# Programação Web com Jsp, Servlets e J2EE

**André Temple**
CPqD Telecom & IT Solutions.

**Rodrigo Fernandes de Mello**
Departamento de Ciências da Computação
Instituto de Ciências Matemáticas e de Computação.
Universidade de São Paulo

**Danival Taffarel Calegari**
CPqD Telecom & IT Solutions.

**Maurício Schiezaro**
CPqD Telecom & IT Solutions.

ISBN: 85-905209-1-9

# Prefácio

Este livro é dividido em duas partes. A primeira parte trata do desenvolvimento de aplicações Web utilizando, principalmente, Servlets. A segunda parte aborda o paradigma de desenvolvimento de aplicações distribuídas utilizando a tecnologia Enterprise Java Beans.

A primeira parte do livro aborda de forma consistente e didática o conteúdo relacionado a Servlets. Nesta abordagem é definido um histórico do desenvolvimento de aplicações Web, passando por CGIs e linguagens de script. Além dos tópicos relacionados a Servlets são abordados tópicos relacionados tais como Java Server Pages (JSP), Java Beans, Taglibs, modelo MVC, instalação de configuração de um Web Container para desenvolvimento, além de tópicos avançados tais como controle de pooling de conexões com banco de dados.

A segunda parte do livro aborda o paradigma de desenvolvimento de aplicações distribuídas, destacando a evolução das técnicas de desenvolvimento desde a programação estrutura até o atual uso de sistemas distribuídos. A tecnologia utilizada para prover distribuição é a plataforma J2EE (Java 2 Enterprise Edition). São detalhados os componentes e possibilidades que esta plataforma oferecem ao desenvolvedor.

O livro foi dividido em partes para oferecer um conteúdo mais abrangente e completar. A primeira parte trata da construção de interfaces e controles para interação com os clientes de uma aplicação Web. A segunda parte aprofunda nos aspectos de distribuição de um sistema, permitindo que este execute em diversos computadores, dividindo sua carga e, conseqüentemente, aumentando seu desempenho.

# Sumário

# Parte I

# Desenvolvendo Interfaces e Controles de Interação com o Usuário

# Capítulo 1

# Introdução

Introduzimos, nesse capítulo, a tecnologia de Servlets e Páginas JSP e mostramos algumas características que tornam essas tecnologias bastante atraentes para o desenvolvimento de aplicações na Web.

## 1.1 Aplicações na Web

Se um dia a Internet era composta, principalmente, de páginas estáticas com conteúdo institucional, hoje ela oferece uma infinidade de aplicações com conteúdo dinâmico e personalizado.

Diversas tecnologias possibilitaram essa revolução: seja para construir um simples site com conteúdo dinâmico ou para construir um complexo sistema de Business-To-Business, é necessária a utilização de ferramentas que possibilitem consultas a bancos de dados, integração com sistemas corporativos, entre outras inúmeras funcionalidades.

Dentre as diversas tecnologias disponíveis atualmente para o desenvolvimento dessa classe de aplicações, destaca-se a de Servlets e a de páginas JSP (Java Server Pages).

A utilização de Servlets e de páginas JSP oferece diversas vantagens em relação ao uso de outras tecnologias (como PHP, ASP e CGI). As principais vantagens são herdadas da própria linguagem Java:

Portabilidade: a aplicação desenvolvida pode ser implantada em diversas plataformas, como por exemplo Windows, Unix e Macintosh, sem que seja necessário modificar ou mesmo reconstruir a aplicação.

Facilidade de programação: a programação é orientada a objetos, simplificando o desenvolvimento de sistemas complexos. Além disso, a linguagem oferece algumas facilidades, como por exemplo o gerenciamento automático de memória (estruturas alocadas são automaticamente liberadas, sem que o desenvolvedor precise se preocupar em gerenciar esse processo).

Flexibilidade: o Java já se encontra bastante difundido, contando com uma enorme comunidade de desenvolvedores, ampla documentação e diversas bibliotecas e códigos prontos, dos quais o desenvolvedor pode usufruir.

Além dessas vantagens, a arquitetura de Servlets e páginas JSP possibilita alguns benefícios adicionais:

Escalabilidade: na maior parte dos servidores de aplicações modernos, é possível distribuir a carga de processamento de aplicações desenvolvidas em diversos servidores, sendo que servidores podem ser adicionados ou removidos de maneira a acompanhar o aumento ou decréscimo dessa carga de processamento.

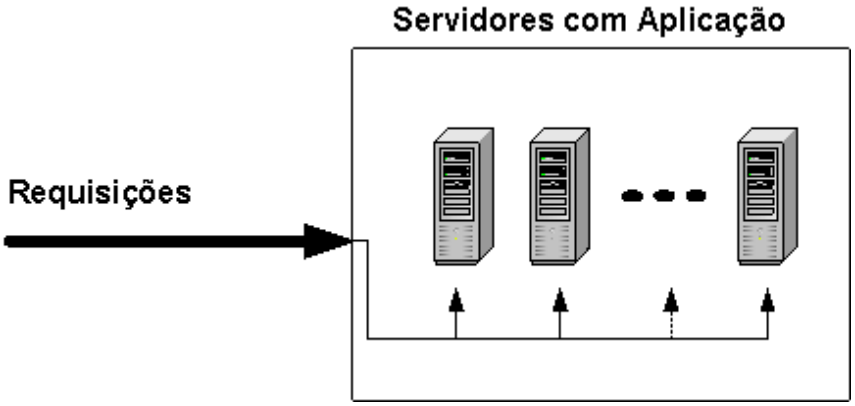


***Figura 1.1 – Exemplo de arquitetura distribuída com Servlets e Páginas JSP.***

Eficiência: os Servlets carregados por um servidor persistem em sua memória até que ele seja finalizado. Assim, ao contrário de outras tecnologias, não são iniciados novos processos para atender cada requisição recebida; por outro lado, uma mesma estrutura alocada em memória pode ser utilizada no atendimento das diversas requisições que chegam a esse mesmo Servlet.

Recompilação automática: páginas JSP modificadas podem ser automaticamente recompiladas, de maneira que passem a incorporar imediatamente as alterações sem que seja necessário interromper o funcionamento da aplicação como um todo.

## 1.1 Comparando Servlets com CGIs

O CGI, ou Common Gateway Interface, surgiu como uma das primeiras tecnologias disponíveis para a geração de conteúdo dinâmico em servidores Web: o desenvolvedor implementa uma aplicação que deve ser executada a cada requisição recebida, sendo que o servidor Web passa para essa aplicação, através de variáveis de ambiente e entrada padrão, os parâmetros recebidos, e retorna a saída da aplicação como resposta da requisição.

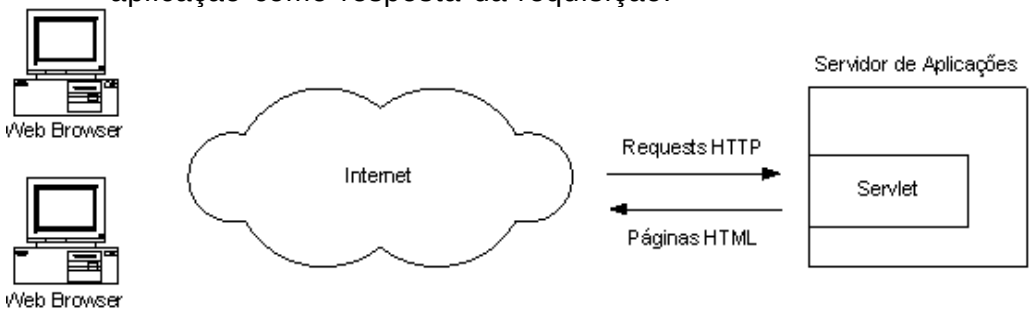


***Figura 1.2 – Funcionamento de um CGI.***

Podemos usar o CGI para analisar algumas das vantagens em se utilizar Servlets.

Em primeiro lugar, há um grande ganho em performance na utilização de Servlets: ao invés de iniciar um novo processo a cada requisição recebida, um Servlet fica

carregado em memória e atende as requisições recebidas através de novos “threads”.

Além disso, um Servlet pode tirar proveito dessa persistência para manter também em memória recursos que demandem grande processamento para serem inicializados. Um exemplo típico, para esse caso, é a manutenção de conexões com banco de dados: ao invés de inicializar uma nova conexão com o banco de dados a cada requisição recebida, um Servlet pode inicializar diversas conexões ao ser carregado, e simplesmente alocar uma conexão desse pool a cada requisição recebida (e retornar a conexão ao pool após o atendimento da requisição).

Além destes ganhos de performance, a utilização de um Servlet possibilita uma maneira mais padronizada e portável de se distribuir / implantar sua aplicação. Conforme explicado mais adiante no *Capítulo 2 – Instalação e Configuração*, o ambiente onde sua aplicação será implantada não precisa ser igual ao ambiente onde foi feito o desenvolvimento: seus Servlets podem ser “instalados” em qualquer ambiente onde haja um Servidor de Aplicações que implemente a especificação de Servlets.

Por fim, estaremos apresentando ao longo deste livro as diversas características e funcionalidades da tecnologia de Servlets que tornam o seu desenvolvimento muito mais simples, e o resultado, muito mais eficiente e robusto.

## 1.2 O que são Servlets?

Servlets são classes Java, desenvolvidas de acordo com uma estrutura bem definida, e que, quando instaladas junto a um Servidor que implemente um Servlet Container (um servidor que permita a execução de Servlets, muitas vezes chamado de Servidor de Aplicações Java), podem tratar requisições recebidas de clientes.

