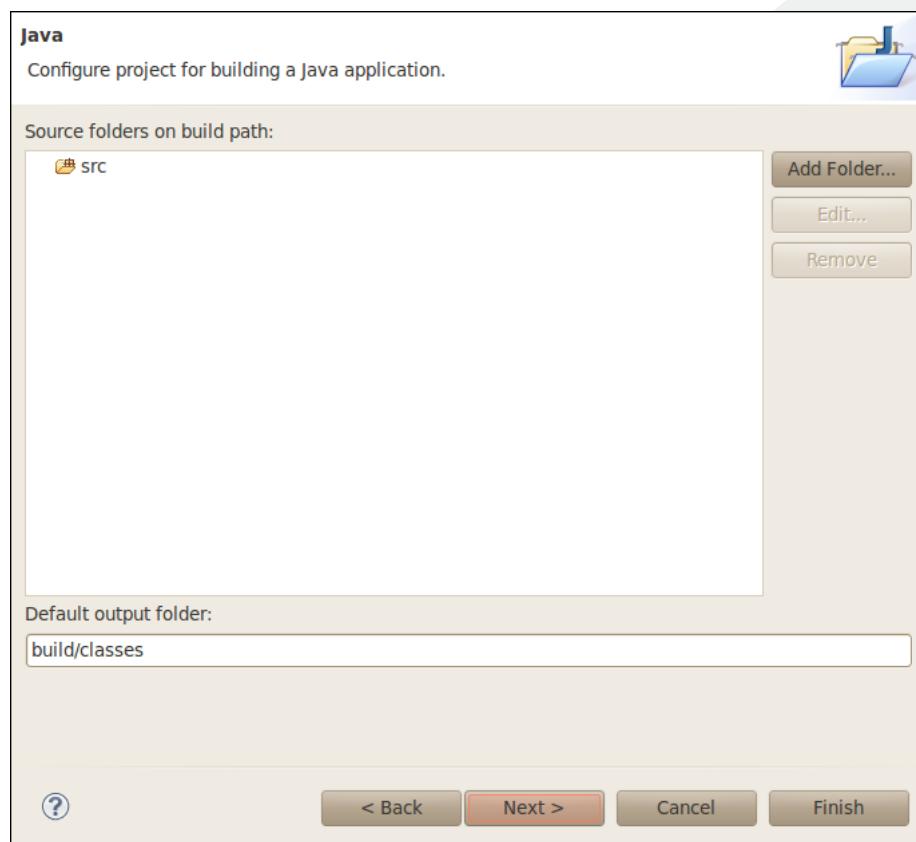
```
1 @Entity
2 public class Autor {
3
4     @Id @GeneratedValue
5     private Long id;
6
7     private String nome;
8
9     @ManyToMany
10    private List<Livro> livros;
11
12    // GETTERS AND SETTERS
13 }
```

Consulte a apostila do curso “Persistência com JPA 2” para obter detalhes sobre o mapeamento das Entity Classes <http://www.k19.com.br/downloads/apostilas-java>.

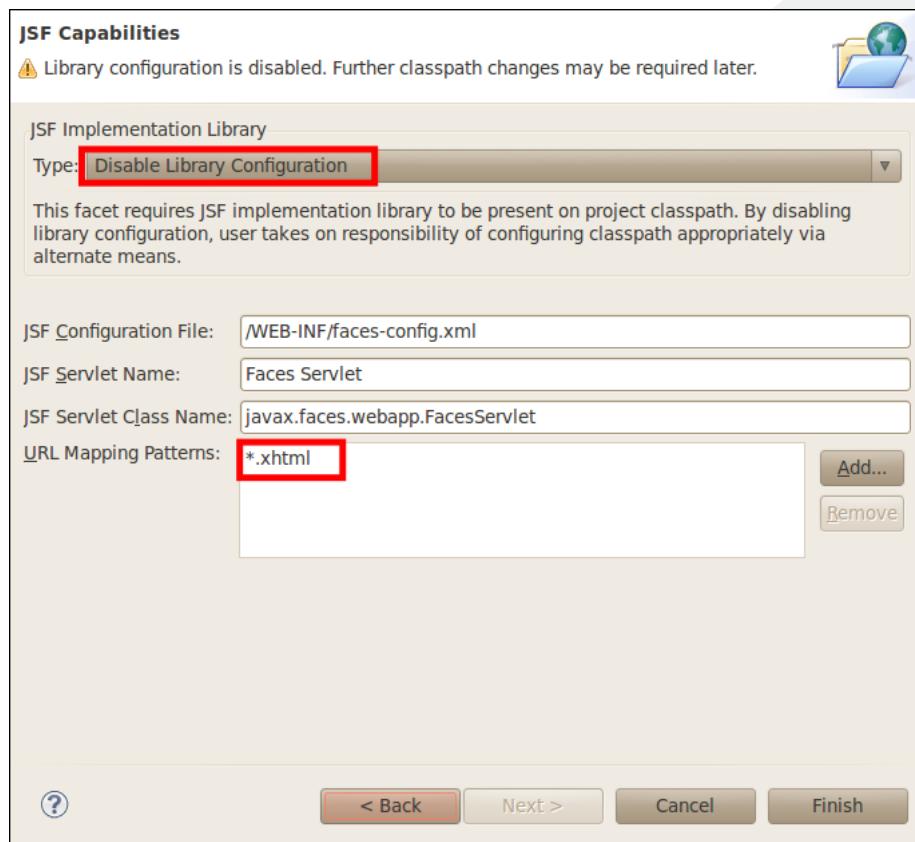
## 5.6 Exercícios

7. Crie um Dynamic Web Project no eclipse chamado **persistenciaWeb**. Você pode digitar “CTRL+3” em seguida “new Dynamic Web Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.





## Persistência



8. Adicione uma pasta chamada **META-INF** na pasta **src** do projeto **persistenciaWeb**.
9. Faça as configurações de persistência adicionando o arquivo **persistence.xml** na pasta **META-INF**.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/←
5          ns/persistence/persistence_1_0.xsd"
6      version="1.0">
7
8      <persistence-unit name="K19" transaction-type="JTA">
9          <provider>org.hibernate.ejb.HibernatePersistence</provider>
10         <jta-data-source>jdbc/K19</jta-data-source>
11
12         <properties>
13             <property name="hibernate.show_sql" value="true" />
14             <property name="hibernate.format_sql" value="true" />
15             <property name="hibernate.hbm2ddl.auto" value="update" />
16             <property name="hibernate.dialect" value="org.hibernate.dialect.←
17                 MySQL5InnoDBDialect"/>
18         </properties>
19     </persistence-unit>
20 </persistence>
```

10. Crie um pacote chamado **entidades** no projeto **persistenciaWeb** e adicione nesse pacote uma Entity Class para definir os livros de uma editora.

```

1  @Entity
2  public class Livro {
3
4      @Id @GeneratedValue
5      private Long id;
6
7      private String nome;
8
9      private Double preco;
10
11     // GETTERS AND SETTERS
12 }
```

11. Adicione no pacote **entidades** uma Entity Class para definir autores dos livros de uma editora.

```

1  @Entity
2  public class Autor {
3
4      @Id @GeneratedValue
5      private Long id;
6
7      private String nome;
8
9      @ManyToMany
10     private List<Livro> livros = new ArrayList<Livro>();
11
12     // GETTERS AND SETTERS
13 }
```

12. Adicione o projeto **persistenciaWeb** no glassfish. Clique com o botão direito no glassfish da view Servers e escolha a opção “Add and Remove”.

## 5.7 Entity Managers

Os Entity Managers são objetos que administram os Entity Beans. As principais responsabilidade dos Entity Managers são:

- Recuperar as informações armazenadas no banco de dados.
- Montar Entity Beans com os dados obtidos do banco de dados através de consultas.
- Sincronizar o conteúdo dos Entity Beans com os registros das tabelas do banco de dados.

Consulte a apostila do curso “Persistência com JPA 2” para obter detalhes sobre o funcionamento dos Entity Managers <http://www.k19.com.br/downloads/apostilas-java>.

### 5.7.1 Obtendo Entity Managers

Em um ambiente Java SE, o controle da criação e do fechamento dos Entity Managers é responsabilidade das aplicações. Uma aplicação Java SE deve chamar, explicitamente, o método **CREATEENTITYMANAGER()** em uma Entity Manager Factory para obter um novo Entity Manager ou o método **CLOSE()** em um Entity Manager para fechá-lo.

```
1 EntityManager manager = factory.createEntityManager();
```

```
1 manager.close();
```

Por outro lado, em um ambiente Java EE, o gerenciamento dos Entity Managers pode ser atribuído ao servidor de aplicação. Nesse caso, para uma aplicação Java EE obter um Entity Manager, ela pode utilizar o recurso de Injeção de Dependência oferecido pelo servidor de aplicação. Por exemplo, dentro de um Session Bean, podemos pedir a injeção de um Entity Manager através da anotação **@PersistenceContext**.

```
1 @Stateless
2 public class CalculadoraBean {
3
4     @PersistenceContext
5     private EntityManager manager;
6
7     // Resto do código
8 }
```

## 5.8 Entity Manager Factories

As Entity Managers Factories são objetos responsáveis pela criação de Entity Managers de acordo com as configurações definidas no arquivo PERSISTENCE.XML, nos Data Sources e através das anotações de mapeamento nas Entity Classes.

### 5.8.1 Obtendo Entity Manager Factories

Em um ambiente Java SE, uma Entity Manager Factory é obtida através do método estático **createEntityManagerFactory()** da classe **Persistence**.

```
1 EntityManagerFactory factory = Persistence.createEntityManagerFactory("unidade");
```

O método **createEntityManagerFactory()** deve ser chamado apenas uma vez a cada execução da aplicação. Não é necessário chamá-lo mais do que uma vez porque a aplicação não necessita de mais do que uma Entity Manager Factory e o custo de criação desse objeto é alto.

Por outro lado, em um ambiente Java EE, o controle sobre a criação das Entity Manager Factories é responsabilidade do servidor de aplicação. Inclusive, o servidor de aplicação evita a criação de fábricas desnecessárias.

Se uma aplicação Java EE deseja obter a Entity Manager Factory criada pelo servidor de aplicação, ela deve utilizar a anotação **@PersistenceUnit** para pedir a injeção desse objeto.

```
1 @Stateless
2 public class CalculadoraBean {
3
4     @PersistenceUnit
5     private EntityManagerFactory factory;
6
7     // Resto do código
8 }
```

Em geral, as aplicações Java EE não necessitam interagir diretamente com as Entity Manager Factories. Na verdade, o comum é utilizar diretamente os Entity Managers que são obtidos com a anotação **@PERSISTENCECONTEXT**.

```
1 @Stateless
2 public class CalculadoraBean {
```

```

3     @PersistenceContext
4     private EntityManager manager;
5
6     // Resto do código
7
8 }
```

## 5.9 Exercícios

13. Crie um pacote chamado **sessionbeans** no projeto **persistenciaWeb** e adicione nesse pacote um SLSB para funcionar como repositório de livros.

```

1  @Stateless
2  public class LivroRepositorio {
3
4      @PersistenceContext
5      private EntityManager manager;
6
7      public void adiciona(Livro livro) {
8          this.manager.persist(livro);
9      }
10
11     public List<Livro> getLivros() {
12         TypedQuery<Livro> query = this.manager.createQuery(
13             "select x from Livro x", Livro.class);
14
15         return query.getResultList();
16     }
17 }
```

14. Crie um pacote chamado **managedbeans** no projeto **persistenciaWeb** e adicione nesse pacote um Managed Bean para oferecer algumas ações para as telas.

```

1  @ManagedBean
2  public class LivroMB {
3
4      @EJB
5      private LivroRepositorio repositorio;
6
7      private Livro livro = new Livro();
8
9      private List<Livro> livrosCache;
10
11     public void adiciona(){
12         this.repositorio.adiciona(this.livro);
13         this.livro = new Livro();
14         this.livrosCache = null;
15     }
16
17     public List<Livro> getLivros() {
18         if(this.livrosCache == null){
19             this.livrosCache = this.repositorio.getLivros();
20         }
21         return this.livrosCache;
22     }
23
24     public Livro getLivro() {
25         return livro;
26     }
27
28     public void setLivro(Livro livro) {
29         this.livro = livro;
30     }
31 }
```

15. Crie uma tela para cadastrar livros. Adicione na pasta **WebContent** um arquivo chamado **livros.xhtml** com o seguinte conteúdo.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10   <title>Livros</title>
11 </h:head>
12
13 <h:body>
14   <h1>Novo Livro</h1>
15   <h:form>
16     <h:outputLabel value="Nome: "/>
17     <h:inputText value="#{livroMB.livro.nome}" />
18
19     <h:outputLabel value="Preço: "/>
20     <h:inputText value="#{livroMB.livro.preco}" />
21
22     <h:commandButton action="#{livroMB.adiciona}" value="Salvar"/>
23   </h:form>
24
25   <h1>Lista de Livros</h1>
26   <h:dataTable value="#{livroMB.livros}" var="livro">
27     <h:column>
28       <h:outputText value="#{livro.nome}" />
29     </h:column>
30     <h:column>
31       <h:outputText value="#{livro.preco}" />
32     </h:column>
33   </h:dataTable>
34 </h:body>
35 </html>
```

16. (Opcional) Implemente a remoção e alteração de livros.  
17. (Opcional) Implemente a adição, listagem, remoção e alteração de autores.



# Capítulo 6

## Transações

Geralmente, uma aplicação realiza diversas tarefas diferentes. Também é comum e muitas vezes necessário dividir as tarefa em pequenos passos. Daí surge o conceito de transação. Uma transação é um conjunto de passos que devem ser executados em uma ordem específica para que uma determinada tarefa seja realizada. Tipicamente, as transações modificam informações armazenadas em **resources** (bases de dados, filas de mensagens, sistemas corporativos de informação - EIS, entre outros).

### 6.1 ACID

Além da restrição natural de ordem, as transações possuem outras quatro propriedades fundamentais: Atomicidade, Consistência, Isolamento e Durabilidade. A sigla ACID é utilizada para indicar a existência dessas propriedades.

**Atomicidade:** Todos os passos de uma transação devem ser executados com sucesso para que a própria transação seja executada com sucesso. Se algum passo falhar a transação falhará e todos os passos realizados até o momento da falha serão desfeitos.

**Consistência:** Não pode existir inconsistência nos dados da aplicação nem antes nem depois da execução de uma transação. Ou seja, uma transação leva a aplicação de um estado consistente para outro estado consistente.

**Isolamento:** Alterações realizadas por uma transação não finalizada não podem afetar operações que não fazem parte da transação.

**Durabilidade:** Após a confirmação de uma transação, as modificações realizadas por ela devem ser refletidas nos resources mesmo que aconteça uma falha de hardware.

### 6.2 Transação Local ou Distribuída

Quando as alterações realizadas por uma transação afetam apenas um resource (bases de dados, filas de mensagens, sistemas corporativos de informação - EIS, entre outros), dizemos que a transação é local. Caso contrário, se dois ou mais resources são modificados por uma única transação ela é dita distribuída.

## 6.3 JTA e JTS

Todo servidor de aplicação Java EE deve oferecer suporte para as aplicações utilizarem transações. As especificações relacionadas a esse tópico são: Java Transaction API - JTA e Java Transaction Service - JTS. Os documentos dessas especificações podem ser obtidos através do site: [www.jcp.org](http://www.jcp.org).

A especificação Enterprise Java Beans (EJB) é fortemente integrada com as especificações JTA e JTS, simplificando bastante o trabalho dos desenvolvedores de aplicação EJB que não precisam em momento nenhum lidar diretamente com JTS e muito pouco com JTA.

Na arquitetura EJB, as aplicações podem gerenciar as transações de dois modos:

- Container Managed Transactions - CMT
- Bean Managed Transactions - BMT

## 6.4 Container Managed Transactions - CMT

Quando optamos pelo gerenciamento CMT, a responsabilidade de abrir, confirmar ou abortar transações é atribuída ao EJB Container. Contudo, a aplicação deve indicar ao EJB Container através de configurações quando ele deve abrir, confirmar ou abortar transações.

Podemos definir o modo CMT individualmente para cada Session Bean da nossa aplicação através da anotação **@TransactionManagement**.

```

1 @Stateful
2 @TransactionManagement(TransactionManagementType.CONTAINER)
3 public class CarrinhoBean {
4     ...
5 }
```

O modo CMT é padrão. Dessa forma, tecnicamente, não é necessário acrescentar a anotação **@TRANSACTIONMANAGEMENT**. Mas, podemos adicioná-la com o intuito de explicitar a opção de gerenciamento transacional.

### 6.4.1 Atributo Transacional

O EJB Container abre, confirma, aborta ou suspende transações de acordo com o atributo transacional de cada método dos Session Beans em modo CMT. O atributo transacional de um método pode ser definido com um dos seguintes valores: **REQUIRED**, **REQUIRES\_NEW**, **SUPPORTS**, **MANDATORY**, **NOT\_SUPPORTED** e **NEVER**.

O comportamento do EJB Container é o seguinte:

## Transações

Atributo Transacional	Já existia uma transação aberta?	O que o EJB Container faz?
REQUIRED	NÃO	Abre uma nova transação
REQUIRED	SIM	Usa a transação que já estava aberta
REQUIRES_NEW	NÃO	Abre uma nova transação
REQUIRES_NEW	SIM	Abre uma transação e Suspende a que estava aberta
SUPPORTS	NÃO	Não faz nada
SUPPORTS	SIM	Usa a transação que já estava aberta
MANDATORY	NÃO	Lança EJBTransactionRequiredException
MANDATORY	SIM	Usa a transação que já estava aberta
NOT_SUPPORTED	NÃO	Não faz nada
NOT_SUPPORTED	SIM	Suspende a que estava aberta
NEVER	NÃO	Não faz nada
NEVER	SIM	Lança EJBException

O atributo transacional de um método pode ser definido pela anotação **@TransactionAttribute**.

```
1 @TransactionAttribute(TransactionAttributeType.REQUIRED)
2 public void adiciona(String produto){
3     ...
4 }
```

Quando queremos que todos os métodos de um Session Bean possuam o mesmo atributo transacional, devemos anotar a classe com **@TRANSACTIONATTRIBUTE**.

```
1 @Stateful
2 @TransactionManagement(TransactionManagementType.CONTAINER)
3 @TransactionAttribute(TransactionAttributeType.REQUIRED)
4 public class CarrinhoBean {
5     ...
6 }
```

Caso nenhum atributo transacional seja definido explicitamente, o EJB Container utilizará por padrão o **REQUIRED**.