Um cenário típico de funcionamento de uma aplicação desenvolvida com Servlets é o seguinte:

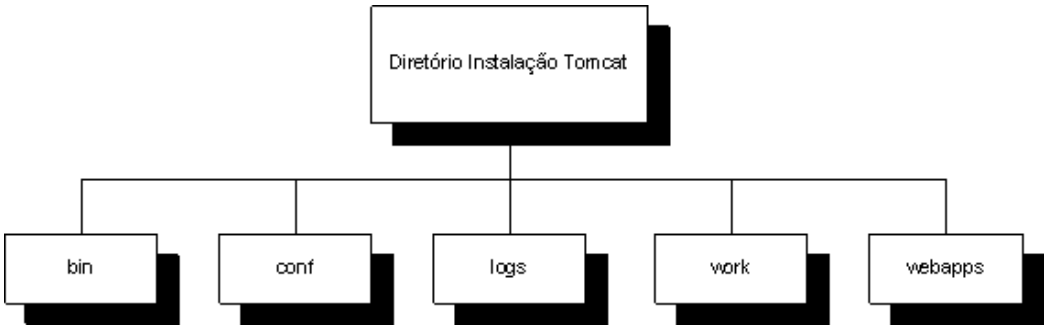


***Figura 1.3 – Exemplo de funcionamento de uma aplicação com Servlets.***

Ao receber uma requisição, um Servlet pode capturar parâmetros desta requisição, efetuar qualquer processamento inerente a uma classe Java, e devolver uma página HTML por exemplo.

**Exemplo de Servlet**

```
import java.io.*;
import javax.servlet.http.*;


// Servlet simples que retorna página HTML com o endereço IP
// do cliente que está fazendo o acesso
public class RemoteIPServlet extends HttpServlet {


    public void doGet( HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");
        l_pw.println("O seu endereço IP é \"" +
p_request.getRemoteAddr () + "\"");
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }
}
```

# 1.3 O que são páginas JSP?

As páginas JSP, ou Java Server Pages, foram criadas para contornar algumas das limitações no desenvolvimento com Servlets: se em um Servlet a formatação da página HTML resultante do processamento de uma requisição se mistura com a lógica da aplicação em si, dificultando a alteração dessa formatação, em uma página JSP essa formatação se encontra separada da programação, podendo ser modificada sem afetar o restante da aplicação.

Assim, um JSP consiste de uma página HTML com alguns elementos especiais, que conferem o caráter dinâmico da página. Esses elementos podem tanto realizar um processamento por si, como podem recuperar o resultado do processamento realizado em um Servlet, por exemplo, e apresentar esse conteúdo dinâmico junto a página JSP.

Existe também um recurso adicional bastante interessante na utilização de páginas JSP: a recompilação automática, que permite que alterações feitas no código da página sejam automaticamente visíveis em sua apresentação. Assim, não é necessário interromper o funcionamento da aplicação para incorporar uma modificação de layout, por exemplo.

### Exemplo de Página JSP

```
<!—Página JSP Simples que imprime endereço IP da máquina que
está fazendo o acesso a esta página —>
<HTML>
   <BODY>
      O seu endereço IP é “<%= request.getRemoteAddr () %>”
   </BODY>
</HTML>
```

# Capítulo 2

# Instalação e Configuração

Nesse capítulo são apresentados os tópicos referentes a instalação e configuração de um ambiente básico para a implantação e execução de aplicações Web com Servlets e páginas JSP.

## 2.1 Pré-requisitos:

O primeiro passo para desenvolver aplicações Web com Servlets e páginas JSP é a configuração de um ambiente básico para a implantação e execução dessas aplicações. Esse ambiente básico pressupõe a instalação de dois componentes principais: o Java 2 Standard Development Kit (J2SDK), utilizado para compilar aplicações Java, e um Servlet Container, que irá executar os Servlets desenvolvidos.

Alguns sistemas operacionais já possuem um J2SDK instalado por "default". Caso esse ainda não se encontre instalado, pode-se obtê-lo no site oficial do Java (http://java.sun.com). Neste site é possível selecionar entre as versões de J2SDK para as diversas plataformas de mercado tais como Windows, Linux, Solaris e outros.

O outro componente necessário para o desenvolvimento de Servlets e JSP é um servidor que implemente um Servlet Container. Esse servidor será responsável por prover um framework básico para as diversas aplicações desenvolvidas, inicializando-as, distribuindo as

requisições entre elas e tratando os resultados do processamento de cada aplicação.

Apesar de existirem diversos servidores disponíveis no mercado, para efeito dos exemplos apresentados neste livro, utilizaremos o Apache Tomcat, disponível no site http://jakarta.apache.org. Esse servidor de aplicações atende às especificações mencionadas anteriormente e pode ser utilizado sem nenhum custo para o desenvolvedor.

Um último componente normalmente utilizado para o desenvolvimento de Servlets e páginas JSP é um ambiente gráfico de desenvolvimento (IDE). Porém, a escolha e configuração de um ambiente desse tipo foge do escopo deste livro, sendo deixado para o leitor a tarefa de escolher a ferramenta que melhor atenda às suas necessidades.

## 2.1 Instalação e configuração no Apache Tomcat

No site do Apache Tomcat são disponibilizadas versões do software com instaladores para os diversos sistemas operacionais.

Normalmente esse processo de instalação é simples, e, uma vez finalizado, tem-se um servidor de aplicações pronto para produção. De qualquer forma, o site disponibiliza toda a documentação necessária para resolver problemas encontrados e esclarecer dúvidas com relação ao processo de instalação e configuração do servidor.

Para entender um pouco mais a fundo o funcionamento do Tomcat, deve-se examinar os diretórios criados durante o processo de instalação. Os principais diretórios criados são:

| Diretório | Descrição |
|---|---|
| bin | Executáveis, incluindo os aplicativos para iniciar e para encerrar a execução do servidor |

| | |
|---|---|
| conf | Arquivos de configuração do Tomcat. O arquivo “server.xml”, em particular, define uma série de parâmetros para a execução do servidor, como por exemplo, a porta onde o servidor irá receber requisições (essa porta é, por default, 8080), devendo ser examinado com cuidado e modificado conforme as necessidades. |
| logs | Arquivos de log do servidor. Além de gerar arquivos de log contendo entradas para cada requisição recebida, como qualquer servidor Web, o Tomcat também pode gerar arquivos de log com tudo o que as aplicações desenvolvidas enviam para as saídas padrão do sistema: tipicamente, o que é impresso através do “System.out” é acrescido no arquivo “stdout.log”, e tudo o que é impresso através do “System.err” é acrescido no arquivo “stderr.log”. |
| work | Diretório temporário do Tomcat. Esse diretório é utilizado, por exemplo, para realizar a recompilação automática de páginas JSP (esse processo é explicado mais adiante no capítulo “Páginas JSP”). |
| webapps | Nesse diretório são instaladas as diversas aplicações web desenvolvidas por você ou por terceiros. |

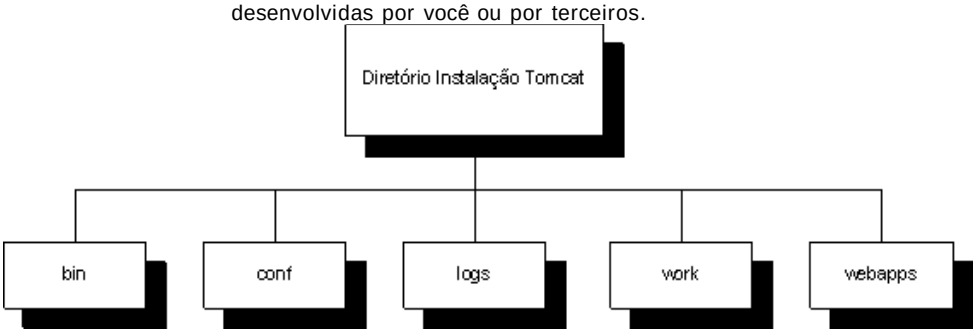


***Figura 2.1 – Subdiretórios na instalação do Apache Tomcat.***

## 2.2 Instalação e Configuração de uma Aplicação Web

Conforme vimos anteriormente, existe um diretório no Apache Tomcat chamado “webapps” onde devem ser instaladas as diversas aplicações desenvolvidas por você ou por terceiros.

Para que possamos mostrar como são feitas essas instalações, precisamos antes definir o que é uma

aplicação Web: a partir de agora, estaremos chamando de uma aplicação Web um conjunto de Servlets, páginas JSP, classes Java, bibliotecas, ícones, páginas HTML e outros elementos, que podem ser empacotados juntos e que provêem as funcionalidades previstas pela aplicação.

Essa definição está contida, na verdade, na própria especificação de Servlets Java, não sendo específica, portanto, à utilização do Servidor de Aplicações Apache Tomcat. Isso significa que as aplicações desenvolvidas por você podem ser instaladas em qualquer servidor que implemente a especificação de Servlets (como o IBM® Websphere® e o Bea Weblogic®): com isso, segue-se o princípio da linguagem Java de desenvolver o código uma só vez e implantá-lo em múltiplas plataformas.

**Aplicação Web:** Aplicação composta de Servlets + Páginas JSP + Bibliotecas e classes Java + imagens + páginas HTML e outros componentes estáticos que podem ser empacotados juntos e instalados em qualquer Servlet Container.

De acordo com a especificação de Servlets, existem duas maneiras pelas quais uma aplicação web pode ser instalada junto a um Servlet Container: por meio de um arquivo WAR (**W**eb **A**pplication A**r**chive), explicado mais adiante no *Capítulo 9*, ou por meio de uma estrutura de diretórios criada junto ao servidor. No caso específico do Tomcat, essa estrutura deve ser criada abaixo do diretório "webapps".

Para uma determinada aplicação Web, a estrutura de diretórios mínima que deve ser criada abaixo do diretório "webapps" é a seguinte:

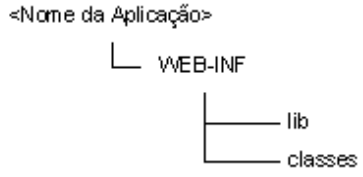


***Figura 2.2 – Estrutura mínima de diretórios de uma Aplicação Web.***

Conforme pode ser visto na figura anterior, deve ser criado, abaixo do diretório webapps, um diretório com o nome da aplicação. Esse diretório deve conter pelo menos um subdiretório WEB-INF; podem haver além do subdiretório WEB-INF, por outro lado, outros subdiretórios e arquivos, como páginas html, páginas JSP etc.

O diretório WEB-INF, por sua vez, deve conter um arquivo chamado "web.xml" e dois subdiretórios: "classes", com todas as classes, e "lib", com as bibliotecas utilizadas. Obviamente, abaixo do diretório "classes" podem haver subdiretórios para refletir o "path" relacionado aos "packages" Java (mais informações sobre packages podem ser obtidas em qualquer livro introdutório sobre a linguagem Java).

Utilizando como exemplo uma aplicação chamada RemoteIP, contendo o Servlet "RemoteIPServlet" do exemplo do capítulo 1 desse livro, uma estrutura possível abaixo do diretório webapps seria a seguinte:

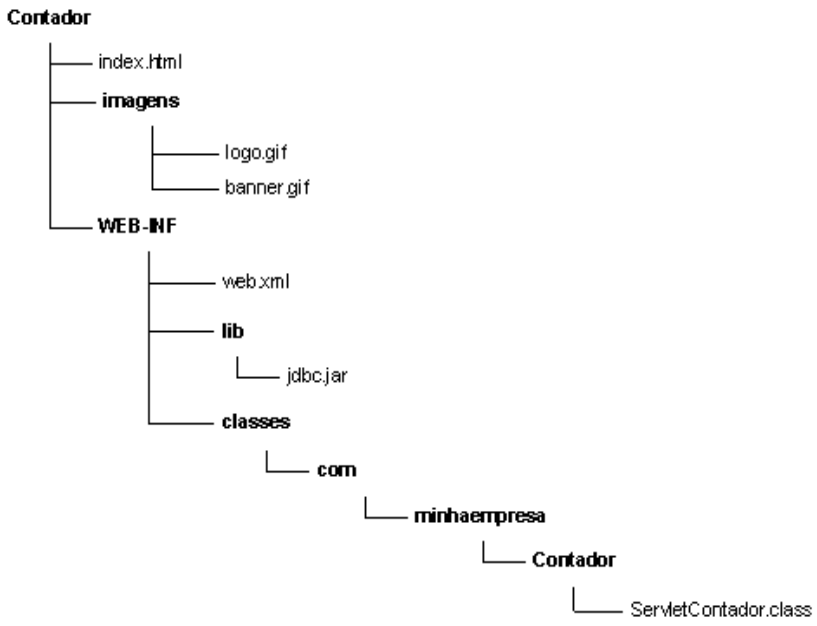

***Figura 2.3 – Exemplo de estrutura para aplicação "RemoteIP ".***

Obviamente, podem haver diversas aplicações instaladas, gerando diversas árvores de diretórios abaixo do diretório "webapps":

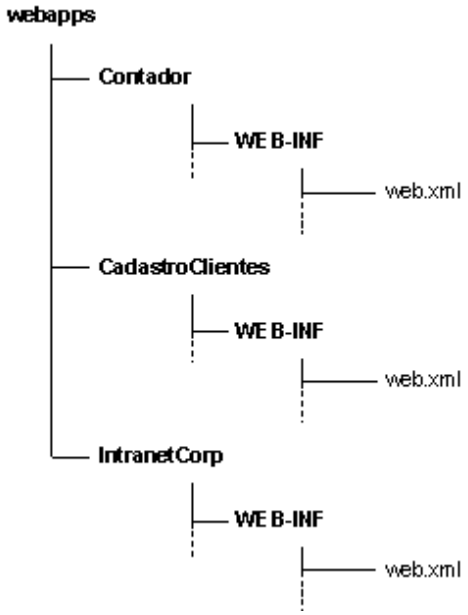


***Figura 2.4 – Exemplo de algumas aplicações instaladas abaixo do diretório webapps.***

Cada uma dessas aplicações é carregada pelo Servidor em um "Servlet Context" (Contexto do Servlet). Cada contexto dá à sua aplicação uma URL base, chamada de "Context Path" (Path do Contexto), e provê um ambiente comum para todos os Servlets da aplicação.

O path do contexto serve para que o Servidor possa mapear e distribuir as requisições recebidas para as diversas aplicações instaladas. No Apache Tomcat, o path do contexto coincide com o nome do subdiretório criado abaixo do "webapps".

Assim, no nosso exemplo, supondo que o endereço IP do servidor onde instalamos o Apache Tomcat é 192.168.0.1, teremos os acessos às URLs iniciadas por

http://192.168.0.1:8080/RemoteIP

direcionadas para a aplicação RemoteIP, os acessos às URLs iniciadas por

http://192.168.0.1:8080/CadastroClientes

direcionadas para a aplicação CadastroClientes, e assim por diante.

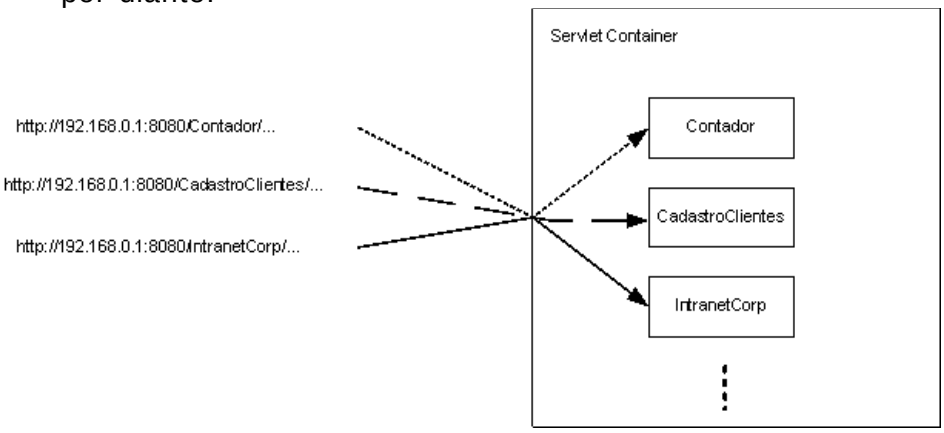


***Figura 2.5 – Exemplo de mapeamento de requisições para aplicações instaladas no Tomcat.***

Por fim, conforme o leitor pode ter reparado nos exemplos citados anteriormente, para cada aplicação há um "Deployment Descriptor": trata-se de um arquivo, chamado "web.xml" e localizado abaixo do diretório WEB-INF, e que contém informações de configuração da aplicação, tais como, parâmetros de inicialização, mapeamentos de Servlets, entre outros.

**Deployment Descriptor:** Arquivo XML com as informações de configuração de uma Aplicação Web. Esse arquivo fica abaixo do diretório "WEB-INF" e se chama "web.xml".

Um possível Deployment Descriptor para a aplicação "RemoteIP", por exemplo, seria o seguinte:

### Exemplo de Deployment Descriptor

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
```

```
<web- app>
<display- name>RemoteIP</display- name>
   <servlet>
      <servlet- name>RemoteIP</servlet- name>
      <servlet- class>RemoteIPServlet</servlet- class>
   </servlet>
      <servlet- mapping>
      <servlet- name>RemoteIP</servlet- name>
      <url- pattern>/RemoteIP</url- pattern>
   </servlet- mapping>
</web- app>
```

Como o Deployment Descriptor é composto de muitas seções, procuraremos apresentar as principais e suas respectivas funções, usando como exemplo a aplicação CadastroClientes mencionada anteriormente. Uma apresentação mais detalhada e aprofundada desse arquivo pode ser encontrada na própria especificação de Servlets.

Pressupomos, para essa apresentação um conhecimento mínimo de XML; caso você não tenha familiaridade com esse tipo de documento, sugerimos que você tente acompanhar os exemplos, e os utilize como templates para criar seus próprios Deployment Descritor's.

### Estrutura geral do Deployment Descriptor

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

<!DOCTYPE web- app
   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN"
   "http://java.sun.com/dtd/web- app_2_3.dtd">

<web- app>
.
.
.
</web- app>
```

Assim, como em qualquer documento XML, inicialmente são colocados os elementos de declaração do XML e do tipo de documento ("XML declaration" e "Document type

declaration"). Na figura anterior, esses elementos correspondem às 6 primeiras linhas listadas.

Em seguida, vem o elemento web-app: esse é o elemento root (raiz) desse XML, ou seja, deve haver somente um elemento web-app, e abaixo dele devem ficar todos os outros elementos do XML.

Os principais elementos abaixo do elemento root são os seguintes: display-name, context-param, session-config, welcome-file-list, error-page, servlet e servlet-mapping.

O elemento display-name deve conter um nome da aplicação a ser apresentado por ferramentas GUI de gerenciamento/desenvolvimento de Aplicações Web. Esse elemento é opcional, porém caso você decida utilizá-lo, é importante que haja somente um desses elementos por Deployment Descriptor.

**Exemplo de utilização do elemento "display-name"**

```
<display-name>Cadastro de Clientes</display-name>
```

O elemento "context-param" serve para que se possam definir parâmetros de inicialização do contexto da aplicação; esses parâmetros estarão disponíveis para todos os Servlets e páginas JSP da aplicação. Cada elemento presente deve conter o nome de um parâmetro e o seu valor correspondente. O desenvolvedor pode também optar por não utilizar nenhum desses elementos em seu XML.

**Exemplo de utilização do elemento "context-param"**

```
<context-param>
   <param-name>NomeBaseDados</param-name>
   <param-value>dbaplic</param-value>
</context-param>
<context-param>
   <param-name>IPBancoDados</param-name>
   <param-value>192.168.0.2</param-value>
</context-param>
```

O elemento seguinte, session-config, serve para que se possa especificar o período máximo, em minutos, de uma

sessão (esse recurso é explicado mais adiante no livro, no capítulo 6 – Sessões). Assim como o elemento display-name, esse elemento é opcional, mas caso o desenvolvedor opte por utilizá-lo, deve existir somente uma instância desse elemento no arquivo.

**Exemplo de utilização do elemento "session-config"**

```
<session-config>
    <session-timeout>15</session-timeout>
</session-config>
```

Os elementos welcome-file-list e error-page contém, respectivamente, a lista ordenada de páginas a serem utilizadas como "index" e as páginas a serem apresentadas em casos de erros "HTTP" ou exceções não tratadas pela aplicação. Esses dois elementos são opcionais, sendo que somente o primeiro admite uma instância por Deployment Descriptor.