### 6.4.2 Rollback com SessionContext

Quando algum erro é identificado pela aplicação, ela pode marcar a transação corrente para rollback através do **SETROLLBACKONLY()** do Session Context que pode ser obtido através de injeção com a anotação **@RESOURCE**.

```

1  @Stateful
2  public class CarrinhoBean {
3
4      @Resource
5      private SessionContext context;
6
7      public void adiciona(String produto) {
8          if(produto == null){
9              context.setRollbackOnly();
10         }
11         ...
12     }
13 }
```

### 6.4.3 Rollback com Exceptions

Quando exceptions ocorrem, transações podem ser abortadas pelo EJB Container. Devemos entender a classificação das exceptions para saber quando as transações serão abortadas.

Na arquitetura EJB, as exceptions são classificadas em dois grupos:

**System Exceptions:** Todas **Unchecked Exceptions** e as **JAVA.RMI.REMOTEEXCEPTION**, por padrão, são consideradas System Exceptions.

**Application Exceptions:** Todas **Checked Exceptions** exceto as **JAVA.RMI.REMOTEEXCEPTION**, por padrão, são consideradas Application Exceptions.

Por padrão, quando um método de um Session Bean lança uma System Exception, o EJB Container aborta a transação corrente. Por outro lado, quando uma Application Exception é lançada, o EJB Container não aborta a transação corrente.

Podemos utilizar a anotação **@ApplicationException** para alterar a classificação de uma System Exception.

```

1  @ApplicationException
2  public class ValorNegativoException extends RuntimeException {
3
4 }
```

A mesma anotação pode alterar o comportamento padrão para rollback das Application Exceptions.

```

1  @ApplicationException(rollback=true)
2  public class ValorNegativoException extends RuntimeException {
3
4 }
```

## 6.5 Bean Managed Transactions - BMT

Quando optamos pelo gerenciamento BMT, a responsabilidade de abrir, confirmar ou abortar transações é atribuída a aplicação. Podemos definir o modo BMT individualmente para cada Session Bean da nossa aplicação através da anotação **@TransactionManagement**.

## *Transações*

---

```
1 @Stateful
2 @TransactionManagement(TransactionManagementType.BEAN)
3 public class CarrinhoBean {
4     ...
5 }
```

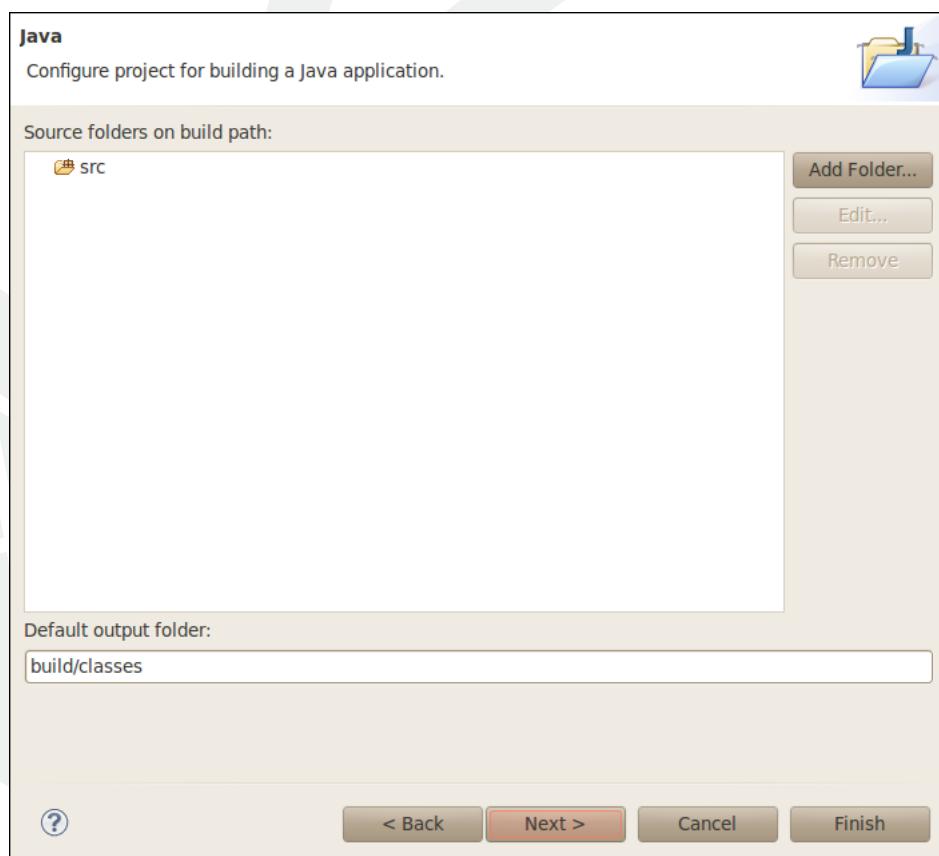
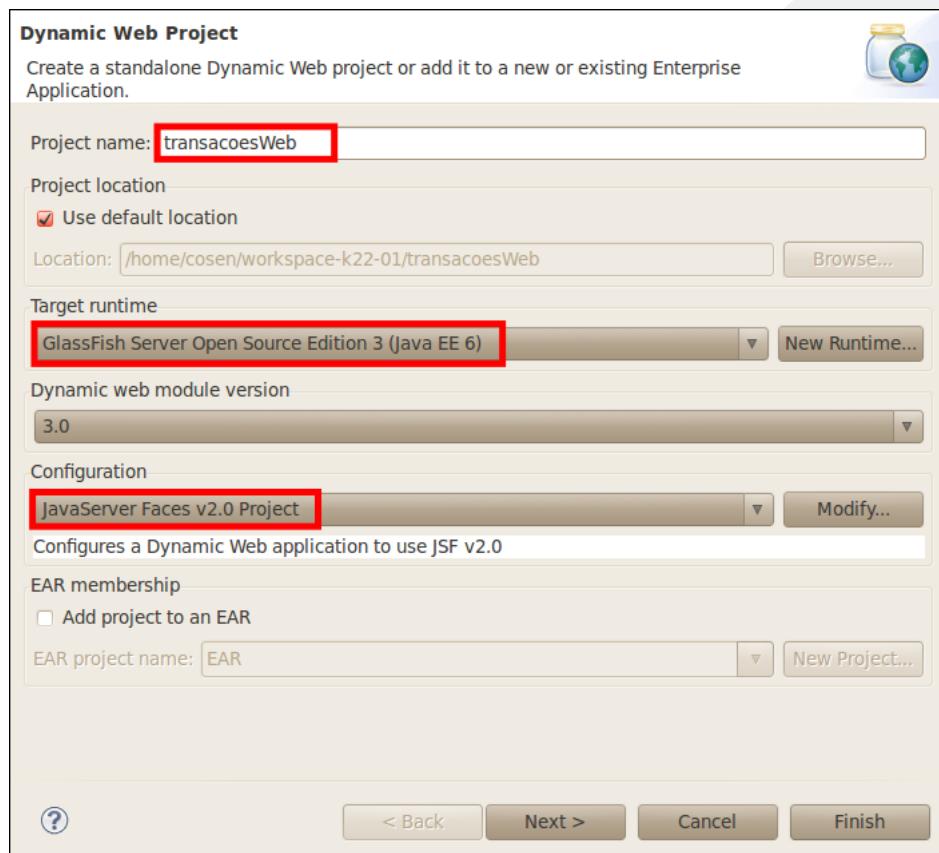
No modo BMT, devemos injetar um **UserTransaction** através da anotação **@RESOURCE**. Esse objeto permite que a aplicação abra, confirme ou aborte transações.

```
1 @Stateful
2 @TransactionManagement(TransactionManagementType.BEAN)
3 public class CarrinhoBean {
4     @Resource
5     private UserTransaction ut;
6
7     public void adiciona(Produto p) {
8         try {
9             ut.begin();
10            // IMPLEMENTACAO
11            ut.commit();
12        } catch(ProdutoInvalidoException e) {
13            ut.rollback();
14        } catch (Exception e) {
15            e.printStackTrace();
16        }
17    }
18 }
```

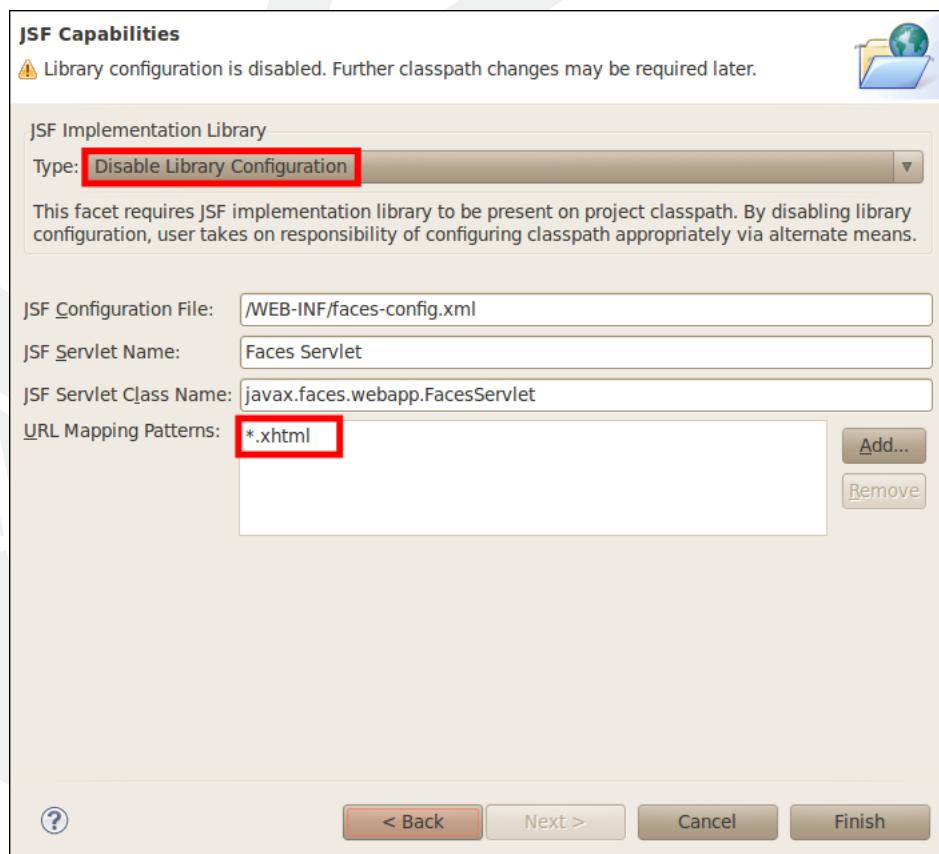
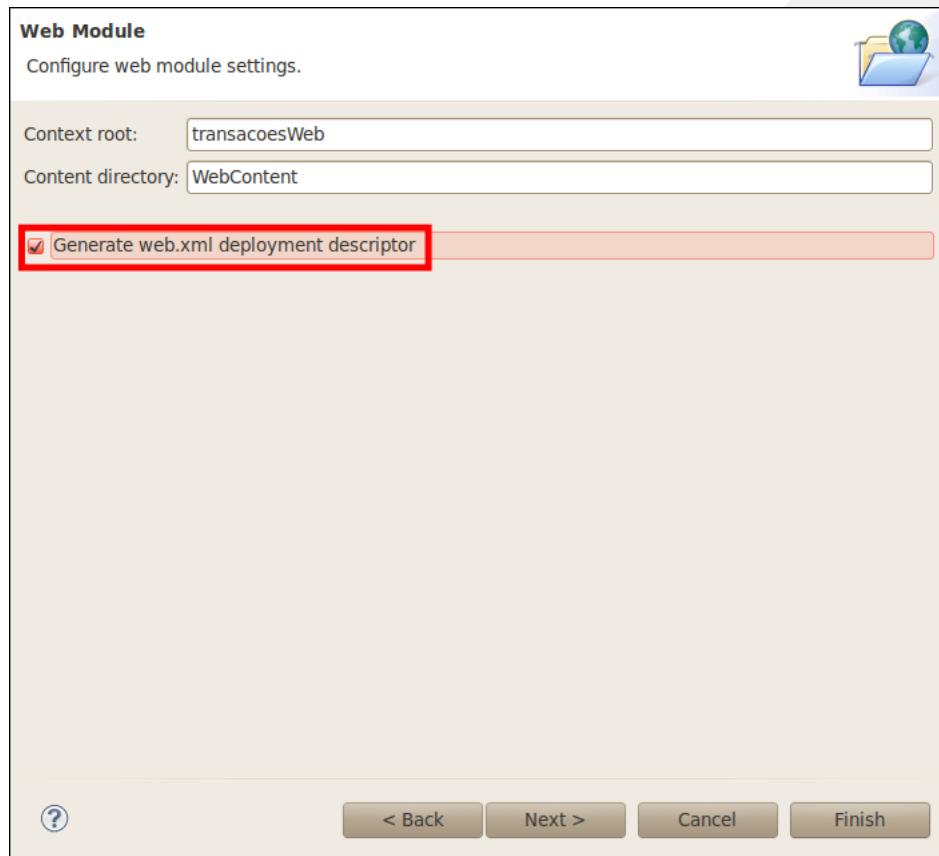
O modo BMT permite um controle maior sobre as transações. Contudo, o modo CMT é mais simples de utilizar e mais fácil de manter.

## **6.6 Exercícios**

1. Crie um Dynamic Web Project no eclipse chamado **transacoesWeb**. Você pode digitar “CTRL+3” em seguida “new Dynamic Web Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.



## Transações



2. Adicione uma pasta chamada **META-INF** na pasta **src** do projeto **transacoesWeb**.
3. Faça as configurações de persistência adicionando o arquivo **persistence.xml** na pasta **META-INF**.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/↔
      ns/persistence/persistence_1_0.xsd"
5   version="1.0">
6
7   <persistence-unit name="K19" transaction-type="JTA">
8     <provider>org.hibernate.ejb.HibernatePersistence</provider>
9     <jta-data-source>jdbc/K19</jta-data-source>
10
11    <properties>
12      <property name="hibernate.show_sql" value="true" />
13      <property name="hibernate.format_sql" value="true" />
14      <property name="hibernate.hbm2ddl.auto" value="update" />
15      <property name="hibernate.dialect" value="org.hibernate.dialect.↔
          MySQL5InnoDBDialect"/>
16    </properties>
17  </persistence-unit>
18</persistence>

```

4. Crie um pacote chamado **entidades** no projeto **transacoesWeb** e adicione nesse pacote uma Entity Class para definir os produtos de uma loja.

```

1 @Entity
2 public class Produto {
3
4   @Id @GeneratedValue
5   private Long id;
6
7   private String nome;
8
9   private double preco;
10
11  // GETTERS AND SETTERS
12 }

```

5. Crie um pacote chamado **sessionbeans** no projeto **transacoesWeb** e adicione nesse pacote um SLSB para funcionar como repositório de produtos.

```

1 @Stateless
2 public class ProdutoRepositorio {
3
4   @PersistenceContext
5   private EntityManager manager;
6
7   @Resource
8   private SessionContext context;
9
10  public void adiciona(Produto produto) {
11    this.manager.persist(produto);
12    if(produto.getPreco() < 0){
13      this.context.setRollbackOnly();
14    }
15  }
16
17  public List<Produto> getProdutos() {
18    TypedQuery<Produto> query = this.manager.createQuery(

```

## Transações

---

```
19         "select x from Produto x", Produto.class);
20
21     return query.getResultList();
22 }
23 }
```

6. Crie um pacote chamado **managedbeans** no projeto **transacoesWeb** e adicione nesse pacote um Managed Bean para oferecer algumas ações para as telas.

```
1 @ManagedBean
2 public class ProdutoMB {
3
4     @EJB
5     private ProdutoRepositorio repositorio;
6
7     private Produto produto = new Produto();
8
9     private List<Produto> produtosCache;
10
11    public void adiciona(){
12        this.repositorio.adiciona(this.produto);
13        this.produto = new Produto();
14        this.produtosCache = null;
15    }
16
17    public List<Produto> getProdutos(){
18        if(this.produtosCache == null){
19            this.produtosCache = this.repositorio.getProdutos();
20        }
21        return this.produtosCache;
22    }
23
24    public void setProduto(Produto produto) {
25        this.produto = produto;
26    }
27
28    public Produto getProduto() {
29        return produto;
30    }
31 }
```

7. Adicione o projeto **persistenciaWeb** no glassfish. Clique com o botão direito no glassfish da view Servers e escolha a opção “Add and Remove”.

8. Crie uma tela para adicionar produtos.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10   <title>Produtos</title>
11 </h:head>
12
13 <h:body>
14   <h1>Novo Produto</h1>
15   <h:form>
16     <h:outputLabel value="Nome: "/>
17     <h:inputText value="#{produtoMB.produto.nome}" />
18
19     <h:outputLabel value="Preço: "/>
20     <h:inputText value="#{produtoMB.produto.preco}" />
21
22     <h:commandButton action="#{produtoMB.adiciona}" value="Salvar"/>
23   </h:form>
24
25   <h1>Lista de Produtos</h1>
26   <h:dataTable value="#{produtoMB.produtos}" var="produto">
27     <h:column>
28       <h:outputText value="#{produto.nome}" />
29     </h:column>
30     <h:column>
31       <h:outputText value="#{produto.preco}" />
32     </h:column>
33   </h:dataTable>
34 </h:body>
35 </html>
```

9. Adicione alguns produtos e observe que produtos com preço negativo não são persistidos devido ao rollback.

# Capítulo 7

## Segurança

Para muitas aplicações, a segurança é um aspecto obrigatório. Da segurança podemos extrair dois processos fundamentais: **Autenticação e Autorização**.

O processo de autenticação consiste na identificação dos usuários através de algum tipo de certificado (usuário e senha). Já o processo de autorização determina o que cada usuário autenticado pode acessar dentro da aplicação.

Na plataforma Java, esses dois processos são padronizados pela especificação JAAS (Java Authentication and Authorization Service).