**Exemplo de utilização dos elementos "welcome-file-list" e "error-page"**

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
<error-page>
    <error-code>404</error-code>
    <location>/404Error.html</location>
</error-page>
<error-page>
    <exception-
type>com.minhaempresa.exceptions.DBConnException</exception-
type>
    <location>/DBError.html</location>
</error-page>
```

De acordo com o que é apresentado na listagem anterior, se tomarmos como exemplo nossa aplicação CadastroClientes, quando é feito um acesso a URL http://192.168.0.1:8080/CadastroClientes/, o Servidor tentará retornar a página "index.html", conforme

especificado na lista do welcome-file-list. Caso essa página não exista, o Servidor tentará utilizar a página "index.jsp".

A figura anterior também demonstra a utilização do elemento error-page duas vezes: a primeira vez para mapear erros HTTP 404 (página não encontrada) a uma página de erro-padrão, e a segunda, para mapear exceptions "com.minhaempresa.exceptions.DBConnException" a uma outra página de erro.

Os últimos dois elementos, servlet e servlet-mapping, servem para definir, respectivamente, os Servlets da aplicação, com seus respectivos parâmetros, e os mapeamentos de URLs a cada Servlet da aplicação.

Cada elemento servlet, por sua vez, é composto dos seguintes elementos:

servlet-name: deve conter o nome do Servlet.

servlet-class: deve conter o nome da classe (incluindo a informação sobre o package, se existir).

init-param: deve conter um parâmetro de inicialização do Servlet; pode haver nenhum, somente um, ou mais de um elemento deste tipo para cada Servlet.

load-on-startup: deve conter um inteiro positivo indicando a ordem de carga deste Servlet em relação aos outros Servlets da aplicação, sendo que inteiros menores são carregados primeiro; se este elemento não existir, ou seu valor não for um inteiro positivo, fica a cargo do Servlet Container decidir quando o Servlet será carregado (possivelmente, no instante em que chegar chegar a primeira requisição a esse Servlet).

### Exemplo de utilização do elemento "servlet"

```
<servlet>
    <servlet-name>ProcessaCadastro</servlet-name>
    <servlet-
class>com.minhaempresa.CadastroClientes.ProcCadastro</servlet-
class>
    <init-param><param-name>Email.ServidorSMTP</param-
name><param-
```

```
        value>smtp.minhaempresa.com.br</param- value></init-
param>
    <init- param><param- name>Email.Remetente</param-
name><param-
        value>site@minhaempresa.com.br</param- value></init-
param>
    <init- param><param- name>Email.Destinatario</param-
name><param-
        value>vendas@minhaempresa.com.br</param- value></init-
param>
    <init- param><param- name>Email.Assunto</param-
name><param- value>Novo cadastro de
        cliente</param- value></init- param>
    <load- on- startup>0</load- on- startup>
</servlet>
```

Por fim, um elemento servlet- mapping contém um nome de Servlet, conforme definido em servlet- name, e um padrão da URL do Servlet no servidor (URL pattern).

**Exemplo de utilização do elemento "servlet-mapping"**

```
<servlet- mapping>
    <servlet- name>ProcessaCadastro</servlet- name>
    <url- pattern>/Processamento</url- pattern>
</servlet- mapping>
```

No exemplo anterior, todos as requisições com URLs iniciadas por /CadastroClientes/Processamento/ serão mapeadas para o Servlet cujo nome é ProcessaCadastro.

Outros mapeamentos interessantes podem ser obtidos através de padrões de URL do tipo *.<extensão>, como por exemplo, *.wm ou *.pdf, de maneira que o acessos a todas as URLs com o sufixo indicado sejam tratados por um mesmo Servlet. Um último exemplo de mapeamento interessante diz respeito ao padrão /, que define o Servlet default para todos os acessos que não se encaixarem em nenhum outro padrão.

Juntando então todos os elementos apresentados anteriormente, temos o Deployment Descriptor de exemplo apresentado a seguir:

## Exemplo de Deployment Descriptor completo para a aplicação “CadastroClientes”

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

<!DOCTYPE web-app
   PUBLIC “-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN”
   “http://java.sun.com/dtd/web-app_2_3.dtd”>

<web-app>
    <display-name>Cadastro de Clientes</display-name>
    <context-param>
        <param-name>NomeBaseDados</param-name>
        <param-value>dbaplic</param-value>
    </context-param>
        <context-param>
        <param-name>IPBancoDados</param-name>
        <param-value>192.168.0.2</param-value>
    </context-param>
    <session-config>
        <session-timeout>15</session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
    <error-page>
        <error-code>404</error-code>
        <location>/404Error.html</location>
    </error-page>
    <error-page>
        <exception-
type>com.minhaempresa.exceptions.DBConnException</exception-
type>
        <location>/DBError.html</location>
    </error-page>
    <servlet>
        <servlet-name>ProcessaCadastro</servlet-name>
```

```
        <servlet-
class>com.minhaempresa.CadastroClientes.ProcCadastro</servlet-
class>
        <init- param><param- name>Email.ServidorSMTP</param-
name><param- value>
            smtp.minhaempresa.com.br</param- value></init-
param>
        <init- param><param- name>Email.Remetente</param-
name><param- value>
            site@minhaempresa.com.br</param- value></init- param>
        <init- param><param- name>Email.Destinatario</param-
name><param- value>
            vendas@minhaempresa.com.br</param- value></init-
param>
        <init- param><param- name>Email.Assunto</param-
name><param- value>
            Novo cadastro de cliente</param- value></init- param>
        <load- on- startup>0</load- on- startup>
    </servlet>
    <servlet>
        <servlet- name>FormularioCadastro</servlet- name>
        <servlet-
class>com.minhaempresa.CadastroClientes.FormCadastro</servlet-
class>
    </servlet>
    <servlet- mapping>
        <servlet- name>ProcessaCadastro</servlet- name>
        <url- pattern>/Processamento</url- pattern>
    </servlet- mapping>
    <servlet- mapping>
        <servlet- name>FormularioCadastro</servlet- name>
        <url- pattern>/Formulario</url- pattern>
    </servlet- mapping>
</web- app>
```

# Capítulo 3

# Servlets – características básicas

Nesse capítulo, exploramos as características básicas de Servlets, mostrando o funcionamento e sua interação com o Servlet Container. Implementamos também nossos primeiros Servlets de exemplo.

## 3.1 Biblioteca e documentação

Antes que você possa iniciar o desenvolvimento de seus Servlets, é imprescindível que você tenha disponível a biblioteca de Servlets Java (normalmente, um arquivo chamado servlet.jar; se você estiver utilizando o Apache Tomcat, você pode encontrar esse arquivo abaixo do diretório de instalação do Tomcat, no subdiretório common\lib). Essa biblioteca contém todas as classes e interfaces necessárias para o desenvolvimento de Servlets, e deve estar contida em seu classpath.

Outro item importante, embora não imprescindível, é a documentação da API de Servlets. Por meio dessa documentação, você poderá verificar todos as classes, com seus respectivos métodos e variáveis, com os quais você poderá contar durante o processo de desenvolvimento. Essa documentação pode ser obtida diretamente do site oficial do Java (http://java.sun.com).

## 3.2 O protocolo HTTP

Embora Servlets possam ser utilizados não só para o desenvolvimento de aplicações HTTP, a maior parte das aplicações desenvolvidas são destinadas a esse fim. Sendo assim, vale a pena estudar um pouco mais a fundo o funcionamento e características desse protocolo.

O protocolo HTTP é utilizado na navegação nas páginas da Internet: quando você abre uma janela de um browser, acessa uma página Web e navega em seus links, você está, na verdade, utilizando esse protocolo para visualizar, em sua máquina, o conteúdo que está armazenado em servidores remotos.

O HTTP é um protocolo stateless de comunicação cliente-servidor: o cliente envia uma requisição para o servidor, este processa a requisição e devolve uma resposta para o cliente, sendo que, a princípio, nenhuma informação é mantida no servidor em relação às requisições previamente recebidas.

Assim, quando digitamos o endereço de uma página em um browser Web, estamos gerando uma requisição a um servidor, que irá, por sua vez, devolver para o browser o conteúdo da página HTML requisitada.

A requisição enviada por um cliente deve conter, basicamente, um comando (também chamado de método), o endereço de um recurso no servidor (também chamado de “path”) e uma informação sobre a versão do protocolo HTTP sendo utilizado.

Supondo, por exemplo, que utilize-se o método GET, o path /index.html e a versão 1.0 do protocolo HTTP (o que equivale a digitar um endereço http://<endereço de algum servidor>/index.html em um browser), temos a seguinte requisição enviada:

**Exemplo de requisição HTTP**

```
GET /index.html HTTP/1.0
```

Existem diversos métodos HTTP que podem ser especificados em requisições, sendo os mais comuns o método GET, normalmente utilizado para obter o conteúdo de um arquivo no servidor, e o método POST, utilizado para enviar dados de formulários HTML ao servidor. Além desses métodos, o protocolo HTTP 1.0 admite também o método HEAD, que permite que o cliente obtenha somente os headers da resposta; já o protocolo HTTP versão 1.1 admite os seguintes métodos:

"PUT": transfere um arquivo do cliente para o servidor

"DELETE": remove um arquivo do servidor

"OPTIONS": obtém a lista dos métodos suportados pelo servidor

"TRACE": retorna o conteúdo da requisição enviada de volta para o cliente

Além do método, path e versão, uma requisição pode conter parâmetros adicionais, chamados "headers". Dois headers comuns são, por exemplo, o header User-Agent, que contém informações sobre o cliente que está gerando a requisição (tipo, versão do browser etc.) e o header Accept, que serve para especificar os tipos de recursos aceitos pelo cliente para a requisição enviada.

**Exemplo de requisição HTTP com headers**

```
GET /index.html HTTP/1.0
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt)
Accept: text/html
```

Uma vez processada a requisição, o servidor, por sua vez, manda uma resposta para o cliente, sendo que essa resposta também tem um formato predeterminado: a primeira linha contém informações sobre a versão do protocolo, um código de status da resposta e uma mensagem associada a esse status; em seguida são enviados também headers (com informações do servidor que gerou a resposta, por exemplo); e finalmente, é enviado o conteúdo, propriamente dito, da resposta.

**Exemplo de resposta HTTP com headers**

```
HTTP/1.1 200 OK
Server: Apache/1.3.26 (Unix)
Last-Modified: Sun, 22 Dec 2002 17:47:59 GMT
Content-Type: text/html
Content-Length: 30

<HTML>
<BODY>
</BODY>
</HTML>
```

Assim, no exemplo anterior, o código de status 200 indica que houve sucesso no atendimento da requisição enviada pelo cliente, os headers indicam o tipo, tamanho e data e hora de última modificação do conteúdo requisitado, e por fim, temos uma página HTML em branco, com o conteúdo propriamente dito.

Outros códigos de status bastante comuns são o 404, que indica que o recurso não foi localizado no servidor, e o código 500, que indica que houve erro no processamento da requisição enviada.

## 3.3 Hierarquia de um Servlet

Conforme descrito anteriormente, um Servlet nada mais é que uma classe Java que obedece a uma estrutura bem definida. Em especial, essa classe deve implementar a interface javax.servlet.Servlet.

Existem duas classes, na biblioteca de Servlets, que implementam essa interface: javax.servlet.GenericServlet e sua sub-classe, javax.servlet.http.HttpServlet. A classe GenericServlet, como o próprio nome indica, serve para atender requisições genéricas (utilizando qualquer protocolo), e a classe HttpServlet, para atender requisições HTTP.

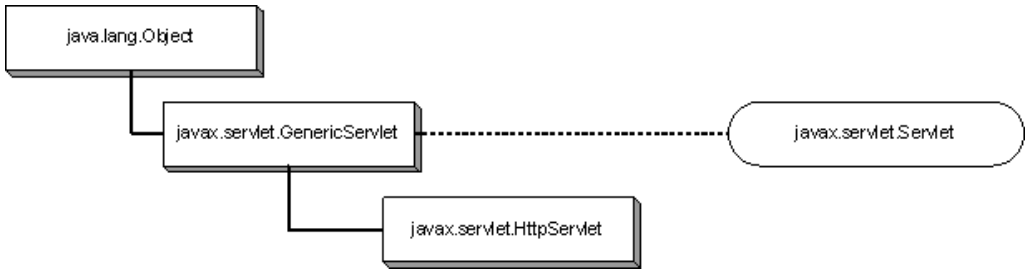


***Figura 3.1 – Hierarquia de classes associadas a um Servlet.***

No desenvolvimento do Servlet de nossa aplicação exemplo CadastroClientes, temos assim a seguinte declaração de classe:

**Declaração do Servlet ProcCadastro**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Servlet para processamento do cadastro de novos clientes
public class ProcCadastro extends HttpServlet {
    ...
}
```

# 3.4 Ciclo de vida de um Servlet

Todo Servlet segue, por outro lado, um ciclo de vida composto de 3 fases: inicialização, atendimento de requisições e finalização.

A inicialização ocorre quando o Servlet Container carrega o Servlet: se o parâmetro load-on-startup, do Deployment Descriptor (vide seção 2.2), estiver presente e contiver um inteiro positivo, essa carga ocorre quando o próprio servidor é iniciado; caso contrário, essa carga ocorre quando é recebida a primeira requisição a ser mapeada para a aplicação que contém o Servlet.

Após a inicialização, o Servlet pode atender requisições. Assim, enquanto o servidor estiver ativo, e a aplicação que

contem o Servlet estiver carregada, este permanecerá na fase 2 de seu ciclo.

Um ponto importante com relação a essa fase, e que na verdade constitui uma vantagem da tecnologia de Servlets e páginas JSP com relação a outras tecnologias, é que o fato do Servlet permanecer carregado permite que dados armazenados em variáveis de classe persistam ao longo das diversas requisições recebidas. Assim, é possível manter um pool de conexões ao banco de dados, por exemplo, de maneira que não seja necessário iniciar e estabelecer uma nova conexão ao banco de dados a cada requisição recebida.

Finalmente, quando o servidor é finalizado, ou quando a aplicação é tornada inativa pelo Servlet Container, o Servlet é finalizado.


***Figura 3.2 – Ciclo de vida de um Servlet.***

Cada uma das fases se traduz, na verdade, em métodos do Servlet que são chamados pelo Servlet Container nos diversos instantes do ciclo.

Apresentamos, nas seções subsequentes, os métodos relacionados às fases de inicialização, finalização e de atendimento de requisições.

## 3.5 Inicialização

Conforme apresentado nos parágrafos anteriores, a inicialização do Servlet ocorre no instante em que é feita a carga da aplicação pelo Servlet Container.

Nesse instante, o Servlet Container executa o método “init” do Servlet, dando chance ao Servlet de executar quaisquer passos necessários para sua inicialização, tais como:

1) leitura de parâmetros de configuração

2) inicialização de variáveis de classe (variáveis estáticas)

3) inicialização de conexões ao banco de dados, etc.

Assim, podemos ter implementado em nosso Servlet "ProcCadastro", por exemplo:

**Inicialização do Servlet "ProcCadastro"**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


// Servlet para processamento do cadastro de novos clientes: a cada
novo cadastro bem sucedido,
// envia email com os dados do cadastro
public class ProcCadastro extends HttpServlet {
    ...
    public void init () {
        ...
    }
    ...
}
```

As assinaturas do método init() são:

**Assinaturas do método "init ()"**

```
public void init();
public void init( javax.servlet.ServletConfig p_config );
```

Conforme pode ser visto, o método init() admite duas assinaturas, sendo que em uma delas, é recebido como parâmetro um objeto da classe javax.servlet.ServletConfig: através desse objeto, o Servlet pode obter os parâmetros de inicialização do Servlet, contidos no Deployment Descriptor (veja seção 2.2). Por outro lado, caso você opte por implementar o método init() sem nenhum parâmetro, é possível também obter uma referência para o objeto ServletConfig por meio da chamada getServletConfig() da própria classe javax.servlet.GenericServlet (a qual nossa classe estende).

**Assinatura do método "getServletConfig ()"**

```
public javax.servlet.ServletConfig getServletConfig ();
```

Para obter um parâmetro de inicialização do Servlet usando o objeto ServletConfig, deve-se utilizar o método getInitParameter(), passando como parâmetro o nome do parâmetro que se deseja obter.

### Assinatura do método "getInitParameter ()"

```
public java.lang.String getInitParameter( java.lang.String
p_parameterName );
```

Temos, a seguir, um exemplo de uso desse método:

### Exemplo de uso do método "getInitParameter()" de um objeto "ServletConfig"

```
public void init(ServletConfig p_servletConfig) throws
ServletException {
    super.init(p_servletConfig);


    String l_servidorSMTP = p_servletConfig.getInitParameter
("Email.ServidorSMTP");
    if(l_servidorSMTP != null) {
        ...
    }
}
```

Obviamente o método getInitParameter() pode retornar um valor nulo caso o parâmetro inicial a ser obtido não tenha sido definido, e por isso é importante que você faça a verificação do String retornado pela chamada do método antes de utilizá-lo.

É possível também, a partir de um objeto da classe ServletConfig, percorrer a lista dos parâmetros de inicialização do Servlet, bastando utilizar o método getInitParameterNames().

### Assinatura do método "getInitParameterNames ()"

```
public java.util.Enumeration getInitParameterNames();
```

Temos, a seguir, um exemplo de uso deste outro método:

### Exemplo de uso do método "getInitParameterNames ()" de um objeto "ServletConfig"

```
public void init(ServletConfig p_servletConfig) throws
ServletException {
    super.init(p_servletConfig);

    Enumeration l_parameterNames =
p_servletConfig.getInitParameterNames ();
    if(l_parameterNames != null)
        while(l_parameterNames.hasMoreElements ())
        {
            String l_parameterName = (String)
l_parameterNames.nextElement ();
            String l_parameterValue = p_servletConfig.getInitParameter
(l_parameterName) ;
            ...
        }
    }
}
```

Assim, em nossa aplicação de exemplo CadastroClientes, podemos implementar o método de inicialização do Servlet ProcCadastro como:

### Inicialização do Servlet "ProcCadastro"

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Servlet para processamento do cadastro de novos clientes:
// a cada novo cadastro bem sucedido, envia email com os dados do
cadastro
public class ProcCadastro extends HttpServlet {

    // Servidor SMTP a ser usado para o envio de email
    private static String _ServidorSMTP = null;

    // Remetente, destinatário e assunto do email a ser enviado a
cada cadastro
    private static String _Remetente = null, _Destinatario = null,
_Assunto = null;

    public void init(ServletConfig p_servletConfig) throws
ServletException {
```

```
        super.init(p_servletConfig);
        // Recuperando os parâmetros de inicialização do Servlet
        _ServidorSMTP = p_servletConfig.getInitParameter
("Email.ServidorSMTP");
        _Remetente = p_servletConfig.getInitParameter
("Email.Remetente");
        _Destinatario = p_servletConfig.getInitParameter
("Email.Destinatario");
        _Assunto = p_servletConfig.getInitParameter("Email.Assunto");
        ...
    }

}
```

Outro uso comum para o método de inicialização do Servlet é para o despacho de um ou mais Threads, que deverão ser executados durante o período em que o Servlet permanecer carregado. Assim, um serlvet pode, por exemplo, iniciar um Thread que irá verificar continuamente a disponibilidade de uma conexão com o banco de dados, independente de qualquer requisição que receba. Este Servlet de exemplo é apresentado a seguir:

### Inicialização do Servlet "VerificaConBD"

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Servlet para verificar conexão com banco de dados: lança thread
que verifica status da conexão periodicamente
public class VerificaConDB extends HttpServlet implements Runnable
{

    // Referência ao thread que irá fazer a verificação da conexão
    Thread _ThreadVerif = null;

    // Inicialização do Servlet
    public void init(ServletConfig p_servletConfig) throws
ServletException {
        super.init(p_servletConfig);
```

```
        // Lançando Thread ...
        _ThreadVerif = new Thread(this);
        _ThreadVerif.start ();
        ...
    }

    // Execução do thread
    public void run() {
        while(_ThreadVerif != null) {
            if(!ConBD.OK ()) {
                ...
            }
        }
    }
    ...

}
```

Uma observação importante com relação a esse processo de inicialização é que o Servlet somente poderá receber requisições após a conclusão de seu processo de inicialização.