### 7.1 Realms

Em um ambiente Java EE, para realizar o processo de autenticação, devemos criar um ou mais **Realms**. Um Realm é uma base de dados na qual os usuários de uma ou mais aplicações estão cadastrados.

Infelizmente, as configurações necessárias para criar um Realm não são padronizadas, ou seja, cada servidor de aplicação as realiza da sua própria maneira. Veremos no exercício como criar um Realm no Glassfish.

### 7.2 Exercícios

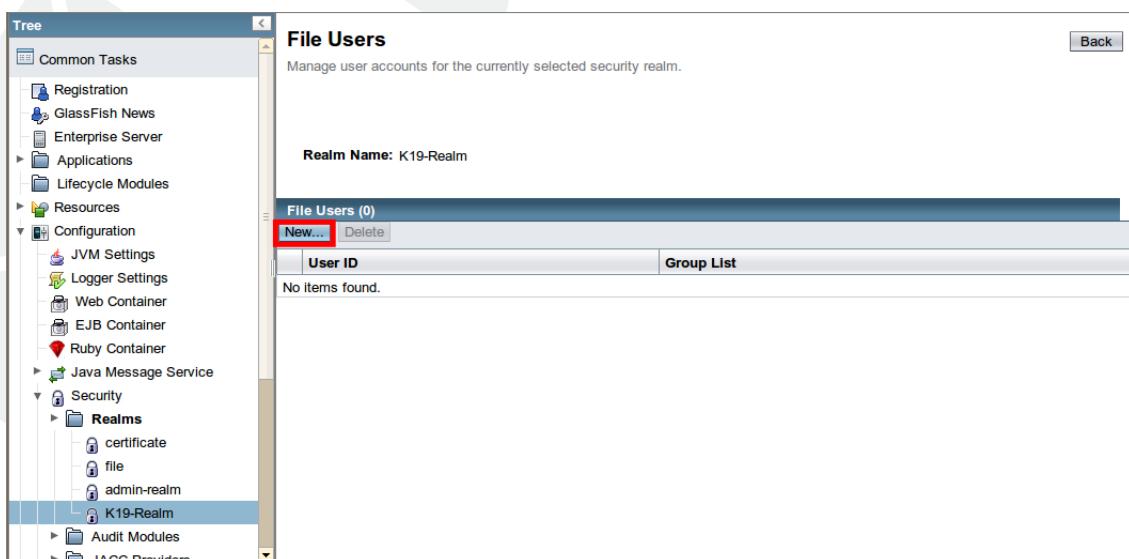
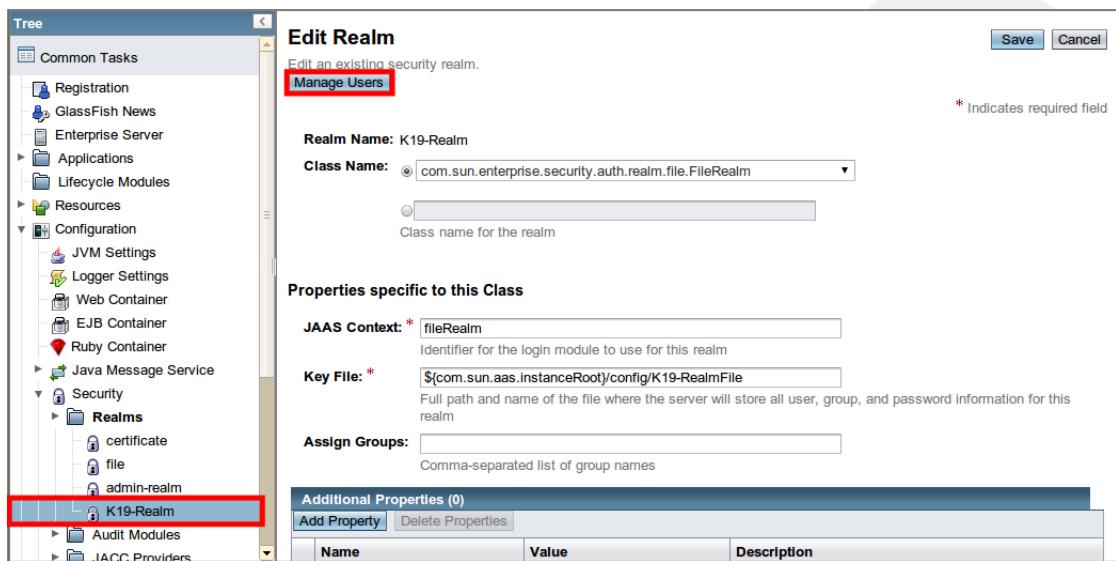
1. Com o Glassfish inicializado, abra a interface de administração acessando a url `localhost:4848`. Siga os passos abaixo:

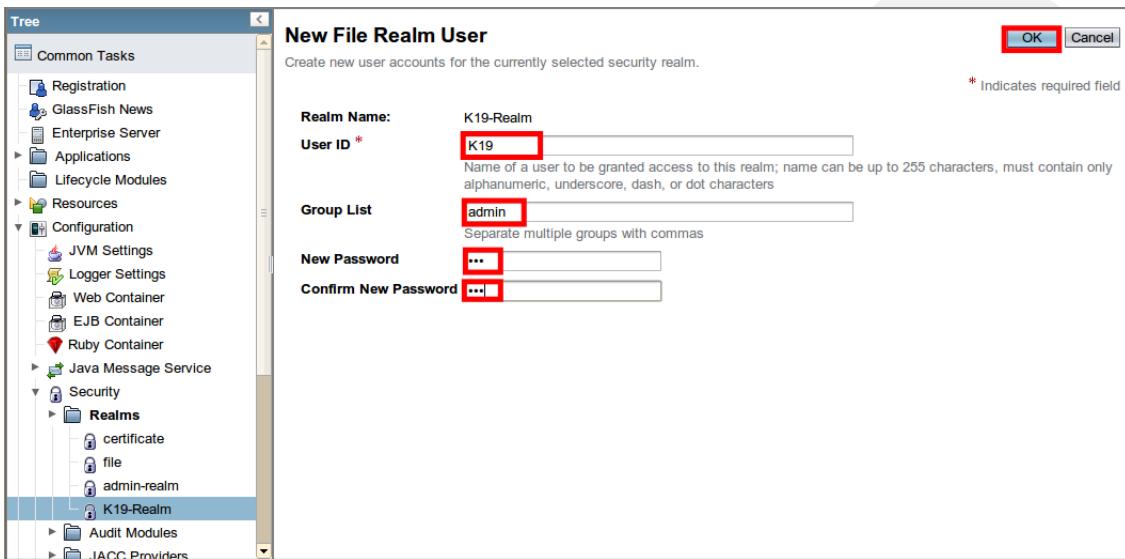
The screenshot shows two windows from the GlassFish Admin Console:

- Realms (3) Window:** Shows a list of existing realms: certificate (Class Name: com.sun.enterprise.security.auth.realm.certificate.CertificateRealm), file (Class Name: com.sun.enterprise.security.auth.realm.file.FileRealm), and admin-realm (Class Name: com.sun.enterprise.security.auth.realm.file.FileRealm). A red box highlights the "Realms" node in the tree on the left.
- New Realm Window:** A dialog for creating a new security realm. It includes fields for "Name" (K19-Realm), "Class Name" (com.sun.enterprise.security.auth.realm.file.FileRealm), "JAAS Context" (fileRealm), and "Key File" (S:/com.sun.aas.instanceRoot/config/K19-RealmFile). A red box highlights the "OK" button.

- Adicione um usuário chamado **K19** dentro de um grupo chamado **admin** com a senha **K19**. Siga os passos abaixo:

## Segurança





3. Repita o processo do exercício anterior e cadastre os seguintes usuários:

Usuário	Grupo	Senha
keizo	admin	keizo
afk	users	afk

## 7.3 Autenticação - Aplicações Web

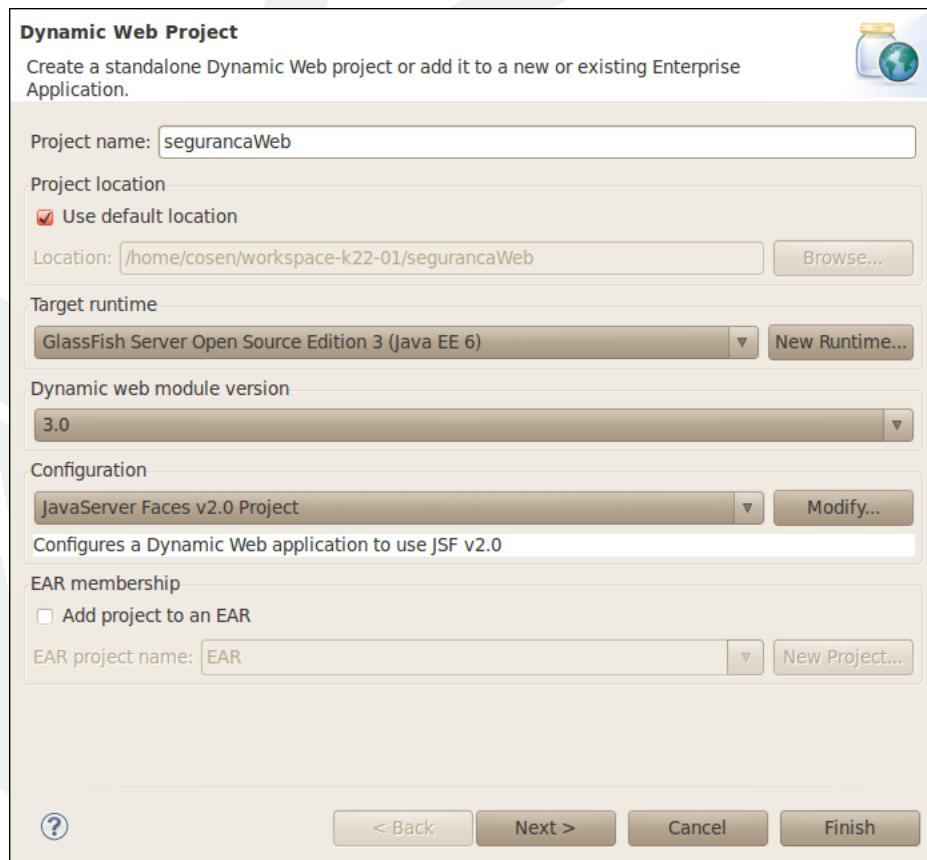
Geralmente, o processo de autenticação é realizado na camada web. Portanto, vamos restringir a nossa discussão a esse tipo de aplicação.

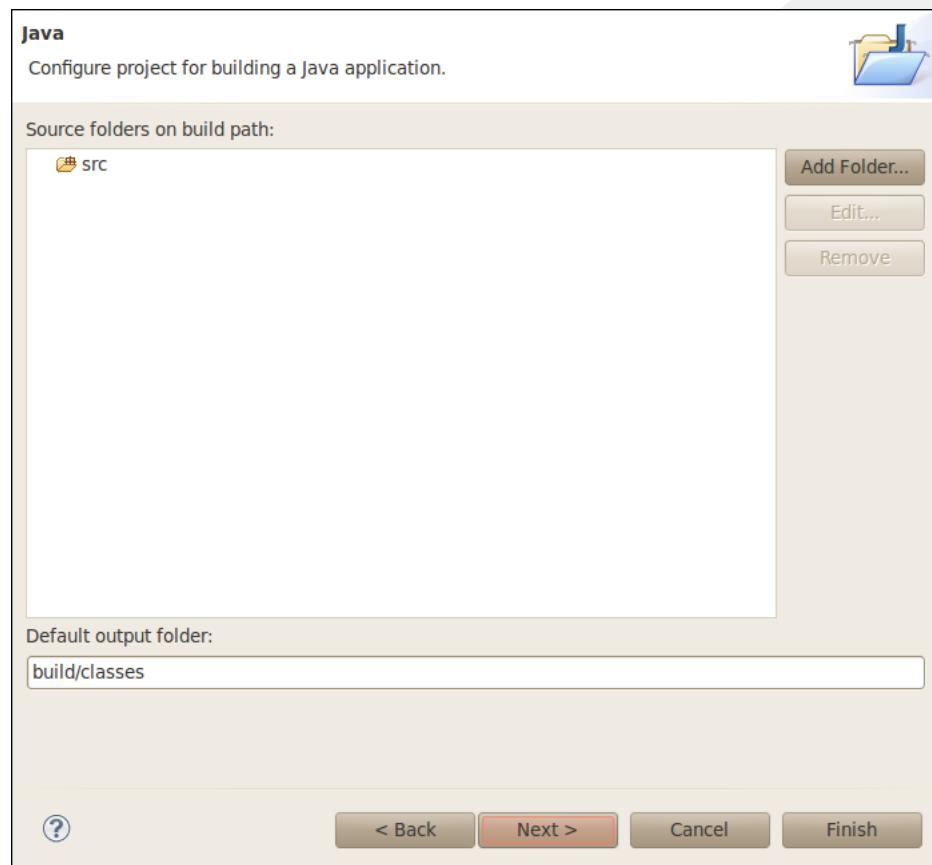
A maior parte das configurações referentes ao processo de autenticação que as aplicações web devem realizar são definidas no arquivo **web.xml**. Contudo, alguns servidores de aplicação

exigem configurações extras. Veremos no exercício como configurar uma aplicação web no Glassfish para realizar o processo de autenticação.

## 7.4 Exercícios

4. Crie um Dynamic Web Project no eclipse chamado **segurancaWeb**. Você pode digitar “CTRL+3” em seguida “new Dynamic Web Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.





## Segurança

---

**Web Module**

Configure web module settings.

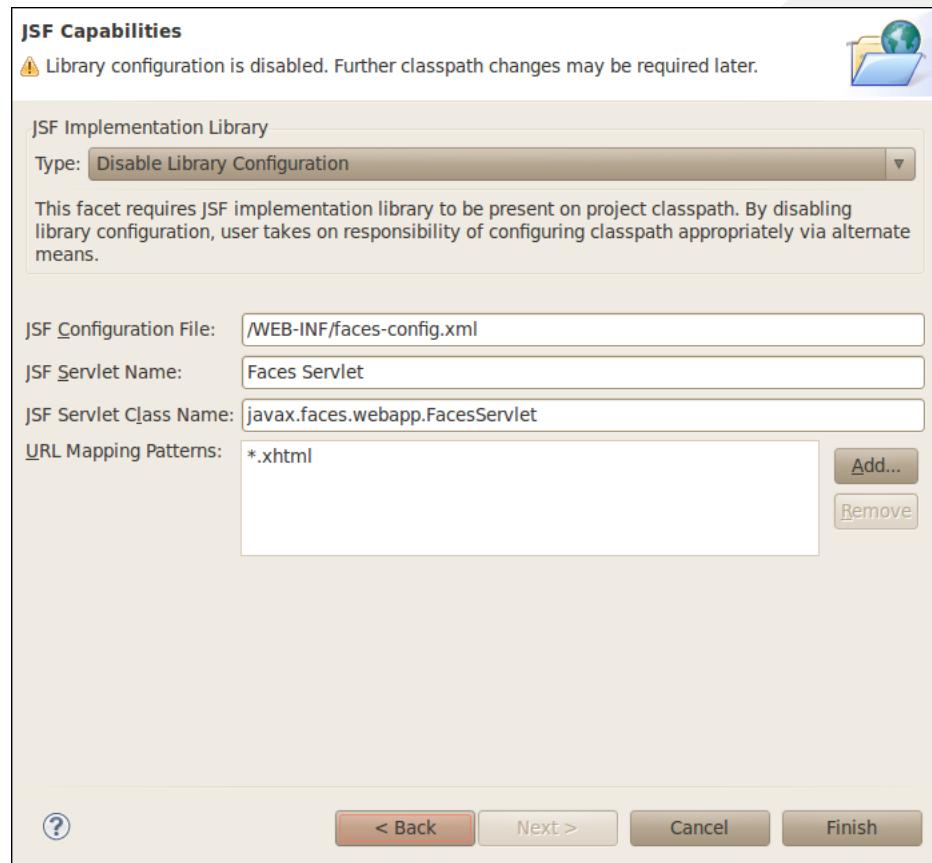
Context root:

Content directory:

Generate web.xml deployment descriptor

?

< Back    Next >    Cancel    Finish



5. Acrescente no arquivo **sun-web.xml** do projeto **segurancaWeb** o seguinte trecho de código logo após o elemento CONTEXT-ROOT.

```

1 <security-role-mapping>
2   <role-name>ADMIN</role-name>
3   <group-name>admin</group-name>
4 </security-role-mapping>
5
6 <security-role-mapping>
7   <role-name>USERS</role-name>
8   <group-name>users</group-name>
9 </security-role-mapping>
```

**Essa configuração é específica do Glassfish.**

**Os grupos são utilizados pelo Glassfish e os roles pela aplicação.**

6. Acrescente no arquivo **web.xml** do projeto **segurancaWeb** o seguinte trecho de código dentro do elemento WEB-APP.

## Segurança

---

```
1 <login-config>
2   <auth-method>BASIC</auth-method>
3   <realm-name>K19-Realm</realm-name>
4 </login-config>
5
6 <security-constraint>
7   <web-resource-collection>
8     <web-resource-name>resources</web-resource-name>
9     <url-pattern>/*</url-pattern>
10    <http-method>GET</http-method>
11    <http-method>POST</http-method>
12  </web-resource-collection>
13  <auth-constraint>
14    <role-name>ADMIN</role-name>
15    <role-name>USERS</role-name>
16  </auth-constraint>
17 </security-constraint>
```

7. Adicione o arquivo **index.xhtml** na pasta **WebContent** do projeto **segurancaWeb** com seguinte conteúdo.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10   <title>Segurança</title>
11 </h:head>
12
13 <h:body>
14   <h1>Você está autenticado!</h1>
15 </h:body>
16 </html>
```

8. Adicione o projeto **segurancaWeb** no glassfish. Clique com o botão direito no glassfish da view Servers e escolha a opção “Add and Remove”.
9. Acesse através de um navegador a url <http://localhost:8080/segurancaWeb/index.xhtml> e teste o processo de autenticação.

## 7.5 Autorização - Aplicações EJB

Podemos limitar o acesso dos usuários aos métodos de um Session Bean. Por exemplo, é possível declarar que um determinado método de um Session Bean só pode ser chamado por usuários administradores ou moderadores.

### 7.5.1 @RolesAllowed

Restrições de acesso podem ser definidas pela anotação **@RolesAllowed** que pode ser aplicada na classe ou nos métodos de um Session Bean. Se aplicada na classe valerá para todos

os métodos. Se aplicada ao mesmo tempo na classe e em algum método, valerá as restrições definidas no método.