O desenvolvedor pode, por outro lado, indicar que esse processo de inicialização não foi bem sucedido, através do lançamento da exceptions ServletException ou UnavailableException; nestes casos, o Servlet Container irá deixar o Servlet em um estado inativo, ou seja, sem poder receber requisições. A exception UnavailableException, em particular, pode receber como parâmetro em seu construtor, um número de segundos com uma estimativa de quanto tempo o Servlet deverá ficar inativo.

### Exemplo de uso da exceção "UnavailableException" para indicar fracasso na inicialização

```
public void init(ServletConfig p_servletConfig)
        throws ServletException, UnavailableException {
    super.init(p_servletConfig);
    ...
    // Recuperando e validando parâmetros de inicialização do
Servlet
```

```
    _ServidorSMTP = p_servletConfig.getInitParameter
("Email.ServidorSMTP");
    _Remetente = p_servletConfig.getInitParameter
("Email.Remetente");
    _Destinatario = p_servletConfig.getInitParameter
("Email.Destinatario");
    _Assunto = p_servletConfig.getInitParameter("Email.Assunto");
    if((_ServidorSMTP == null) || (_Remetente == null) || (_Assunto
== null))
        throw new UnavailableException("Erro: parâmetros de
inicialização não
            encontrados!");
```

No caso de você não ter percebido, todos os métodos init() apresentados até agora nos exemplos foram declarados de maneira a possibilitar o lançamento do exception ServletException: isso é necessário devido à chamada do método super.init (), que pode, por si, lançar essa exceção para indicar problemas em sua execução.

## 3.6 A classe "ServletContext"

Além da referência ao objeto da classe ServletConfig recebido como parâmetro na inicialização, o Servlet pode obter também uma referência a um objeto da classe javax.servlet.ServletContext através do método getServletContext (), herdado da classe GenericServlet.

### Assinatura do método "getServletContext ()"

```
public javax.servlet.ServletContext getServletContext();
```

Esse objeto ServletContext contém os atributos e informações sobre o contexto em que o Servlet está sendo executado e, sendo assim, é compartilhado por todos os Servlets que fazem parte da Aplicação Web.

Analogamente ao que acontece com o objeto ServletConfig, existem métodos para recuperar os parâmetros iniciais do contexto definidos no "DeploymentDescriptor" (vide seção 2.2).

### Assinatura dos métodos "getInitParameterNames ()" e "getInitParameter ()"

```
public java.util.Enumeration getInitParameterNames();
public java.util.Enumeration getInitParameter(java.lang.String
p_parameterName);
```

Por outro lado, é importante fazer a distinção entre os parâmetros iniciais do Servlet e os parâmetros iniciais do contexto, lembrando sempre que esses parâmetros são definidos em seções distintas do "DeploymentDescriptor".

**Exemplo de uso do método "getInitParameter" do objeto "ServletConfig" e do objeto "ServletContext"**

```
public void init(ServletConfig p_servletConfig) throws
ServletException {
    super.init(p_servletConfig);


    // Recuperando parâmetros de execução do Servlet
    String l_paramExec = p_servletConfig.getInitParameter
("ParametroExecucao");


    // Se não existir tal parâmetro do Servlet, tentamos como
parâmetro do contexto
    if(l_paramExec == null) {
        ServletContext l_servletContext = getServletContext ();
        l_paramExec = l_servletContext.getInitParameter
("ParametroExecucao");
    }
    ...
}
```

Além dos parâmetros de inicialização do contexto do Servlet, podemos usar esse objeto para atribuir e recuperar atributos que serão compartilhados por todos os Servlets do contexto. Assim, temos os métodos:

**Assinatura dos métodos "getAttribute ()", "getAttributeNames ()", "removeAttribute()" e "setAttribute()"**

```
public java.lang.Object getAttribute(java.lang.String p_attributeName);
public java.util.Enumeration getAttributeNames();
public void removeAttribute(java.lang.String p_attributeName);
```

```
public void setAttribute(java.lang.String p_attributeName,
        java.lang.Object p_attributeValue);
```

No exemplo a seguir, temos dois Servlets que fazem parte de uma mesma Aplicação Web e que utilizam os atributos do contexto para indicar falhas em suas respectivas inicializações.

### Exemplo de uso dos métodos “getAttribute” e “setAttribute” do objeto “ServletContext”

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Primeiro Servlet do exemplo
public class PrimeiroServlet extends HttpServlet {

    public void init(ServletConfig p_servletConfig) throws
ServletException {
        super.init(p_servletConfig);

        // Carregando o parâmetro inicial
        boolean l_sucesso = true;
        String l_paramInicial =
                p_servletConfig.getInitParameter
(“ParamPrimeiroServlet”);
        if(l_paramInicial == null) l_sucesso = false;
        ...
        // Usando o atributo de contexto para indicar status da
inicialização
        ServletContext l_servletContext = getServletContext ();
        l_servletContext.setAttribute(“PrimeiroServlet”, new Boolean
(l_sucesso));

        // Verificando status do segundo Servlet
        Boolean  l_statusSegundoServlet = (Boolean)
                l_servletContext.getAttribute(“SegundoServlet”);
        while(l_statusSegundoServlet == null) {
            Thread.sleep(5000); // Espera 5 segundos e verifica o
status novamente
            l_statusSegundoServlet = (Boolean)
                l_servletContext.getAttribute(“SegundoServlet”);
```

```
        }

        // Se houve fracasso na inicialização deste Servlet ou do
segundo,
        // lançamos uma exceção
        if((l_sucesso == false) || (l_statusSegundoServlet.booleanValue
() == false))
            throw new UnavailableException("Erro: os dois Servlets não
puderam ser
            carregados com sucesso!");
    }

    ...
}

// Segundo Servlet do exemplo
public class SegundoServlet extends HttpServlet {

    public void init(ServletConfig p_servletConfig) throws
ServletException {
        super.init(p_servletConfig);

        // Carregando o parâmetro inicial
        boolean l_sucesso = true;
        String l_paramInicial =
                p_servletConfig.getInitParameter
("ParamSegundoServlet");
        if(l_paramInicial == null) l_sucesso = false;
        ...
        // Usando o atributo de contexto para indicar status da
inicialização
        ServletContext l_servletContext = getServletContext ();
        l_servletContext.setAttribute("SegundoServlet", new Boolean
(l_sucesso));

        // Verificando status do segundo Servlet
        Boolean  l_statusPrimeiroServlet = (Boolean)
                 l_servletContext.getAttribute("PrimeiroServlet");
        while(l_statusPrimeiroServlet == null) {
            Thread.sleep(5000); // Espera 5 segundos e verifica o
status novamente
```

```
            l_statusPrimeiroServlet = (Boolean)
                  l_servletContext.getAttribute("PrimeiroServlet");
        }

        // Se houve fracasso na inicialização deste Servlet ou do
primeiro,
        // lançamos uma exceção
        if((l_sucesso == false) || (l_statusPrimeiroServlet.booleanValue
() == false))
            throw new UnavailableException("Erro: os dois Servlets não
puderam
            ser carregados!");
    }

    ...
}
```

Por fim, há um outro método da classe ServletContext que vale a pena conhecer: o método log() permite que você adicione mensagens em um arquivo de log do Servidor de Aplicações. Você poderá utilizar esse método para depurar seus Servlets, gerar alertas de problemas na sua execução etc.

### Assinatura dos método "log ()"

```
public void log(java.lang.String p_msg);
```

Em alguns Servidores de Aplicação você pode também tentar usar as saídas-padrão ("System.out", "System.err") para gerar suas mensagens de log, porém, é muito mais interessante que você use o método anterior, de maneira que o ServletContainer possa separar as suas mensagens em um log diferenciado. No caso específico do Tomcat, as mensagens geradas por meio do método log() são adicionadas a um arquivo de log normalmente chamado localhost_log.ext.txt, onde ext é uma extensão contendo a data corrente.

### Exemplo de uso do método "log ()" (da classe "ServletContext")

```
import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;

// Servlet para teste de geração de mensagens de log
public class TesteLog extends HttpServlet {

    // Valor default para parametro inicial do Servlet
    private static final String _ValorDefaultParamInicial = "";

    public void init(ServletConfig p_servletConfig) throws
ServletException {
        super.init(p_servletConfig);

        // Recuperando os parâmetros de inicialização do Servlet
        String l_paramInicial = p_servletConfig.getInitParameter
("ParametroInicial");
        if(l_paramInicial == null) {
            ServletContext l_context =
p_servletConfig.getServletContext ();
            l_context.log("Aviso: não foi possível se carregar o
parâmetro inicial;
                atribuindo valor default ...");
            l_paramInicial = _ValorDefaultParamInicial;
        }
        ...
    }

    ...
}
```

## 3.7 Finalização

A finalização de um Servlet deve ser tratada através da implementação do método destroy: no instante em que o Servlet é "descarregado", seu método destroy, se tiver sido implementado, é chamando, permitindo a execução de rotinas de finalização (como por exemplo, o encerramento de conexões com bancos de dados, finalização de threads que tenham sido lançados etc.).

A assinatura do método destroy() é a seguinte:

### Assinatura do método “destroy ()”

```
public void destroy();
```

Utilizando nosso exemplo de cadastro de clientes, temos a seguir um exemplo de nosso Servlet VerificaConBD com os métodos init() e destroy() implementados:

### Inicialização do Servlet “VerificaConBD”

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Servlet para verificar conexão com banco de dados: lança thread
que verifica status da conexão periodicamente
public class VerificaConDB extends HttpServlet implements Runnable
{

    // Referência ao thread que irá fazer a verificação da conexão
    Thread _ThreadVerif = null;

    // Inicialização do Servlet
    public void init(ServletConfig p_servletConfig) throws
ServletException {
        super.init(p_servletConfig);

        // Lançando Thread ...
        _ThreadVerif = new Thread(this);
        _ThreadVerif.start ();
        ...
    }

    // Execução do thread
    public void run() {
        while(_ThreadVerif != null) {
            if(!ConBD.OK ()) {
                ...
            }
        }
    }
```

```
    // Finalização do Servlet
    public void destroy() {
        _ThreadVerif = null;
    }

    ...
}
```

No exemplo, o método destroy() serve para indicar que o thread deve ser finalizado: a atribuição do valor null à variável _ThreadVerif faz com que o looping principal do método run() seja finalizado.

## 3.8 Atendimento de Requisições

Conforme descrito anteriormente na seção sobre o ciclo de vida de um Servlet, entre as fases de inicialização e finalização, existe uma fase onde o Servlet irá, efetivamente, atender as requisições recebidas.

No caso da classe GenericServlet, que é a classe utilizada para Servlets genéricos (classe-pai para a classe HttpServlet, que atende requisições HTTP), o método relacionado a essa fase é o método service ().

**Assinatura do método "service ()"**

```
public void service(javax.servlet.ServletRequest p_request,
        javax.servlet.ServletResponse p_response);
```

Assim, para cada requisição recebida de um cliente, o ServletContainer efetua uma chamada a esse método service(...) do Servlet; os parâmetros desse método são referências para um objeto que encapsula a requisição recebida e para um objeto que encapsula a resposta que deverá ser encaminhada para o cliente.

Por outro lado, como você normalmente estará desenvolvendo Servlets HTTP, dificilmente você terá que implementar esse método; em vez disso, para classes que extendam a classe HttpServlet, você deverá implementar um ou mais dos seguintes métodos: doDelete, doGet, doOptions, doPost, doPut ou doTrace.

**Assinatura dos métodos de atendimento de requests da classe HttpServlet**

```
public void doGet(javax.servlet.http.HttpServletRequest p_request,
        javax.servlet.http.HttpServletResponse p_response);
public void doPost(javax.servlet.http.HttpServletRequest p_request,
        javax.servlet.http.HttpServletResponse p_response);
public void doDelete( javax.servlet.http.HttpServletRequest p_request,

        javax.servlet.http.HttpServletResponse p_response);
public void doPut(javax.servlet.http.HttpServletRequest p_request,
        javax.servlet.http.HttpServletResponse p_response);
public void doOptions(javax.servlet.http.HttpServletRequest
p_request,
        javax.servlet.http.HttpServletResponse p_response);
public void doTrace(javax.servlet.http.HttpServletRequest p_request,
        javax.servlet.http.HttpServletResponse p_response);
```

Quando uma requisição HTTP é recebida por uma classe que estende HttpServlet, seu método service() é chamado, sendo que a implementação default desse método irá chamar a função doXXX () correspondente ao método da requisição recebida. Ou seja, caso uma requisição com método GET, por exemplo, seja recebida (vide seção 3.2, sobre o protocolo HTTP), o método doGet() implementado por você será chamado.

Geralmente, desenvolvedores de Servlets implementam somente os métodos doGet() e doPost (); os métodos restantes só são implementados em casos especiais, e requerem um conhecimento mais avançado por parte do desenvolvedor. Por ora, estaremos utilizando o método doGet(); nos capítulos seguintes demonstraremos com mais detalhes alguns dos outros métodos (principalmente, o método doPost()).

Um exemplo simples de implementação do método doGet() pode ser observado a seguir:

**Servlet "HelloWorld"**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
// Servlet para página HelloWorld
public class HelloWorld extends HttpServlet {

    // Atendimento de requisições HTTP com método GET
    public void doGet(HttpServletRequest p_request,
            HttpServletResponse p_response) throws IOException {
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");
        l_pw.println("Hello World!");
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }
}
```

## 3.9 Concorrência no atendimento de requisições

Durante o ciclo de vida de um Servlet, o ServletContainer irá fazer a carga de um Servlet instanciando um único objeto e chamando seu método init(); a finalização também é efetuada chamando o método destroy desse objeto.

Na fase de atendimento de requisições, por outro lado, o método service() (e, consequentemente, os métodos doXXX (), no caso de Servlets HTTP), são chamados na medida em que são recebidas as requisições, ou seja, pode haver, em um determinado instante, um ou mais threads do ServletContainer executando métodos service() simultaneamente.

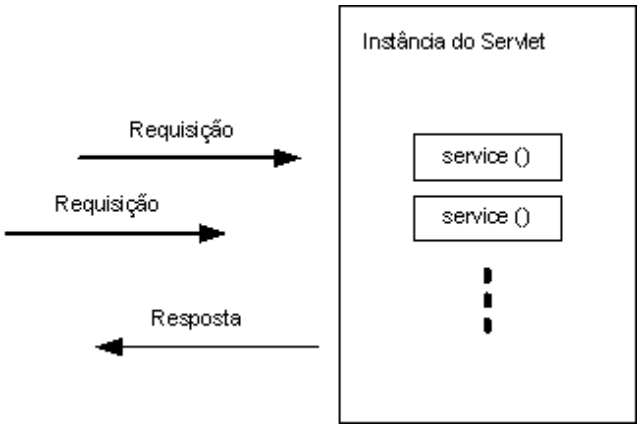


***Figura 3.3 – Concorrência no atendimento de requisições.***

Por isso é muito importante que você se preocupe com acesso a variáveis de instância ou classe e concorrência no seu desenvolvimento (maiores detalhes sobre esses tópicos podem ser obtidos, por exemplo, no livro *Aprendendo Java 2*, da Editora Novatec).

Nesse sentido, uma opção para garantir a execução livre de problemas de concorrência é a implementação da interface SingleThreadModel em seu Servlet.

Essa interface não define, na verdade, novos métodos ou variáveis de classe, ela serve somente para indicar ao ServletContainer que o atendimento de requisições do Servlet em questão deve ser feito de forma a serializar as chamadas ao método service(). Ou seja, somente uma requisição será atendida por vez pelo seu Servlet.

### Exemplo de Servlet que implementa a interface "SingleThreadModel"

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


// Servlet simples que retorna página HTML com o número de
// requisições recebidas até o momento
public class ServletContador extends HttpServlet implements
SingleThreadModel {
```

```
    // Contador das requisições recebidas
    private int _Contador = 0;

    public void doGet(HttpServletRequest p_request,
HttpServletResponse
            p_response) throws IOException {
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");
        l_pw.println("Requisições recebidas: " + Integer.toString(++
_Contador));
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }

}
```

No exemplo anterior, o Servlet utiliza uma variável de instância _Contador e, por isso, há a necessidade de se preocupar com a concorrência no acesso a esta variável.

Imagine, por exemplo, que esse Servlet não implemente a interface SingleThreadModel, e receba duas requisições simultaneamente: o primeiro doGet() poderia incrementar o contador, seguido do segundo doGet() incrementando o contador, seguido dos dois threads, cada um por sua vez, imprimindo o valor do contador. Nessa situação, o mesmo valor de contador seria impresso nas duas páginas HTML resultantes.

**Resultado da primeira requisição**

```
<HTML><BODY>
Requisições recebidas: 2
</BODY></HTML>
```

**Resultado da segunda requisição**

```
<HTML><BODY>
Requisições recebidas: 2
</BODY></HTML>
```

Obviamente a implementação dessa interface tem um custo na performance de execução do Servlet, pois, no

Servlet anterior, por exemplo, não só o acesso a variável _Contador é serializado, mas a execução do método doGet () como um todo. Em particular, a execução dos códigos de geração do header e do footer HTML não precisariam ser serializados, mas são.

Por isso, em vez de implementar esta interface, na maioria das vezes é mais conveniente implementar diretamente um código de sincronização nos trechos que precisam ser serializados. O Servlet ServletContador, por exemplo, poderia ser implementado da seguinte forma (com exatamente o mesmo resultado):

**Exemplo de Servlet que substitui a implementação da interface "SingleThreadModel" por código de sincronização**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Servlet simples que retorna página HTML com o número de
// requisições recebidas até o momento
public class ServletContador extends HttpServlet {

    // Contador das requisições recebidas
    private int _Contador = 0;

    public void doGet(HttpServletRequest p_request,
HttpServletResponse
            p_response) throws IOException {
        // Imprimindo cabeçalho
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");

        // Armazenando valor do contador em uma variável local
usando sincronização
        int l_contador;
        synchronized(this) {
            l_contador = ++ _Contador;
        }
```

```
        // Imprimindo número de requisições recebidas e rodapé
        l_pw.println("Requisições recebidas: " + Integer.toString
(l_contador));
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }

}
```

## 3.10 Retornando informações sobre o Servlet

Além dos métodos doXXX(), init() e destroy(), você também pode implementar o método getServletInfo() de um Servlet.

**Assinatura do método "getServletInfo ()" da classe HttpServlet**

```
public String getServletInfo ();
```

A implementação desse método deverá retornar um texto contendo informações gerais sobre o Servlet desenvolvido, como por exemplo, o autor, versão e informações de copyright de seu Servlet.

Implementando esse método para o nosso Servlet HelloWorld, temos:

**Servlet "HelloWorld" com implementação do método "getServletInfo()"**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Servlet para página HelloWorld
public class HelloWorld extends HttpServlet {

    // Atendimento de requisições HTTP com método GET
    public void doGet(HttpServletRequest p_request,
HttpServletResponse
            p_response) throws IOException {
```

```
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");
        l_pw.println("Hello World!");
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }

    // Retornando informações sobre esse Servlet
    public String getServletInfo () {
        return "Autor: Autores do livro; Versão: 1.0";
    }

}
```

Caso você não implemente esse método, um texto vazio será retornado pela implementação "default" desse método.