```
1 @RolesAllowed({"administrador", "moderador"})
2 public void adiciona(Produto produto) {
3     this.manager.persist(produto);
4 }
```

```
1 @RolesAllowed({"administrador", "moderador"})
2 @Stateful
3 class CarrinhoBean {
4     ...
5 }
```

## 7.5.2 @PermitAll

Podemos utilizar a anotação **@PermitAll** para permitir que qualquer tipo de usuário tenha acesso. Para conseguir o mesmo efeito com a anotação **@RolesAllowed**, teríamos que listar todos os Roles. Além disso, caso um Role fosse criado ou destruído, alterações seriam necessárias.

```
1 @PermitAll
2 public void adiciona(Produto produto) {
3     this.manager.persist(produto);
4 }
```

```
1 @PermitAll
2 @Stateful
3 class CarrinhoBean {
4     ...
5 }
```

## 7.5.3 @DenyAll

O funcionamento da anotação **@DenyAll** é exatamente o oposto da **@PERMITALL**. Podemos utilizar a anotação **@DENYALL** em aplicações que são implantadas em ambientes diferentes. Sendo que em determinados ambientes certas funcionalidades devem ser desabilitadas.

```
1 @DenyAll
2 public void adiciona(Produto produto) {
3     this.manager.persist(produto);
4 }
```

```
1 @DenyAll
2 @Stateful
3 class CarrinhoBean {
4     ...
5 }
```

## 7.5.4 @RunAs

Eventualmente, um Session Bean chama outro Session Bean. Suponha, que os métodos do primeiro possam ser executados por usuários moderadores e os métodos do segundo por administradores. Para que o primeiro Session Bean possa chamar o Segundo, temos que definir o papel de administrador para o primeiro Session Bean através da anotação **@RunAs**.

## Segurança

---

```
1  @Stateless
2  @RunAs("administrador")
3  class MensagemRepositorio {
4
5      @PersistenceContext
6      private EntityManager manager;
7
8      @EJB
9      private TopicoRepositorio topicoRepositorio;
10
11     @RolesAllowed({"moderador"})
12     public void remove(Long id) {
13         Mensagem m = this.manager.find(Mensagem.class, id);
14         this.manager.remove(m);
15
16         Topico t = m.getTopico();
17
18         if(t.getMensagens().size() == 1) {
19             this.topicoRepositorio.remove(t);
20         }
21     }
22 }
```

## 7.6 Exercícios

10. Crie um pacote chamado **sessionbeans** no projeto **segurancaWeb** e adicione nesse pacote um Singleton Session Bean.

```
1  @Singleton
2  public class TarefasBean {
3
4      private List<String> tarefas = new ArrayList<String>();
5
6      @RolesAllowed({"ADMIN", "USERS"})
7      public void adiciona(String tarefa) {
8          this.tarefas.add(tarefa);
9      }
10
11     @RolesAllowed({"ADMIN", "USERS"})
12     public List<String> listaTarefas() {
13         return this.tarefas;
14     }
15
16     @RolesAllowed({"ADMIN"})
17     public void remove(String tarefa) {
18         this.tarefas.remove(tarefa);
19     }
20 }
```

11. Crie um pacote chamado **managedbeans** no projeto **segurancaWeb** e adicione nesse pacote um Managed Bean.

```
1  @ManagedBean
2  public class TarefasMB {
3
4      @EJB
5      private TarefasBean tarefasBean;
6
7      private String tarefa;
8
9      public void adiciona(){
10         this.tarefasBean.adiciona(this.tarefa);
11     }
12 }
```

```

12
13     public void remove(String tarefa) {
14         this.tarefasBean.remove(tarefa);
15     }
16
17     public List<String> getTarefas() {
18         return this.tarefasBean.listaTarefas();
19     }
20
21     // GETTERS AND SETTERS
22 }
```

**12.** Altere o arquivo **index.xhtml** do projeto **segurancaWeb**.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9  <h:head>
10    <title>Segurança</title>
11  </h:head>
12
13  <h:body>
14    <h1>Você está autenticado!</h1>
15    <h1>Nova Tarefa</h1>
16    <h:form>
17      <h:outputLabel value="Tarefa: " />
18      <h:inputText value="#{tarefasMB.tarefa}" />
19
20      <h:commandButton action="#{tarefasMB.adiciona}" value="Salvar" />
21
22      <h1>Lista de Tarefas</h1>
23      <h:dataTable value="#{tarefasMB.tarefas}" var="tarefa">
24        <h:column>
25          <h:outputText value="#{tarefa}" />
26        </h:column>
27        <h:column>
28          <h:commandLink action="#{tarefasMB.remove(tarefa)}" >remove</h:commandLink>
29        </h:column>
30      </h:dataTable>
31    </h:form>
32  </h:body>
33 </html>
```

**13.** Adicione e remova tarefas utilizando todos os usuários cadastrados. Observe que usuários com Role USERS não conseguem remover tarefas.

# Capítulo 8

## Interceptadores

Uma aplicação EJB pode definir, através de métodos de callback, lógicas a serem executadas pelo EJB Container quando uma instância de um Session Bean muda de estado.

- O método de callback “PostConstruct” é executado quando uma instância de um Session Bean de qualquer tipo muda do estado NÃO EXISTE para o PRONTO.
- O método de callback “PreDestroy” é executado quando uma instância de um Session Bean muda do estado PRONTO para o NÃO EXISTE.
- O método de callback “PrePassivate” é executado quando uma instância de um Stateful Session Bean muda do estado PRONTO para o PASSIVADO.
- O método de callback “PostActivate” é executado quando uma instância de um Stateful Session Bean muda do estado PASSIVADO para o PRONTO.

Há um método de callback para cada uma das seguintes transições: NÃO EXISTE->PRONTO, PRONTO->PASSIVADO, PASSIVADO->PRONTO e PRONTO->NÃO EXISTE. Além dessas transições, podemos considerar que toda vez que um método de negócio é chamado ocorre a transição PRONTO->PRONTO. Não há método de callback para essa transição especial. Contudo, na arquitetura EJB, podemos utilizar a ideia de interceptadores para conseguir executar lógicas antes ou depois da execução de um método de negócio.

É comum utilizar interceptadores para tarefas que não estão diretamente relacionadas às regras de negócio implementadas nos Session Beans. Por exemplo, podemos implementar logging ou controle de acesso com interceptadores.

### 8.1 Interceptor Methods

A lógica de um interceptador é definida dentro de um método anotado com **@AroundInvoke** ou registrado através de XML. Não há restrições em relação a visibilidade desse método, ou seja, ele pode ser público, protegido, padrão ou privado. Contudo, ele deve possuir uma assinatura compatível com o seguinte formato:

1    Object <MÉTODO>(InvocationContext) throws Exception

Veja um exemplo concreto de método interceptador:

```

1  @AroundInvoke
2  public Object interceptador(InvocationContext ic) throws Exception {
3      // IDA
4      System.out.println("ANTES DO MÉTODO DE NEGÓCIO");
5
6      // CHAMANDO O MÉTODO DE NEGÓCIO E PEGANDO O SEU RETORNO
7      Object retornoDoMetodoDeNegocio = ic.proceed();
8
9      // VOLTA
10     System.out.println("DEPOIS DO MÉTODO DE NEGÓCIO");
11     return retornoDoMetodoDeNegocio;
12 }
```

## 8.2 Internal Interceptors

Um interceptador interno é criado quando um método interceptador é definido dentro de um Session Bean. Cada Session Bean pode ter no máximo um interceptador interno. Um interceptador interno atua em todos os métodos de negócio do seu respectivo Session Bean.

```

1  @Stateless
2  class CalculadoraBean {
3
4      // MÉTODOS DE NEGÓCIO
5
6      // MÉTODOS DE CALLBACK
7
8      // MÉTODO INTERCEPTADOR
9      @AroundInvoke
10     public Object interceptador(InvocationContext ic) throws Exception {
11         // IDA
12         System.out.println("ANTES DO MÉTODO DE NEGÓCIO");
13
14         // CHAMANDO O MÉTODO DE NEGÓCIO E PEGANDO O SEU RETORNO
15         Object retornoDoMetodoDeNegocio = ic.proceed();
16
17         // VOLTA
18         System.out.println("DEPOIS DO MÉTODO DE NEGÓCIO");
19         return retornoDoMetodoDeNegocio;
20     }
21 }
```

## 8.3 External Interceptors

Os interceptadores externos são criados quando um método interceptador é definido fora de um Session Bean em uma classe comum. Novamente, não mais do que um método interceptador pode ser definido em uma mesma classe.

```

1  class LoggingInterceptor {
2
3      @AroundInvoke
4      public Object interceptador(InvocationContext ic) throws Exception {
5          // IDA
6          System.out.println("ANTES DO MÉTODO DE NEGÓCIO");
7
8          // CHAMANDO O MÉTODO DE NEGÓCIO E PEGANDO O SEU RETORNO
9          Object retornoDoMetodoDeNegocio = ic.proceed();
10
11         // VOLTA
12         System.out.println("DEPOIS DO MÉTODO DE NEGÓCIO");
```

## Interceptadores

---

```
13     return retornoDoMetodoDeNegocio;
14 }
15 }
```

### 8.3.1 Method-Level Interceptors

Interceptadores externos podem ser associados a métodos de negócio através da anotação **@Interceptors**.

```
1 @Stateless
2 class CalculadoraBean {
3
4     @Interceptors({LoggingInterceptor.class})
5     public double soma(double a, double b){
6         return a + b;
7     }
8 }
```

Vários interceptadores externos podem ser associados a um método de negócio através da anotação **@INTERCEPTORS**.

```
1 @Stateless
2 class CalculadoraBean {
3
4     @Interceptors({LoggingInterceptor.class, SegurancaInterceptor.class})
5     public double soma(double a, double b){
6         return a + b;
7     }
8 }
```

### 8.3.2 Class-Level Interceptors

Interceptadores externos também podem ser associados a Session Beans através da anotação **@Interceptors**. Quando associado a um Session Bean, um interceptador externo será aplicado a todos os métodos de negócio desse Session Bean.

```
1 @Stateless
2 @Interceptors({LoggingInterceptor.class})
3 class CalculadoraBean {
4
5     public double soma(double a, double b){
6         return a + b;
7     }
8 }
```

Vários interceptadores externos podem ser associados a um Session Bean através da anotação **@INTERCEPTORS**.

```
1 @Stateless
2 @Interceptors({LoggingInterceptor.class, SegurancaInterceptor.class})
3 class CalculadoraBean {
4
5     public double soma(double a, double b){
6         return a + b;
7     }
8 }
```

### 8.3.3 Default Interceptors

Interceptadores externos também podem ser associados a métodos de negócio através de configurações adicionadas no arquivo de configuração do EJB, o **ejb-jar.xml**. Esse arquivo deve ser colocado em uma pasta chamada **META-INF** dentro do módulo EJB da aplicação.

Por exemplo, suponha que o interceptador externo definido pela classe **LOGGINGINTERCEPTOR** tenha que ser aplicado em todos os métodos de negócio de todos os Session Beans.

```

1 <interceptor-binding>
2   <ejb-name>*</ejb-name>
3   <interceptor-class>interceptadores.LoggingInterceptor</interceptor-class>
4 </interceptor-binding>
```

Ou podemos aplicar a apenas um Session Bean.

```

1 <interceptor-binding>
2   <ejb-name>sessionbeans.CalculadoraBean</ejb-name>
3   <interceptor-class>INTERCEPTOR</interceptor-class>
4 </interceptor-binding>
```

### 8.4 Excluindo Interceptadores

Podemos excluir os Default Interceptors e os Class-Level Interceptors através das anotações **@ExcludeDefaultInterceptors** e **@ExcludeClassInterceptors** respectivamente.

A anotação **@EXCLUDEDEFAULTINTERCEPTORS** pode ser aplicada em um método de negócio ou no Session Bean.

```

1 @Stateless
2 @ExcludeDefaultInterceptors
3 class CalculadoraBean {
4
5   public double soma(double a, double b) {
6     return a + b;
7   }
8 }
```

A anotação **@EXCLUDECLASSINTERCEPTORS** pode ser aplicada em um método de negócio.

```

1 @Stateless
2 @Interceptors({LoggingInterceptor.class, SegurancaInterceptor.class})
3 class CalculadoraBean {
4
5   @ExcludeClassInterceptors
6   public double soma(double a, double b) {
7     return a + b;
8   }
9 }
```

## 8.5 Invocation Context

Um método interceptador recebe um Invocation Context como parâmetro. Através dos Invocation Context, os métodos interceptadores podem acessar a instância do Session Bean que será utilizada para atender a chamada, descobrir qual método de negócio será executado, quais parâmetros foram passados e até mesmo trocar os parâmetros antes de chegar no método de negócio. Veja os métodos disponíveis nessa interface.

```
1 public interface InvocationContext {  
2     public Object getTarget();  
3     public Method getMethod();  
4     public Object[] getParameters();  
5     public void setParameters(Object[]);  
6     public java.util.Map<String, Object> getContextData();  
7     public Object proceed() throws Exception;  
8 }
```

## 8.6 Ordem dos Interceptadores

A ordem na qual os interceptadores são executados pode afetar o funcionamento da aplicação. A especificação dos interceptadores define a seguinte ordem:

1. Default Interceptors
2. Class-Level Interceptors
3. Method-Level Interceptors
4. Internal Interceptors

Quando dois ou mais Default Interceptors estão associados a um método de negócio, eles serão executados na ordem em que foram definidos no **ejb-jar.xml**.

Quando dois ou mais Class-Level Interceptors estão associados a um método de negócio, eles serão executados na ordem em que foram declarados na anotação **@Interceptors**.

Quando dois ou mais Method-Level Interceptors estão associados a um método de negócio, eles serão executados na ordem em que foram declarados na anotação **@Interceptors**.

## 8.7 Exercícios

1. Crie um Dynamic Web Project no eclipse chamado **interceptadoresWeb**. Você pode digitar “CTRL+3” em seguida “new Dynamic Web Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.

**Dynamic Web Project**

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: interceptadoresWeb

Project location

Use default location

Location: /home/cosen/workspace-k22-01/interceptadoresWeb

Target runtime

GlassFish Server Open Source Edition 3 (Java EE 6)

Dynamic web module version

3.0

Configuration

JavaServer Faces v2.0 Project

Configures a Dynamic Web application to use JSF v2.0

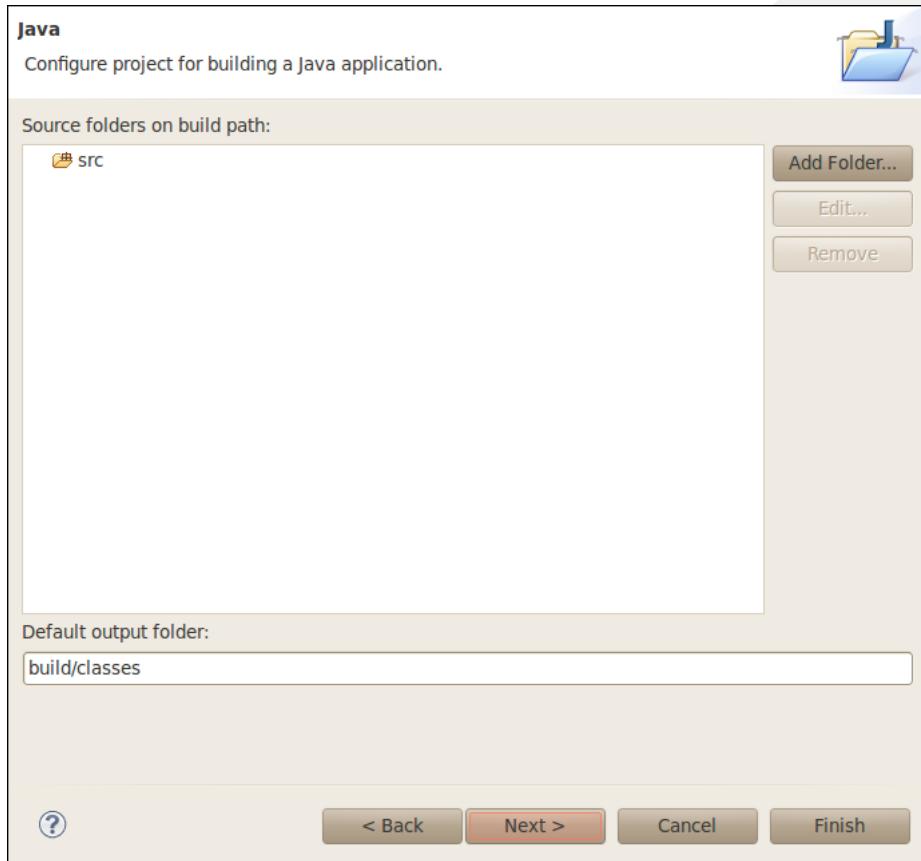
EAR membership

Add project to an EAR

EAR project name: EAR

## *Interceptadores*

---

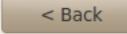
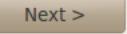
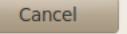
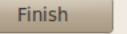


**Web Module**  
Configure web module settings.

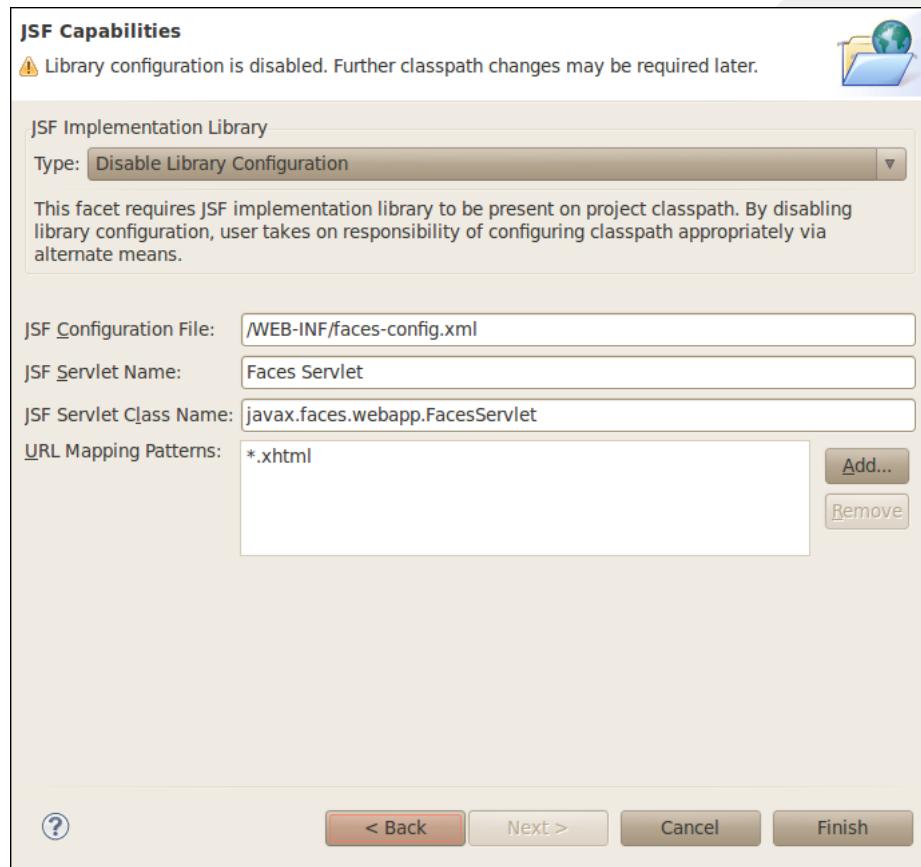
Context root:  

Content directory:

Generate web.xml deployment descriptor

## Interceptadores



- Configure a aplicação **interceptadoresWeb** para utilizar o data source **jdbc/K19** criado no capítulo 5, adicionando o arquivo **persistence.xml** na pasta **META-INF** dentro do **src** do projeto **interceptadoresWeb**.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/←
5          ns/persistence/persistence_1_0.xsd"
6      version="1.0">
7
8      <persistence-unit name="K19" transaction-type="JTA">
9          <provider>org.hibernate.ejb.HibernatePersistence</provider>
10         <jta-data-source>jdbc/K19</jta-data-source>
11
12         <properties>
13             <property name="hibernate.hbm2ddl.auto" value="update" />
14             <property name="hibernate.dialect" value="org.hibernate.dialect.←
15                 MySQL5InnoDBDialect"/>
16         </properties>
17     </persistence-unit>
18 </persistence>
```

- Crie um pacote chamado **entidades** no projeto **interceptadoresWeb** e adicione nesse pacote um Entity Bean para modelar mensagens.

```
1  @Entity
2  public class Mensagem {
```

```

3     @Id @GeneratedValue
4     private Long id;
5
6     private String texto;
7
8     // GETTERS AND SETTERS
9
10 }
```

4. Crie um pacote chamado **sessionbeans** no projeto **interceptadoresWeb** e adicione nesse pacote um SLSB para funcionar como repositório de mensagens.

```

1  @Stateless
2  public class MensagemRepositorio {
3
4      @PersistenceContext
5      private EntityManager manager;
6
7      public void adiciona(Mensagem mensagem) {
8          this.manager.persist(mensagem);
9      }
10
11     public List<Mensagem> getMensagens() {
12         TypedQuery<Mensagem> query = this.manager.createQuery(
13             "select x from Mensagem x", Mensagem.class);
14
15         return query.getResultList();
16     }
17 }
```

5. Crie um pacote chamado **managedbeans** no projeto **interceptadoresWeb** e adicione nesse pacote um Managed Bean para oferecer algumas ações para as telas.

```

1  @ManagedBean
2  public class MensagemMB {
3
4      @EJB
5      private MensagemRepositorio repositorio;
6
7      private Mensagem mensagem = new Mensagem();
8
9      private List<Mensagem> mensagensCache;
10
11     public void adiciona(){
12         this.repositorio.adiciona(this.mensagem);
13         this.mensagem = new Mensagem();
14         this.mensagensCache = null;
15     }
16
17     public List<Mensagem> getMensagens(){
18         if(this.mensagensCache == null){
19             this.mensagensCache = this.repositorio.getMensagens();
20         }
21         return this.mensagensCache;
22     }
23
24     public Mensagem getMensagem() {
25         return mensagem;
26     }
27
28     public void setMensagem(Mensagem mensagem) {
29         this.mensagem = mensagem;
30     }
31 }
```

## Interceptadores

---

6. Crie uma tela para cadastrar mensagens. Adicione na pasta **WebContent** do projeto **interceptadoresWeb** um arquivo chamado **mensagens.xhtml** com o seguinte conteúdo.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10   <title>Mensagens</title>
11 </h:head>
12
13 <h:body>
14   <h1>Nova Mensagem</h1>
15   <h:form>
16     <h:outputLabel value="Mensagem: "/>
17     <h:inputText area value="#{mensagemMB.mensagem.texto}" />
18
19     <h:commandButton action="#{mensagemMB.adiciona}" value="Salvar"/>
20   </h:form>
21
22   <h1>Lista de Mensagens</h1>
23   <h:dataTable value="#{mensagemMB.mensagens}" var="mensagem">
24     <h:column>
25       <h:outputText value="#{mensagem.id}" />
26     </h:column>
27     <h:column>
28       <h:outputText value="#{mensagem.texto}" />
29     </h:column>
30   </h:dataTable>
31 </h:body>
32 </html>
```

7. Implemente um interceptador externo para realizar o logging da aplicação. Crie um pacote chamado **interceptadores** no projeto **interceptadoresWeb** e adicione nesse pacote a seguinte classe.

```
1 public class LoggingInterceptor {
2
3   @AroundInvoke
4   public Object interceptador(InvocationContext ic) throws Exception {
5     System.out.println("CHAMANDO O MÉTODO: " + ic.getMethod());
6
7     Object retornoDoMetodoDeNegocio = ic.proceed();
8
9     System.out.println("MÉTODO " + ic.getMethod() + " FINALIZADO");
10    return retornoDoMetodoDeNegocio;
11  }
12}
```

8. Adicione na pasta **WEB-INF** do projeto **interceptadoresWeb** o arquivo **ejb-jar.xml** com o seguinte conteúdo.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/←
5     javaee/ejb-jar_3_1.xsd">
6
7   <interceptors>
8     <interceptor>
9       <interceptor-class>interceptadores.LoggingInterceptor</interceptor-class>
10    </interceptor>
11  </interceptors>
12
13  <assembly-descriptor>
14    <interceptor-binding>
15      <ejb-name>*</ejb-name>
16      <interceptor-class>interceptadores.LoggingInterceptor</interceptor-class>
17    </interceptor-binding>
18  </assembly-descriptor>
19 </ejb-jar>

```

9. Acesse a url <http://localhost:8080/interceptadoresWeb/mensagens.xhtml> e depois observe as mensagens do interceptador no console do eclipse.
10. Implemente um interceptador externo para eliminar palavras proibidas das mensagens adicionas o logging da aplicação. Adicione no pacote **interceptadores** do projeto **interceptadoresWeb** a seguinte classe.

```

1 public class CensuraInterceptor {
2
3   private List<String> palavrasProibidas = new ArrayList<String>();
4
5   public CensuraInterceptor() {
6     this.palavrasProibidas.add("coca-cola");
7     this.palavrasProibidas.add("fiat");
8     this.palavrasProibidas.add("sony");
9   }
10
11 @AroundInvoke
12 public Object interceptador(InvocationContext ic) throws Exception {
13   Object[] parameters = ic.getParameters();
14   Mensagem mensagem = (Mensagem)parameters[0];
15
16   for (String palavraProibida : this.palavrasProibidas) {
17     String textoOriginal = mensagem.getTexto();
18     String textoCensurado = textoOriginal.replaceAll(palavraProibida, "!!←
19           CENSURADO!!");
20     mensagem.setTexto(textoCensurado);
21   }
22   return ic.proceed();
23 }

```

11. Associe o interceptador de censura ao método de adicionar mensagens do Session Bean **MensagemRepositorio**.

```

1 @Interceptors({CensuraInterceptor.class})
2 public void adiciona(Mensagem mensagem) {
3   this.manager.persist(mensagem);
4 }

```

12. Adicione mensagens com palavras proibidas através da url <http://localhost:8080/interceptadoresWeb/mensagens.xhtml>.

## *Interceptadores*

---

13. Implemente um interceptador externo para calcular o tempo gasto para executar cada método de negócio. Associe esse interceptador a todos os métodos de negócio da aplicação.



# Capítulo 9

## Scheduling

Algumas aplicações possuem a necessidade de agendar tarefas para serem executadas periodicamente ou uma única vez após um determinado tempo. Por exemplo, suponha uma aplicação que calcula o salário dos funcionários de uma empresa de acordo com as horas registradas. Possivelmente, esse cálculo deve ser realizado uma vez por mês.

Outro exemplo, suponha que uma empresa vende seus produtos através da internet. As entregas só são realizadas após a confirmação dos pagamentos. Quando um cliente realiza um pedido, o sistema da empresa deve esperar alguns dias para verificar se o pagamento correspondente foi realizado para que a entrega possa ser liberada.

### 9.1 Timers

Para agendar tarefas, podemos criar alarmes (timers) através do **TimerService**. Por exemplo, suponha que seja necessário executar uma tarefa uma única vez depois de 30 minutos.

```
1 timerService.createTimer(30 * 60 * 1000, "info");
```

O primeiro parâmetro do método **CREATETIMER()** é quantidade de tempo que ele deve esperar para “disparar” e o segundo é uma informação que podemos associar ao timer. Também, podemos criar um alarme periódico que “dispara” a cada 30 minutos através do **TIMERSERVICE** utilizando a sobrecarga do método **CREATE TIMER()**.

```
1 timerService.createTimer(30 * 60 * 1000, 30 * 60 * 1000, "info");
```

Podemos obter o **TIMERSERVICE** por injeção através da anotação **@Resource**.

```
1 @Stateless
2 class FolhaDePagamentoBean {
3
4     @Resource
5     private TimerService timerService;
6
7     ...
8 }
```

Os alarmes não podem ser criados para Stateful Session Beans. Essa funcionalidade deve ser adicionada em versões futuras da especificação Enterprise Java Beans.

## 9.2 Métodos de Timeout

Quando um alarme (timer) “dispara”, o EJB Container executa um método de timeout no Bean que criou o alarme. Para definir um método de timeout devemos utilizar a anotação **@Timeout**.

```

1  @Stateless
2  class PedidoBean {
3
4      @Resource
5      private TimerService timerService;
6
7      public void registraPedido(Pedido pedido){
8          this.timerService.createTimer(5 * 24 * 60 * 60 * 1000, pedido);
9      }
10
11     @Timeout
12     public void verificaPagamento(Timer timer){
13         Pedido pedido = (Pedido)timer.getInfo();
14         // verifica o pagamento do pedido
15     }
16 }
```

## 9.3 Timers Automáticos

Na versão 3.1 da especificação Enterprise Java Beans, os alarmes podem ser criados e automaticamente associados a métodos de timeout através da anotação **@Schedule**.

```

1  @Stateless
2  class FolhaDePagamentoBean {
3
4      @Schedule(dayOfMonth="1")
5      public void calculaSalarios(){
6          // implementacao
7      }
8 }
```

## 9.4 Exercícios

1. Crie um Dynamic Web Project no eclipse chamado **schedulingWeb**. Você pode digitar “CTRL+3” em seguida “new Dynamic Web Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.

## Scheduling

**Dynamic Web Project**  
Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name:

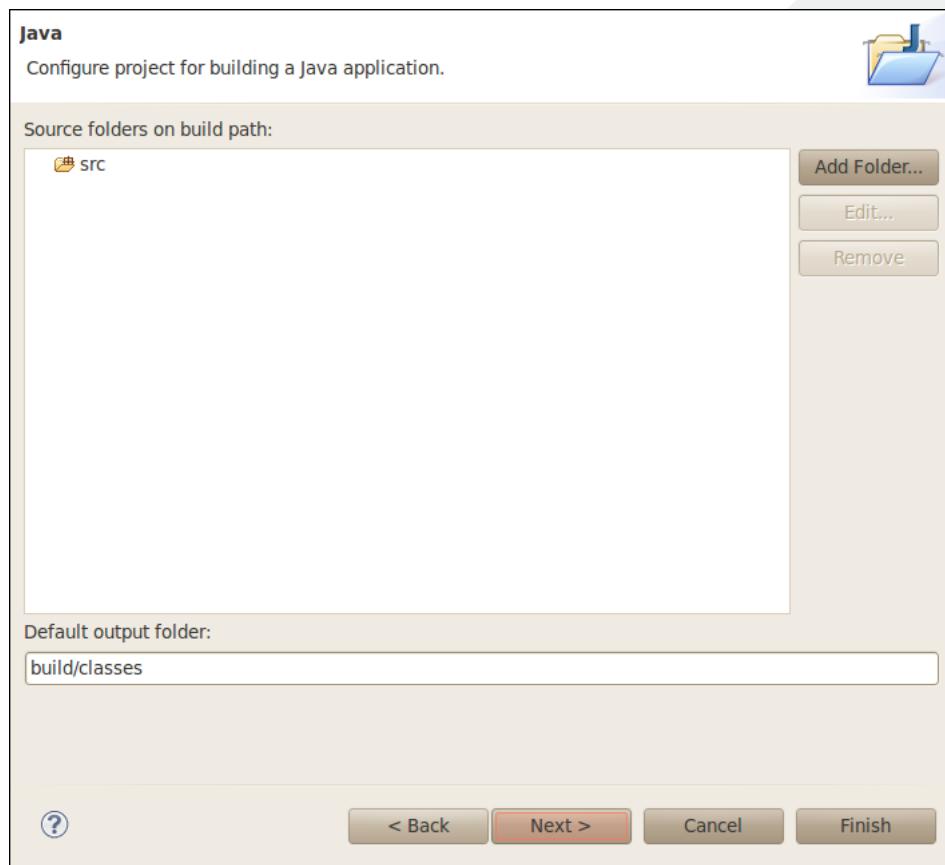
Project location  
 Use default location  
Location:

Target runtime

Dynamic web module version

Configuration  
   
Configures a Dynamic Web application to use JSF v2.0

EAR membership  
 Add project to an EAR  
EAR project name:



## Scheduling

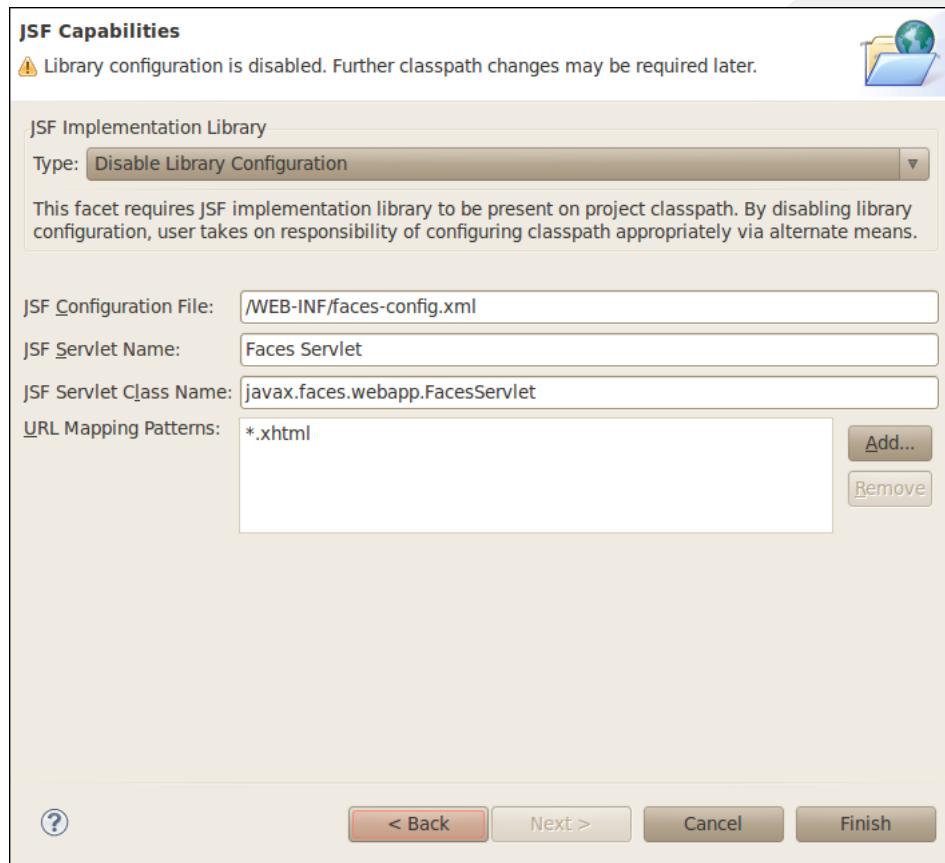
**Web Module**  
Configure web module settings.

Context root:

Content directory:

Generate web.xml deployment descriptor

[?](#)   [< Back](#)   [Next >](#)   [Cancel](#)   [Finish](#)



- Configure a aplicação **schedulingWeb** para utilizar o data source **jdbc/K19** criado no capítulo **5**, adicionando o arquivo **persistence.xml** na pasta **META-INF** dentro do **src** do projeto **schedulingWeb**.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/↔
      ns/persistence/persistence_1_0.xsd"
5   version="1.0">
6
7   <persistence-unit name="K19" transaction-type="JTA">
8     <provider>org.hibernate.ejb.HibernatePersistence</provider>
9     <jta-data-source>jdbc/K19</jta-data-source>
10
11    <properties>
12      <property name="hibernate.hbm2ddl.auto" value="update" />
13      <property name="hibernate.dialect" value="org.hibernate.dialect.↔
          MySQL5InnoDBDialect"/>
14    </properties>
15  </persistence-unit>
16 </persistence>

```

- Crie um pacote chamado **entidades** no projeto **schedulingWeb** e adicione nesse pacote um Entity Bean para modelar produtos.

```

1 @Entity
2 public class Produto {

```

## Scheduling

---

```
3  @Id @GeneratedValue
4  private Long id;
5
6  private String nome;
7
8  private double preco;
9
10 // GETTERS AND SETTERS
11
12 }
```

4. Crie um pacote chamado **sessionbeans** no projeto **schedulingWeb** e adicione nesse pacote um SLSB para funcionar como repositório de produtos.

```
1  @Stateless
2  public class ProdutoRepositorio {
3
4      @PersistenceContext
5      private EntityManager manager;
6
7      public void adiciona(Produto produto) {
8          this.manager.persist(produto);
9      }
10
11     public List<Produto> getProdutos() {
12         TypedQuery<Produto> query = this.manager.createQuery(
13             "select x from Produto x", Produto.class);
14
15         return query.getResultList();
16     }
17 }
```

5. Crie um pacote chamado **managedbeans** no projeto **schedulingWeb** e adicione nesse pacote um Managed Bean para oferecer algumas ações para as telas.

```
1  @ManagedBean
2  public class ProdutoMB {
3
4      @EJB
5      private ProdutoRepositorio repositorio;
6
7      private Produto produto = new Produto();
8
9      private List<Produto> produtosCache;
10
11     public void adiciona(){
12         this.repositorio.adiciona(this.produto);
13         this.produto = new Produto();
14         this.produtosCache = null;
15     }
16
17     public List<Produto> getProdutos(){
18         if(this.produtosCache == null){
19             this.produtosCache = this.repositorio.getProdutos();
20         }
21         return this.produtosCache;
22     }
23
24     public void setProduto(Produto produto) {
25         this.produto = produto;
26     }
27
28     public Produto getProduto() {
29         return produto;
30     }
31 }
```

6. Crie uma tela para cadastrar produtos. Adicione na pasta **WebContent** do projeto **schedulingWeb** um arquivo chamado **produtos.xhtml** com o seguinte conteúdo.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10   <title>Produtos</title>
11 </h:head>
12
13 <h:body>
14   <h1>Novo Produto</h1>
15   <h:form>
16     <h:outputLabel value="Nome: "/>
17     <h:inputText value="#{produtoMB.produto.nome}" />
18
19     <h:outputLabel value="Preço: "/>
20     <h:inputText value="#{produtoMB.produto.preco}" />
21
22     <h:commandButton action="#{produtoMB.adiciona}" value="Salvar"/>
23   </h:form>
24
25   <h1>Lista de Produtos</h1>
26   <h:dataTable value="#{produtoMB.produtos}" var="produto">
27     <h:column>
28       <h:outputText value="#{produto.nome}" />
29     </h:column>
30     <h:column>
31       <h:outputText value="#{produto.preco}" />
32     </h:column>
33   </h:dataTable>
34 </h:body>
35 </html>
```

7. Adicione produtos através da url <http://localhost:8080/schedulingWeb/produtos.xhtml>.
8. A cada 5 minutos um produto cadastrado deve ser escolhido para ser colocado em destaque. Implemente essa lógica através dos recursos de scheduling. Adicione a seguinte classe no pacote **sessionbeans** do projeto **schedulingWeb**.

```

1 @Singleton
2 public class ProdutoDestaqueBean {
3
4   @EJB
5   private ProdutoRepositorio repositorio;
6
7   private Produto produtoDestaque;
8
9   @Schedule(second="30", minute="*", hour="*")
10  public void trocaProdutoDestaque() {
11    Random gerador = new Random();
12    List<Produto> produtos = this.repositorio.getProdutos();
13    int i = gerador.nextInt(produtos.size());
14    this.produtoDestaque = produtos.get(i);
15  }
16
17  public void setProdutoDestaque(Produto produtoDestaque) {
```

## *Scheduling*

---

```
18     this.produtoDestaque = produtoDestaque;
19 }
20
21 public Produto getProdutoDestaque() {
22     return produtoDestaque;
23 }
24 }
```

9. Adicione na classe **ProdutoMB** do projeto **schedulingWeb** o seguinte atributo.

```
1 @EJB
2 private ProdutoDestaqueBean produtoDestaqueBean;
```

10. Adicione na classe **ProdutoMB** do projeto **schedulingWeb** o seguinte método.

```
1 public Produto getProdutoDestaque() {
2     return this.produtoDestaqueBean.getProdutoDestaque();
3 }
```

11. Acrescente na tela **produtos.xhtml** do projeto **schedulingWeb** o produto em destaque.

```
1 <h1>Produto Destaque</h1>
2
3 <h:outputLabel value="Nome: "/>
4 <h:outputText value="#{produtoMB.produtoDestaque.nome}"/>
5 <h:outputLabel value="Preço: "/>
6 <h:outputText value="#{produtoMB.produtoDestaque.preco}"/>
```

12. Acesse periodicamente a url <http://localhost:8080/schedulingWeb/produtos.xhtml> para verificar que o produto em destaque muda de tempos em tempos.



# Capítulo 10

## Contexts and Dependency Injection - CDI

Aplicações corporativas costumam utilizar tanto o container WEB para a camada de apresentação quanto o container EJB para a camada de negócio. A integração entre o container WEB e o container EJB pode ser mais facilmente realizada através dos recursos definidos pela especificação **Contexts and Dependency Injection - CDI**.

Dos recursos existentes na arquitetura CDI, podemos destacar o mecanismo de Injeção de Dependência e o gerenciamento do ciclo de vida dos objetos através de contextos. De acordo com a especificação CDI, os seguintes tipos de objetos possuem suporte a esses dois recursos:

- Managed Beans
- Session Beans
- Objetos criados por Producer Methods
- Objetos disponibilizados por Producer Fields
- Resources (Java EE resources, Persistence Contexts, Persistence Units, Remote EJBs and Web Services)

### 10.1 Managed Beans

Na arquitetura Java EE, os Managed Beans são objetos gerenciados pelo container Java EE. O container deve oferecer um pequeno conjunto de serviços fundamentais aos Managed Beans.

A definição básica do conceito de Managed Beans está documentada na especificação **Java EE 6 Managed Beans**. Contudo essa especificação permite que outras especificações estendam a idéia original de Managed Beans.

Não devemos confundir o conceito de Managed Beans do Java EE com o conceito de Managed Bean do JSF. Na verdade, um Managed Bean do JSF é um caso particular de um Managed Bean do Java EE.

A especificação CDI estende a definição de Managed Beans. Na arquitetura CDI, os Managed Beans são definidos por classes que devem respeitar certas restrições. Na seção 3.1.1 da especificação CDI são definidas essas restrições.

A top-level Java class is a managed bean if it is defined to be a managed bean by any other Java EE specification, or if it meets all of the following conditions:

- It is not a non-static inner class.
- It is a concrete class, or is annotated `@Decorator`.
- It is not annotated with an EJB component-defining annotation or declared as an EJB bean class in `ejb-jar.xml`.
- It does not implement `javax.enterprise.inject.spi.Extension`.
- It has an appropriate constructor—either:
  - the class has a constructor with no parameters, or
  - the class declares a constructor annotated `@Inject`.

All Java classes that meet these conditions are managed beans and thus no special declaration is required to define a managed bean.

As classes que se encaixam nessas restrições atuam como fonte de objetos que serão administrados pelo container CDI e poderão ser injetados em outros objetos.

## 10.2 Producer Methods and Fields

Os Producer Methods são apenas métodos que produzem objetos que serão administrados pelo container CDI e injetados em outros objetos. Os Producer Methods devem ser anotados com `@Produces`.

```

1 @Produces
2 public List<Produto> listaProdutos() {
3     // implementacao
4 }
```

Atributos também podem ser utilizados como fonte de objetos para o container Java EE. Os Producer Fields devem ser anotados com `@Produces`.

```

1 @Produces
2 public List<Produto> produtos;
```

## 10.3 EL Names

Na arquitetura CDI, páginas JSP ou JSF podem acessar objetos através de EL. Somente objetos com um **EL Name** podem ser acessados por páginas JSP ou JSF. A princípio, os seguintes tipos de objetos podem possuir um EL Name:

- Managed Beans
- Session Beans
- Objetos criados por Producer Methods
- Objetos disponibilizados por Producer Fields

## Contexts and Dependency Injection - CDI

Devemos aplicar a anotação **@Named** aos objetos que devem possuir um EL Name. Utilizando essa anotação, automaticamente, os objetos receberão um EL Name que é determinado de acordo com o tipo de objeto.

```
1 @Named // Managed Bean - EL Name: geradorDeApostas
2 public class GeradorDeApostas {
3     // implementacao
4 }
```

```
1 @Named
2 @Stateless // Session Bean - EL Name: geradorDeApostas
3 public class GeradorDeApostas {
4     // implementacao
5 }
```

```
1 @Named
2 @Produces // Producer Method - EL Name: listaProdutos
3 public List<Produto> listaProdutos() {
4     // implementacao
5 }
6
7 @Named
8 @Produces // Producer Method - EL Name: produtos
9 public List<Produto> getProdutos() {
10    // implementacao
11 }
```

```
1 @Named
2 @Produces // Producer Field - EL Name: produtos
3 public List<Produto> produtos;
```

É possível alterar o padrão definindo EL Names diretamente na anotação **@NAMED**.

```
1 @Named("gerador")
2 public class GeradorDeApostas {
3     // implementacao
4 }
```

## 10.4 beans.xml

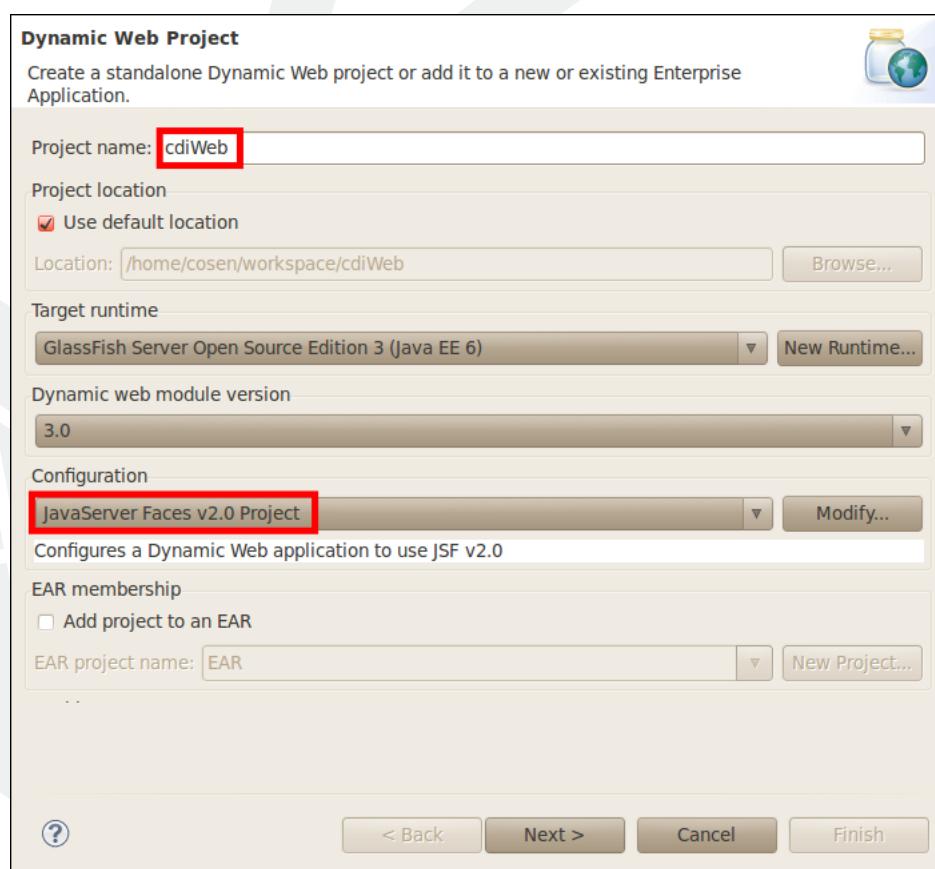
Para habilitar os recursos do CDI para os objetos de uma aplicação, é necessário adicionar um arquivo chamado **beans.xml** na pasta **META-INF** no classpath.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://java.sun.com/xml/ns/javaee"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="
5         http://java.sun.com/xml/ns/javaee
6         http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
7 </beans>
```

## 10.5 Exercícios

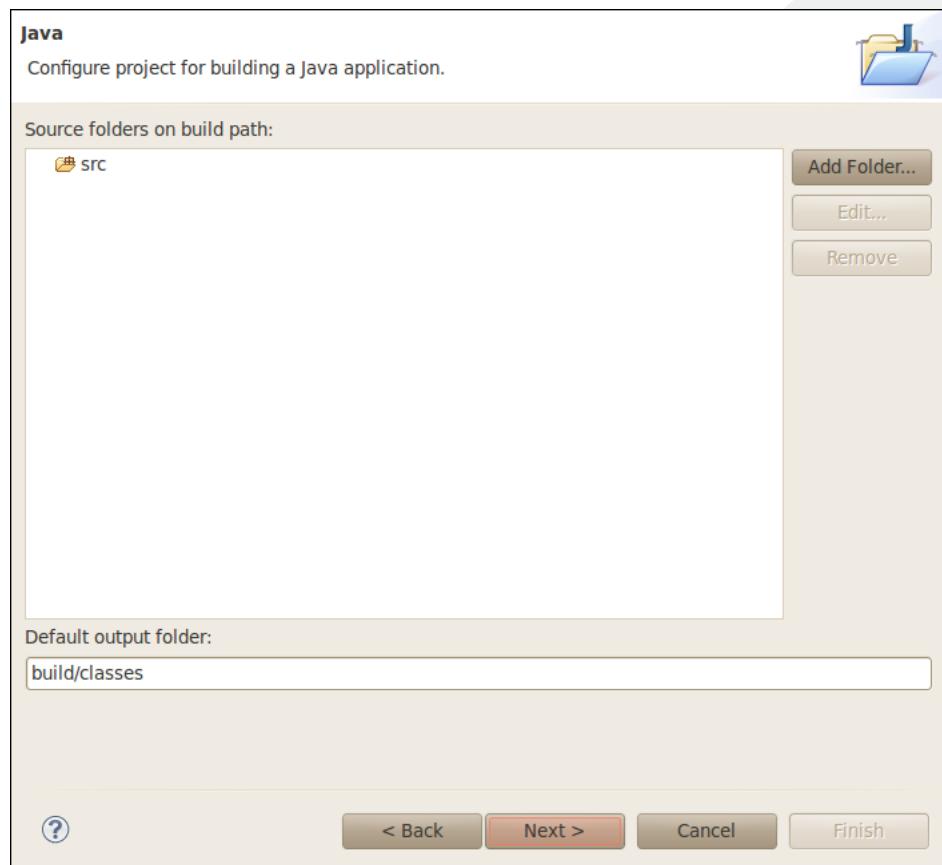
1. Crie um Dynamic Web Project no eclipse chamado **cdiWeb**. Você pode digitar “CTRL+3” em seguida “new Dynamic Web Project” e “ENTER”. Depois, siga exatamente as ima-

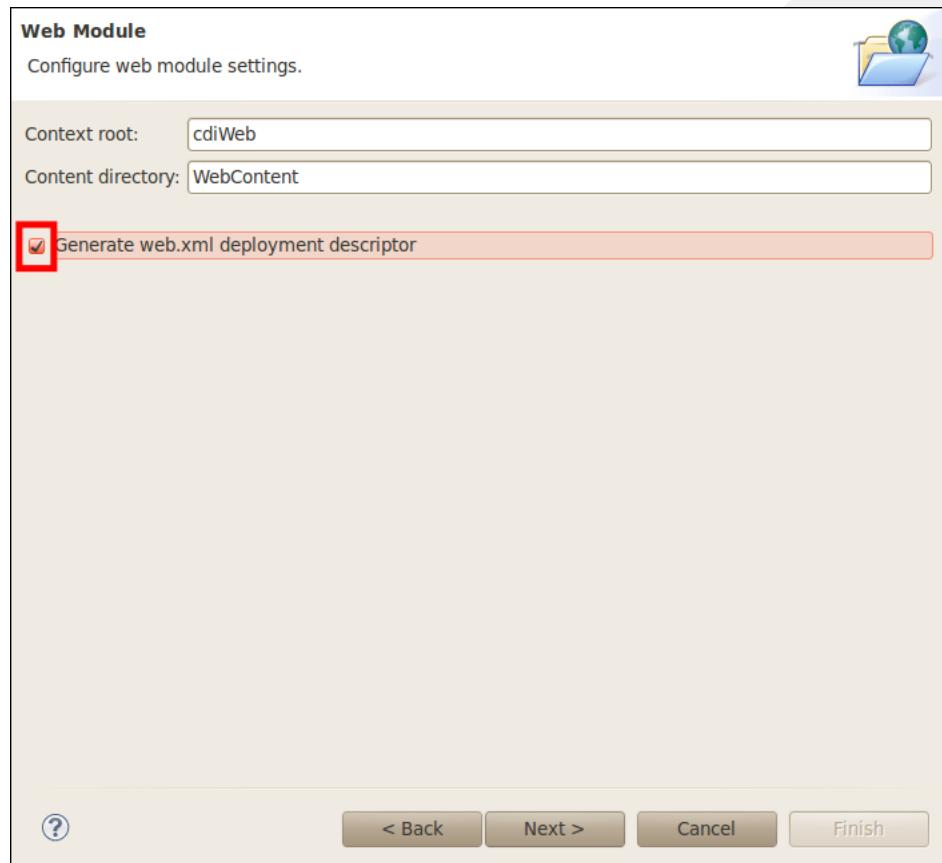
gens abaixo.

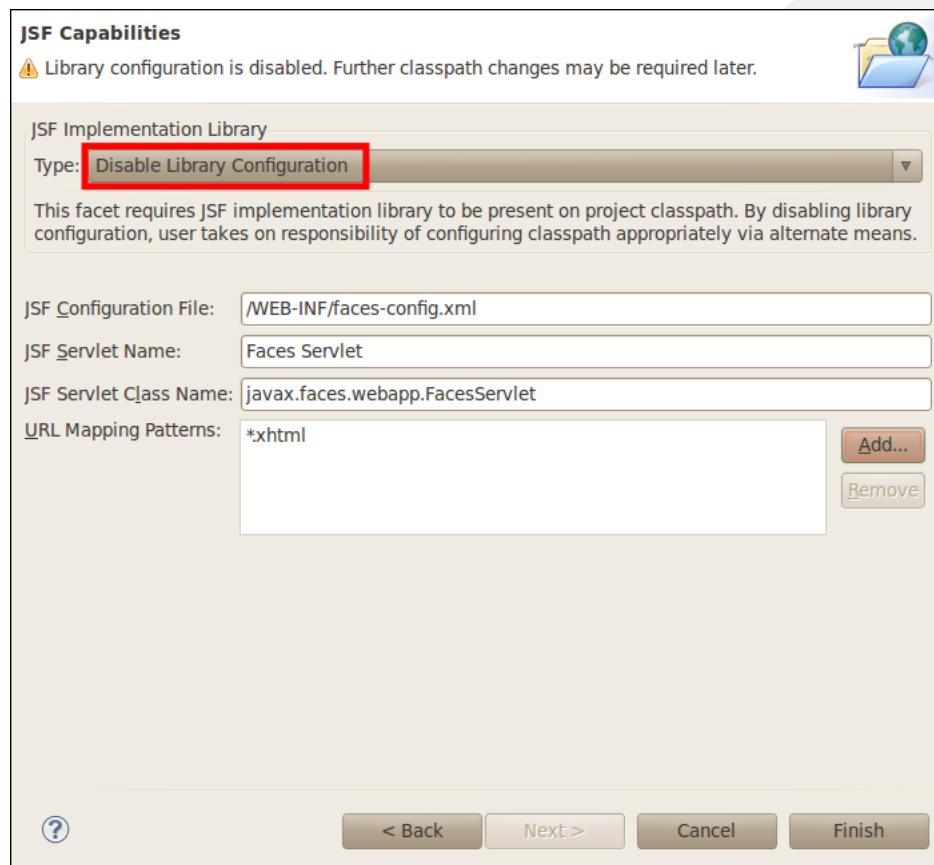


## *Contexts and Dependency Injection - CDI*

---







2. Habilite os recursos do CDI adicionando um arquivo chamado **beans.xml** na pasta **WEB-INF** do projeto **cdiWeb**.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://java.sun.com/xml/ns/javaee"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="
5          http://java.sun.com/xml/ns/javaee
6          http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
7  </beans>
```

3. Crie um pacote chamado **sessionbeans** no projeto **cdiWeb** e adicione nesse pacote um SLSB para funcionar como lançador de moeda.

```
1  @Named
2  @Stateless
3  public class LancadorDeMoedaBean {
4
5      private String resultado;
6
7      public void lanca() {
8          if(Math.random() < 0.5) {
9              this.resultado = "CARA";
10         } else {
11             this.resultado = "COROA";
12         }
13     }
14 }
```

```

15     public void setResultado(String resultado) {
16         this.resultado = resultado;
17     }
18
19     public String getResultado() {
20         return resultado;
21     }
22 }
```

- Crie uma tela para utilizar o lançador de moedas. Adicione na pasta **WebContent** do projeto **cdiWeb** um arquivo chamado **moeda.xhtml** com o seguinte conteúdo.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10    <title>Moeda</title>
11 </h:head>
12
13 <h:body>
14    <h1>Moeda</h1>
15    <h:form>
16        <h:commandButton action="#{lancadorDeMoedaBean.lanca}" value="Jogar"/>
17    </h:form>
18    <h2>Resultado: <h:outputText value="#{lancadorDeMoedaBean.resultado}"></h2>
19 </h:body>
20 </html>
```

- Acesse a url <http://localhost:8080/cdiWeb/moeda.xhtml>.
- Crie um pacote chamado **managedbeans** no projeto **cdiWeb** e adicione nesse pacote uma classe para gerar números aleatórios.

```

1 public class GeradorDeNumeros {
2
3     @Named
4     @Produces
5     public List<Double> getNumeros() {
6         List<Double> numeros = new ArrayList<Double>();
7         for (int i = 0; i < 5; i++) {
8             numeros.add(Math.random());
9         }
10        return numeros;
11    }
12 }
```

- Crie uma tela para utilizar o gerador de números. Adicione na pasta **WebContent** do projeto **cdiWeb** um arquivo chamado **numeros.xhtml** com o seguinte conteúdo.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10   <title>Números</title>
11 </h:head>
12
13 <h:body>
14   <h1>Números</h1>
15
16   <h:dataTable value="#{numeros}" var="numero">
17     <h:column>
18       <h:outputText value="#{numero}"/>
19     </h:column>
20   </h:dataTable>
21
22 </h:body>
23 </html>
```

8. Acesse a url <http://localhost:8080/cdiWeb/numeros.xhtml>.

## 10.6 Escopos e Contextos

Os objetos administrados pelo container CDI são armazenados em **contextos**. Conceitualmente, um contexto é uma coleção de objetos relacionados logicamente que devem existir durante um período de tempo específico. A especificação CDI define quatro contextos padrões.

**Request Context:** Quando se trata de aplicações Java WEB, para cada requisição HTTP um novo Request Context é criado pelo container CDI e destruído no final do processamento da mesma requisição.

Múltiplos Request Contexts podem existir simultaneamente.

**Session Context:** Um Session Context está sempre associado a uma HTTP Session. Quando uma HTTP Session é criada pelo container WEB, o container CDI cria um Session Context associado a essa HTTP Session. Quando uma HTTP Session é destruída pelo container WEB, o container CDI também destrói o Session Context correspondente.

Múltiplos Session Contexts podem existir simultaneamente.

**Application Context:** O container CDI cria um Application Context quando a aplicação é inicializada e o destrói quando a aplicação é finalizada.

Múltiplos Session Contexts **não** podem existir simultaneamente.

**Conversation Context:** Há dois tipos de Conversation Context: **transient** e **long-running**. Um Conversation Context do tipo transient se comporta de maneira muito parecida com o Request Context. Basicamente, um Conversation do tipo long-running é criado na chamada do método CONVERSATION-BEGIN() e destruído quando o método CONVERSATION-END() é executado.

Múltiplos Conversation Contexts podem existir simultaneamente.

Todo objeto administrado pelo container CDI possui um escopo. O escopo de um objeto define em qual contexto ele será armazenado quando criado pelo container CDI. A especificação CDI define cinco escopos padrões: Request, Session, Application, Conversation e Dependent.

Objetos com escopo Request, Session, Application e Conversation são armazenados no Request Context, Session Context, Application Context e Conversation Context respectivamente.

Um objeto com escopo Dependent pertence a outro objeto. O objeto dependente é armazenado indiretamente em algum contexto de acordo com o escopo do objeto a qual ele pertence.

As anotações: **@RequestScoped**, **@SessionScoped**, **@ApplicationScoped**, **@ConversationScoped** e **@Dependent** são utilizadas para definir o escopo dos objetos. Por padrão, se nenhuma anotação for definida o escopo dos objetos é o Dependent.

```
1 @RequestScoped
2 public class GeradorDeApostas {
3     // implementacao
4 }
```

```
1 @Produces
2 @SessionScoped
3 public List<Produto> listaProdutos() {
4     // implementacao
5 }
```

## 10.7 Injection Points

Quando um objeto é criado pelo container CDI, todas as dependências são injetados pelo container nesse objeto. As dependências são outros objetos pertencentes ao mesmo contexto do objeto que está sendo criado. Se alguma dependência não estiver criada o container se encarrega de criá-la antes.

As dependências de um objeto são definidas através de **Injection Points**. Há três tipos de Injection Points:

### 10.7.1 Bean Constructors

As dependências de um objeto podem ser definidas através de construtores com a anotação **@Inject**.

```
1 public class CarrinhoDeCompras {
2
3     @Inject
4     public CarrinhoDeCompras(Usuario usuario) {
5
6     }
7 }
```

### 10.7.2 Field

As dependências de um objeto podem ser definidas através de atributos com a anotação **@Inject**.

```
1 public class CarrinhoDeCompras {
2
3     @Inject
```

```
4     private Usuario usuario;
5
6 }
7 }
```

### 10.7.3 Initializer methods

As dependências de um objeto podem ser definidas através de métodos inicializadores com a anotação `@Inject`.

```
1 public class CarrinhoDeCompras {
2
3     private Usuario usuario;
4
5     @Inject
6     public void setUsuario(Usuario usuario){
7         this.usuario = usuario;
8     }
9 }
```

## 10.8 Exercícios

9. Altere o método `GETNUMEROS()` da classe `GERADORDENUMEROS` do projeto `cdiWeb` para que ele adicione uma mensagem no console toda vez que for chamado.

```
1 public class GeradorDeNumeros {
2
3     @Named
4     @Produces
5     public List<Double> getNumeros() {
6         System.out.println("GERANDO NÚMEROS");
7         List<Double> numeros = new ArrayList<Double>();
8         for (int i = 0; i < 5; i++) {
9             numeros.add(Math.random());
10        }
11        return numeros;
12    }
13 }
```

10. Acesse a url `http://localhost:8080/cdiWeb/numeros.xhtml` e depois observe as mensagens impressas no console do eclipse.
11. Para evitar que o método `GETNUMEROS` seja chamado mais do que uma vez por requisição HTTP, utilize o Resquet Scope.

```
1 public class GeradorDeNumeros {
2
3     @Named
4     @Produces
5     @RequestScoped
6     public List<Double> getNumeros() {
7         System.out.println("GERANDO NÚMEROS");
8         List<Double> numeros = new ArrayList<Double>();
9         for (int i = 0; i < 5; i++) {
10            numeros.add(Math.random());
11        }
12        return numeros;
13    }
14 }
```

12. Acesse novamente a url <http://localhost:8080/cdiWeb/numeros.xhtml> e depois observe as mensagens impressas no console do eclipse.
13. Utilize os recursos do CDI para implementar uma aplicação com todas as funcionalidades de CRUD. Utilize alunos como entidade dessa aplicação.

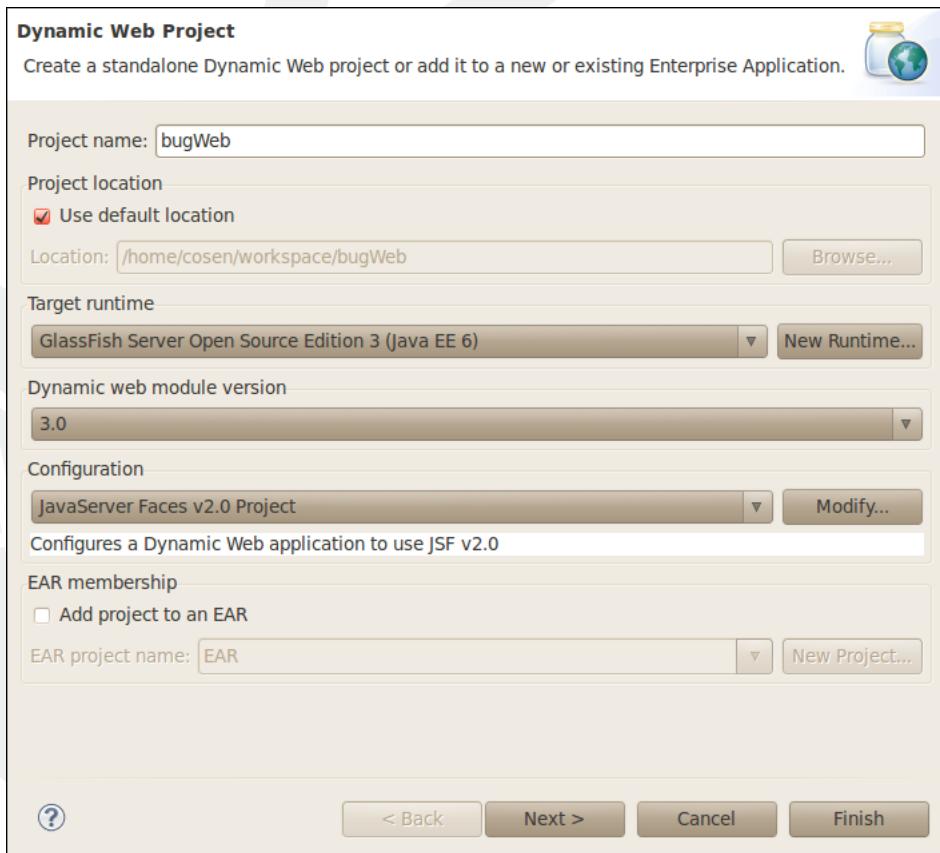
# Capítulo 11

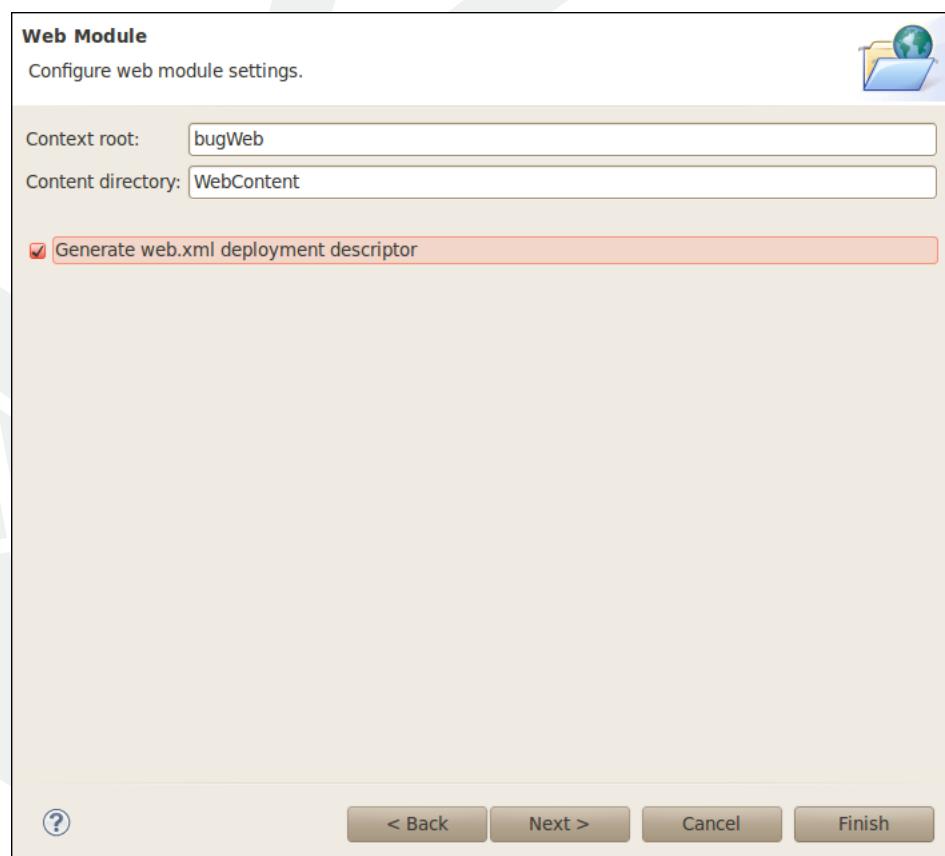
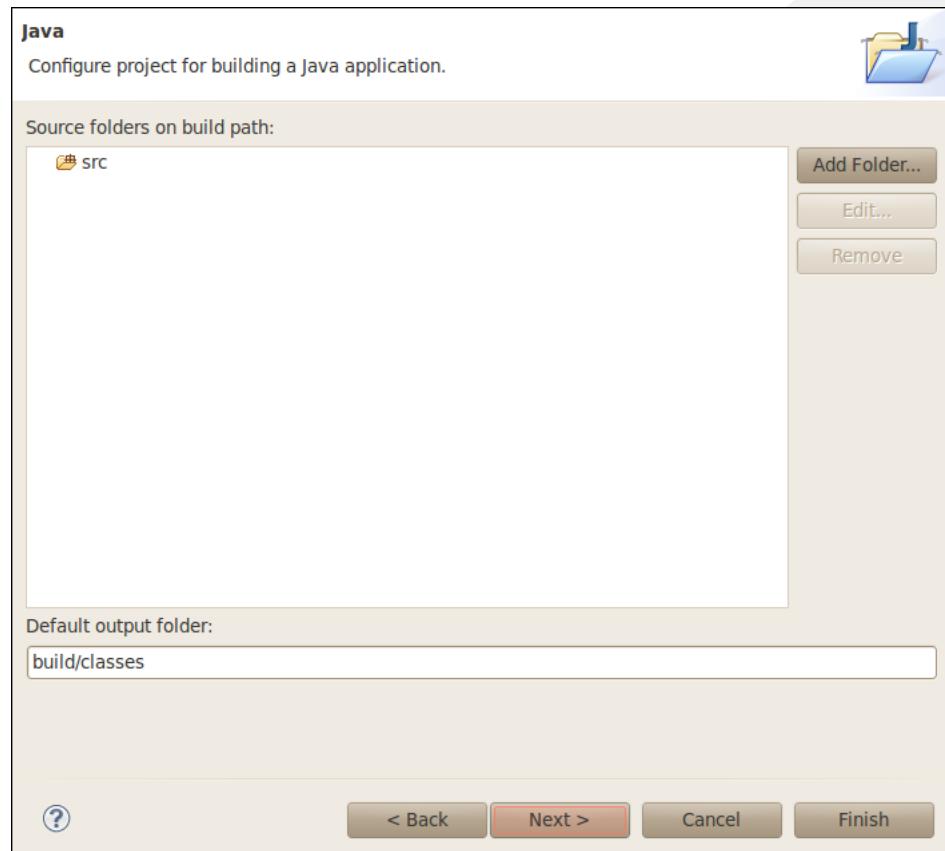
## Projeto

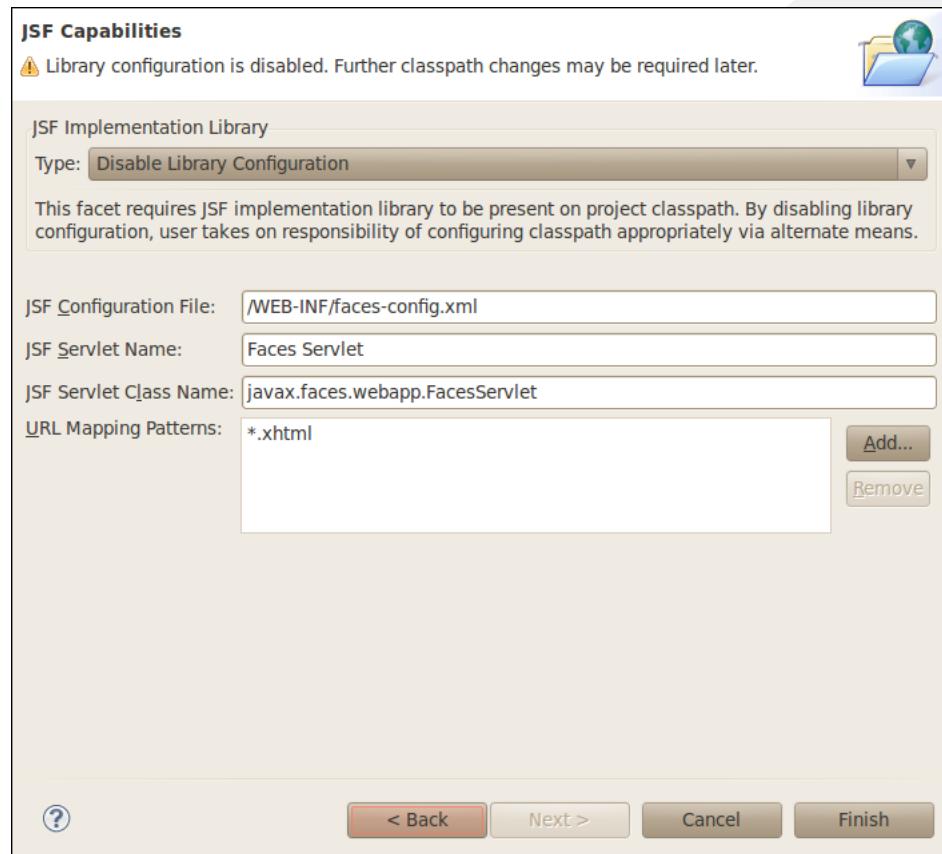
Neste capítulo, implementaremos um pequeno projeto para praticar os conceitos discutidos nos capítulos anteriores. Criaremos um sistema simples de cadastro de bugs.

### 11.1 Exercícios

1. Crie um Dynamic Web Project no eclipse chamado **bugWeb**. Você pode digitar “CTRL+3” em seguida “new Dynamic Web Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.







2. Configure a aplicação **bugWeb** para utilizar o data source **jdbc/K19** criado no capítulo **5**, adicionando o arquivo **persistence.xml** na pasta **META-INF** dentro do **src** do projeto **bugWeb**.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml-
5   /ns/persistence/persistence_1_0.xsd"
6   version="1.0">
7
8   <persistence-unit name="K19" transaction-type="JTA">
9     <provider>org.hibernate.ejb.HibernatePersistence</provider>
10    <jta-data-source>jdbc/K19</jta-data-source>
11
12    <properties>
13      <property name="hibernate.hbm2ddl.auto" value="update" />
14      <property name="hibernate.dialect" value="org.hibernate.dialect.←
15        MySQL5InnoDBDialect"/>
16    </properties>
17  </persistence-unit>
18</persistence>
```

3. Habilite os recursos do CDI adicionando o arquivo **beans.xml** na pasta **WEB-INF** do projeto **bugWeb**.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://java.sun.com/xml/ns/javaee"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="
5     http://java.sun.com/xml/ns/javaee"
```

```

6      http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
7  </beans>

```

4. Configure a aplicação **bugWeb** para que ele utilize o Realm **K19-Realm** criado no capítulo **7**, adicionando no arquivo **web.xml** do projeto **bugWeb** as configurações necessárias.

```

1 <login-config>
2   <realm-name>K19-Realm</realm-name>
3 </login-config>

```

5. Faça o mapeamento dos Groups do K19-Realm para os Roles da aplicação **bugWeb**.

```

1 <security-role-mapping>
2   <role-name>ADMIN</role-name>
3   <group-name>admin</group-name>
4 </security-role-mapping>
5
6 <security-role-mapping>
7   <role-name>USERS</role-name>
8   <group-name>users</group-name>
9 </security-role-mapping>

```

6. Crie um pacote chamado **entities** no projeto **bugWeb**.

7. Adicione no pacote **entities** um Entity Bean para modelar projetos e outro para modelar bugs.

```

1 @Entity
2 public class Project {
3
4   @Id @GeneratedValue
5   private Long id;
6
7   private String name;
8
9   private String description;
10
11  // GETTERS AND SETTERS
12 }

```

```

1 @Entity
2 public class Bug {
3
4   @Id @GeneratedValue
5   private Long id;
6
7   private String description;
8
9   private String severity;
10
11  @ManyToOne
12  private Project project;
13
14  // GETTERS AND SETTERS
15 }

```

8. Crie um pacote chamado **sessionbeans** no projeto **bugWeb**.

9. Adicione no pacote **sessionbeans** um SLSB para funcionar como repositório de projetos e outro para funcionar como repositório de bugs.

## Projeto

---

```
1  @Stateless
2  @RolesAllowed({ "ADMIN", "USERS" })
3  public class ProjectRepository {
4
5      @PersistenceContext
6      private EntityManager manager;
7
8      public void add(Project project) {
9          this.manager.persist(project);
10     }
11
12     public void edit(Project project) {
13         this.manager.merge(project);
14     }
15
16     @RolesAllowed({ "ADMIN" })
17     public void removeById(Long id) {
18         Project project = this.manager.find(Project.class, id);
19
20         TypedQuery<Bug> query = this.manager.createQuery(
21             "select x from Bug x where x.project = :project", Bug.class);
22         query.setParameter("project", project);
23         List<Bug> bugs = query.getResultList();
24         for (Bug bug : bugs) {
25             this.manager.remove(bug);
26         }
27
28         this.manager.remove(project);
29     }
30
31     @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
32     public List<Project> findAll() {
33         TypedQuery<Project> query = this.manager.createQuery(
34             "select x from Project x", Project.class);
35         return query.getResultList();
36     }
37
38     @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
39     public Project findById(Long id) {
40         return this.manager.find(Project.class, id);
41     }
42 }
```

```
1  @Stateless
2  @RolesAllowed({ "ADMIN", "USERS" })
3  public class BugRepository {
4      @PersistenceContext
5      private EntityManager manager;
6
7      public void add(Bug bug) {
8          this.manager.persist(bug);
9      }
10
11     public void edit(Bug bug) {
12         this.manager.merge(bug);
13     }
14
15     @RolesAllowed({ "ADMIN" })
16     public void removeById(Long id) {
17         Bug bug = this.manager.find(Bug.class, id);
18         this.manager.remove(bug);
19     }
20
21     @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
22     public List<Bug> findAll() {
23         TypedQuery<Bug> query = this.manager.createQuery("select x from Bug x",
24             Bug.class);
25         return query.getResultList();
```

```

26     }
27
28     @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
29     public Bug findById(Long id) {
30         return this.manager.find(Bug.class, id);
31     }
32 }
```

10. Crie um pacote chamado **managedbeans** no projeto **bugWeb**.

11. Adicione no pacote **managedbeans** um Managed Bean CDI para oferecer para as telas JSF as funcionalidades de CRUD relacionadas aos projetos.

```

1  @Named
2  @RequestScoped
3  public class ProjectMB {
4
5      @Inject
6      private ProjectRepository projectRepository;
7
8      private Project project = new Project();
9
10     private List<Project> projects;
11
12     public void save(){
13         if(this.getProject().getId() == null){
14             this.projectRepository.add(this.getProject());
15         } else {
16             this.projectRepository.edit(this.getProject());
17         }
18         this.project = new Project();
19         this.projects = null;
20     }
21
22     public void delete(Long id){
23         this.projectRepository.removeById(id);
24         this.projects = null;
25     }
26
27     public void prepareEdit(Long id){
28         this.project = this.projectRepository.findById(id);
29     }
30
31     public Project getProject() {
32         return project;
33     }
34
35     public List<Project> getProjects() {
36         if(this.projects == null){
37             this.projects = this.projectRepository.findAll();
38         }
39         return projects;
40     }
41 }
```

12. Adicione no pacote **managedbeans** um Managed Bean CDI para oferecer para as telas JSF as funcionalidades de CRUD relacionadas aos bugs.

```

1  @Named
2  @RequestScoped
3  public class BugMB {
4
5      @Inject
6      private BugRepository bugRepository;
7
8      @Inject
```

## Projeto

---

```
9  private ProjectRepository projectRepository;
10 private Bug bug = new Bug();
11 private Long projectId;
12 private List<Bug> bugs;
13
14 public void save(){
15     Project project = this.projectRepository.findById(this.projectId);
16     this.bug.setProject(project);
17
18     if(this.getBug().getId() == null){
19         this.bugRepository.add(this.getBug());
20     } else {
21         this.bugRepository.edit(this.getBug());
22     }
23     this.bug = new Bug();
24     this.bugs = null;
25 }
26
27 public void delete(Long id){
28     this.bugRepository.removeById(id);
29     this.bug = null;
30 }
31
32 public void prepareEdit(Long id){
33     this.bug = this.bugRepository.findById(id);
34 }
35
36 public Bug getBug() {
37     return bug;
38 }
39
40 public List<Bug> getBugs() {
41     if(this.bugs == null){
42         this.bugs = this.bugRepository.findAll();
43     }
44     return bugs;
45 }
46
47 public void setprojectId(Long projectId) {
48     this.projectId = projectId;
49 }
50
51 public Long getprojectId() {
52     return projectId;
53 }
54
55 }
```

13. Adicione no pacote **managedbeans** um Managed Bean CDI para implementar o processo de login e logout.

```
1 @Named
2 @RequestScoped
3 public class AuthenticatorMB {
4     private String username;
5     private String password;
6
7     public String login() throws ServletException{
8         FacesContext context = FacesContext.getCurrentInstance();
9         HttpServletRequest request = (HttpServletRequest) context.getExternalContext() ↳
10            .getRequest();
11         request.login(this.username, this.password);
12
13         return "/projects";
14     }
15     public String logout() throws ServletException {
```

```

15     FacesContext context = FacesContext.getCurrentInstance();
16     HttpServletRequest request = (HttpServletRequest) context.getExternalContext() ←
17         .getRequest();
18     request.logout();
19     return "/login";
20 }
21 // GETTERS AND SETTERS
22 }
```

14. Crie um pacote chamado **filters** no projeto **bugWeb**.

15. Adicione no pacote **filters** um Filtro para verificar a autenticação dos usuários.

```

1 @WebFilter(servletNames = { "Faces Servlet" })
2 public class AuthenticatorFilter implements javax.servlet.Filter {
3
4     @Override
5     public void doFilter(ServletRequest request, ServletResponse response,
6             FilterChain chain) throws IOException, ServletException {
7         HttpServletRequest req = (HttpServletRequest) request;
8
9         if (req.getRemoteUser() == null && !req.getRequestURI().endsWith(req.←
10             getContextPath() + "/login.xhtml")) {
11             HttpServletResponse res = (HttpServletResponse) response;
12             res.sendRedirect(req.getContextPath() + "/login.xhtml");
13         } else {
14
15             chain.doFilter(request, response);
16         }
17     }
18
19     @Override
20     public void init(FilterConfig filterConfig) throws ServletException {
21     }
22
23     @Override
24     public void destroy() {
25 }
```

16. Crie um menu para as telas da aplicação **bugWeb**. Adicione um arquivo chamado **menu.xhtml** na pasta **WebContent** do projeto **bugWeb**.

```

1 <ui:composition xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:ui="http://java.sun.com/jsf/facelets"
3     xmlns:h="http://java.sun.com/jsf/html"
4     xmlns:f="http://java.sun.com/jsf/core">
5
6     <h:form>
7         <h:panelGrid>
8             <h:commandLink action="#{authenticatorMB.logout}">logout</h:commandLink>
9
10            <h:outputLink value="#{request.contextPath}/projects.xhtml">Projects</→
11                h:outputLink>
12
13                <h:outputLink value="#{request.contextPath}/bugs.xhtml">Bugs</h:outputLink→
14            >
15        </h:panelGrid>
16    </h:form>
17 </ui:composition>
```

17. Crie um tela de login na aplicação **bugWeb**. Adicione um arquivo chamado **login.xhtml** na pasta **WebContent** do projeto **bugWeb**.

## Projeto

---

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10   <title>Login</title>
11 </h:head>
12
13 <h:body>
14   <h1>Login</h1>
15   <h:form>
16     <h:panelGrid>
17       <h:outputLabel value="Username:>" />
18       <h:inputText value="#{authenticatorMB.username}" />
19
20       <h:outputLabel value="Password:>" />
21       <h:inputSecret value="#{authenticatorMB.password}" />
22
23       <h:commandButton action="#{authenticatorMB.login}" value="login" />
24     </h:panelGrid>
25   </h:form>
26 </h:body>
27 </html>
```

18. Crie uma tela para administrar os projetos. Adicione um arquivo chamado **projects.xhtml** na pasta **WebContent** do projeto **bugWeb**.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10   <title>Projects</title>
11 </h:head>
12
13 <h:body>
14   <ui:include src="/menu.xhtml"/>
15
16   <hr/>
17
18   <h1>New Project</h1>
19   <h:form>
20     <h:panelGrid>
21       <h:inputHidden value="#{projectMB.project.id}" />
22
23       <h:outputLabel value="Name:>" />
24       <h:inputText value="#{projectMB.project.name}" />
25
26       <h:outputLabel value="Description:>" />
27       <h:inputTextarea value="#{projectMB.project.description}" />
28
29       <h:commandButton action="#{projectMB.save}" value="Save" />
30     </h:panelGrid>
31   </h:form>
32
33   <hr/>
34
35   <h1>Project List</h1>
36   <h:dataTable value="#{projectMB.projects}" var="project"
```

```

37      rendered="#{not empty projectMB.projects}" border="1">
38      <h:column>
39          <f:facet name="header">Id</f:facet>
40          #{project.id}
41      </h:column>
42
43      <h:column>
44          <f:facet name="header">Name</f:facet>
45          #{project.name}
46      </h:column>
47
48      <h:column>
49          <f:facet name="header">Description</f:facet>
50          #{project.description}
51      </h:column>
52
53      <h:column>
54          <f:facet name="header">Delete</f:facet>
55          <h:form>
56              <h:commandLink action="#{projectMB.delete(project.id)}" >delete</>
57                  h:commandLink>
58          </h:form>
59      <h:column>
60          <f:facet name="header">Edit</f:facet>
61          <h:form>
62              <h:commandLink action="#{projectMB.prepareEdit(project.id)}" >edit</>
63                  h:commandLink>
64          </h:form>
65      </h:column>
66  </h:dataTable>
67 </h:body>
</html>

```

19. Crie uma tela para administrar os bugs. Adicione um arquivo chamado **bugs.xhtml** na pasta **WebContent** do projeto **bugWeb**.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5      xmlns:ui="http://java.sun.com/jsf/facelets"
6      xmlns:h="http://java.sun.com/jsf/html"
7      xmlns:f="http://java.sun.com/jsf/core">
8
9  <h:head>
10     <title>Bugs</title>
11 </h:head>
12
13 <h:body>
14     <ui:include src="/menu.xhtml"/>
15
16     <hr />
17
18     <h1>New Bug</h1>
19     <h:form>
20         <h:panelGrid>
21             <h:inputHidden value="#{bugMB.bug.id}" />
22
23             <h:outputLabel value="Severity:> />
24             <h:selectOneMenu value="#{bugMB.bug.severity}">
25                 <f:selectItem itemValue="LOW" />
26                 <f:selectItem itemValue="MEDIUM" />
27                 <f:selectItem itemValue="HIGH" />
28             </h:selectOneMenu>
29
30             <h:outputLabel value="Description:> />
31

```

## Projeto

---

```
32         <h:inputTextarea value="#{bugMB.bug.description}" />
33
34         <h:outputLabel value="Project:" />
35         <h:selectOneMenu value="#{bugMB.projectId}">
36             <f:selectItems value="#{projectMB.projects}" var="project"
37                 itemLabel="#{project.name}" itemValue="#{project.id}" />
38         </h:selectOneMenu>
39
40         <h:commandButton action="#{bugMB.save}" value="Save" />
41     </h:panelGrid>
42 </h:form>
43
44 <hr />
45
46 <h1>Bug List</h1>
47 <h: dataTable value="#{bugMB.bugs}" var="bug"
48     rendered="#{not empty bugMB.bugs}" border="1">
49     <h:column>
50         <f:facet name="header">Id</f:facet>
51         #{bug.id}
52     </h:column>
53
54     <h:column>
55         <f:facet name="header">Project</f:facet>
56         #{bug.project.name}
57     </h:column>
58
59     <h:column>
60         <f:facet name="header">Severity</f:facet>
61         #{bug.severity}
62     </h:column>
63
64     <h:column>
65         <f:facet name="header">Description</f:facet>
66         #{bug.description}
67     </h:column>
68
69     <h:column>
70         <f:facet name="header">Delete</f:facet>
71         <h:form>
72             <h:commandLink action="#{bugMB.delete(bug.id)}" delete=>/h:commandLink>
73         </h:form>
74     </h:column>
75     <h:column>
76         <f:facet name="header">Edit</f:facet>
77         <h:form>
78             <h:commandLink action="#{bugMB.prepareEdit(bug.id)}" edit=>/h:commandLink>
79         </h:form>
80     </h:column>
81 </h: dataTable>
82 </h:body>
83 </html>
```

20. Crie uma tela para os erros internos e os erros de autorização. Adicione um arquivo chamado **error.xhtml** na pasta **WebContent** do projeto **bugWeb**.

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5      xmlns:ui="http://java.sun.com/jsf/facelets"
6      xmlns:h="http://java.sun.com/jsf/html"
7      xmlns:f="http://java.sun.com/jsf/core">
8
9  <h:head>
10     <title>Error</title>
11 </h:head>
```

```
12 <h:body>
13     <h3>Internal Error or Client not authorized for this invocation.</h3>
14 </h:body>
15 </html>
```

21. Configure a página de erro no arquivo **web.xml** do projeto **bugWeb**. Adicione o seguinte trecho de código nesse arquivo.

```
1 <error-page>
2     <exception-type>java.lang.Exception</exception-type>
3     <location>/error.xhtml</location>
4 </error-page>
```

22. Teste a aplicação!!!