# Capítulo 4

# Servlets – Geração da saída

Embora já tenhamos visto um pouco sobre o funcionamento da geração de uma resposta simples de um Servlet, estaremos neste capítulo analisando esse processo mais a fundo e apresentando algumas funcionalidades mais avançadas.

## 4.1 Geração de saída HTML simples

Quando o Servlet recebe uma requisição, sem método doXXX() é chamado com dois parâmetros: uma referência a um objeto da classe javax.servlet.http.HttpServletRequest, que encapsula a requisição recebida, e uma referência a um objeto da classe javax.servlet.http.HttpServletResponse, que encapsula a resposta do Servlet.

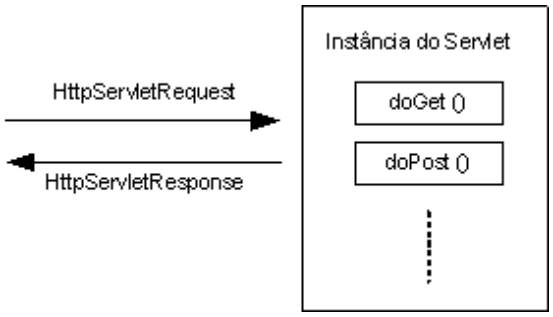


***Figura 4.1 – Atendimento de uma requisição por um Servlet.***

Sendo assim, a manipulação da resposta do Servlet passa, na verdade, pela manipulação do objeto dessa classe javax.servlet.http.HttpServletResponse.

Para gerar uma saída simples, por exemplo, você deve utilizar o método getWriter() desse objeto. A chamada desse método irá retornar uma referência a um objeto da classe java.io.PrintWriter, que encapsula um stream de saída para um conteúdo do tipo texto. Esse stream deve ser utilizado para enviar a resposta de seu Servlet para o cliente que enviou a requisição.

### Assinatura do método "getWriter ()"

```
public java.io.PrintWriter getWriter () throws java.io.IOException;
```

Embora a análise dos métodos dessa classe fuja um pouco ao escopo deste livro (trata-se de uma classe do próprio core Java), é interessante apresentar alguns de seus métodos aqui.

Os métodos print () e println () dessa classe, por exemplo, podem ser utilizados para adicionar Strings ao stream de saída do Servlet; como a saída é mantida em um buffer por questões de performance, você pode também utilizar o método flush () para forçar a liberação desse buffer de saída, fazendo que o conteúdo da resposta definido por você seja imediatamente enviado para o cliente.

### Exemplo de geração de saída simples de Servlet

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


// Servlet para página HelloWorld
public class HelloWorld extends HttpServlet {


    // Atendimento de requisições HTTP com método GET
    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.print( "<HTML><BODY>" );
```

```
        l_pw.print( “Hello World!” );
        l_pw.println( “</BODY></HTML>” );
        l_pw.flush ();
    }

}
```

Assim, nesse exemplo, utilizamos o objeto da classe java.io.PrintWriter para adicionar um conteúdo texto (no caso uma página HTML) ao stream de saída do Servlet. A resposta recebida pelo cliente será justamente essa página HTML.

Se você observou o exemplo anterior com cuidado (e, na verdade, todos os exemplos anteriores que incluiam algum método doXXX ()), você percebeu que o método doGet () foi declarado de maneira a possibilitar o lançamento de uma exceção java.io.IOException: essa exceção pode ser lançada pelo método getWriter () caso haja algum problema geração de saída do Servlet.

Outra opção de implementação para o Servlet HelloWorld seria, portanto:

Exemplo de geração de saída simples de Servlet com captura de exceção:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Segunda implementação do Servlet para página HelloWorld
public class HelloWorld extends HttpServlet {

    // Atendimento de requisições HTTP com método GET
    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response) {
        try {
            PrintWriter l_pw = p_response.getWriter ();
            l_pw.print(“<HTML><BODY>”);
            l_pw.print(“Hello World!”);
            l_pw.println(“</BODY></HTML>”);
            l_pw.flush ();
```

```
        }catch( IOException p_e ){
            ServletContext l_context = getServletContext ();
            l_context.log("Erro: não foi possível utilizar a referência ao
objeto PrintWriter");
        }
    }

}
```

Dessa forma, podemos capturar e tratar a exceção java.io.IOException caso ela ocorra, gerando, por exemplo, uma mensagem de log.

Mais detalhes sobre essas exceções lançadas durante o processo de geração da saída do Servlet, incluindo suas possíveis causas, são apresentadas ao longo das próximas seções desse capítulo.

## 4.2 Headers da resposta HTTP

Conforme apresentado na seção 3.2 deste livro, a resposta a uma requisição HTTP é composta de diversos elementos, sendo que alguns desses elementos são headers (cabeçalhos) HTTP.

**Exemplo de resposta HTTP do Servlet "HelloWorld"**

```
HTTP/1.1 200 OK
Server: Apache/1.3.26 (Unix)
Last-Modified: Sun, 22 Dec 2002 17:47:59 GMT
Content-Type: text/html
Content-Length: 40

<HTML><BODY>Hello World!</BODY></HTML>
```

Embora alguns headers sejam adicionados por default pelo "ServletContainer", como no caso da resposta do Servlet "HelloWorld" acima, podem haver situações em que você queira definir ou modificar seus próprios headers HTTP, e para fazer isso, você deve utilizar o método "setHeader ()".

**Assinatura do método "setHeader ()"**

```
public void setHeader( java.lang.String p_headerName,
java.lang.String p_headerValue );
```

Para indicar, por exemplo, que a resposta de um Servlet não deve ficar armazenada em nenhum cache (armazenamento temporário) do browser do usuário, nem em nenhum proxy, podemos definir alguns headers adicionais especiais por meio do seguinte trecho de código:

**Headers para evitar cacheamento da resposta de um Servlet**

```
public void doGet( HttpServletRequest p_request,
HttpServletResponse p_response ) {
    ...
    p_response.setHeader( "Cache-Control", "no-cache, must-
revalidate" );
    p_response.setHeader( "Pragma", "no-cache" );
    p_response.setHeader( "Expires", "Mon, 26 Jul 1997 05:00:00
GMT" );
    p_response.setDateHeader( "Last-Modified",
System.currentTimeMillis ());
    ...
}
```

Nesse código, o header "Expires" indica a data de expiração, o header "Last-Modified" indica a data de última modificação, e os headers "Cache-Control" e "Pragma" indicam o tratamento que o documento (ou seja, a resposta do Servlet) deve receber se houver cacheamento.

Com a inclusão do código anterior, a resposta do Servlet passa a ser:

**Exemplo de resposta HTTP do Servlet "HelloWorld" com headers adicionais**

```
HTTP/1.1 200 OK
Server: Apache/1.3.26 (Unix)
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
Expires: Mon, 26 Jul 1997 05:00:00 GMT
```

```
Last-Modified: Sun, 22 Dec 2002 17:47:59 GMT
Content-Type: text/html
Content-Length: 40

<HTML><BODY>Hello World!</BODY></HTML>
```

Uma observação muito importante é que essas adições / modificações de headers HTTP devem acontecer antes da geração do conteúdo da saída, para garantir a ordem dos elementos da resposta HTTP (ou seja, primeiro status, depois headers e, por fim, conteúdo). A alteração de um header após a escrita de parte ou de todo o conteúdo pode gerar uma exceção java.lang.IllegalStateException e interromper o processo de geração da saída do Servlet.

Assim, o código completo para o Servlet HelloWorld sem cacheamento pode ser escrito como:

### Servlet "HelloWorld" com headers para evitar cacheamento da página

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Terceira implementação do Servlet para página HelloWorld
public class HelloWorld extends HttpServlet {

    // Atendimento de requisições HTTP com método GET
    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response) {
        try {

            // Passo 1 – Definindo headers
            p_response.setHeader("Cache-Control", "no-cache, must-
revalidate");
            p_response.setHeader("Pragma", "no-cache");
            p_response.setHeader("Expires", "Mon, 26 Jul 1997
05:00:00 GMT");
            p_response.setDateHeader("Last-Modified",
System.currentTimeMillis ());

            // Passo 2 – Gerando a página de resposta
```

```
            PrintWriter l_pw = p_response.getWriter ();
            l_pw.print("<HTML><BODY>");
            l_pw.print("Hello World!");
            l_pw.println("</BODY></HTML>");
            l_pw.flush ();

        } catch( IOException p_e ) {
            ServletContext l_context = getServletContext ();
            l_context.log("Erro: não foi possível obter referência ao objeto
    PrintWriter");
        }
    }
}
```

Se você observou atentamente o código para a modificação dos headers, você deve ter reparado no uso de um método setDateHeader(): esse método é, na verdade, uma variante do método setHeader() que simplifica a definição de um header com uma data. Assim como esse método, existe um outro método setIntHeader() para a definição de headers que contenham valores inteiros.

### Assinaturas dos métodos "setDateHeader()" e "setIntHeader()"

```
public void setDateHeader(java.lang.String p_headerName, long
p_date);
public void setIntHeader(java.lang.String p_headerName, long p_int);
```

O segundo parâmetro do método setDateHeader() deve conter o número de milisegundos desde "epoch" (meia-noite, do dia 1º de Janeiro de 1970); o método currentTimeMillis() do objeto java.lang.System retorna esse valor para o instante corrente.

Além dos métodos anteriores, existe o método containsHeader(), para verificar se um header já foi definido, e os métodos addHeader(), addDateHeader() e addIntHeader() que permitem a adição de mais de um valor para um mesmo header.

### Assinaturas dos métodos “containsHeader()”, “addHeader()”,“addDateHeader()” e “addIntHeader ()”

```
public boolean containsHeader(java.lang.String p_headerName);
public void addHeader(java.lang.String p_headerName,
java.lang.String p_headerValue);
public void addDateHeader(java.lang.String p_headerName, long
p_date);
public void addIntHeader(java.lang.String p_headerName, long p_int);
```

Podemos usar o método “containsHeader()” para verificar, por exemplo, se o header “Content- Type” já foi definido, e defini- lo em caso negativo:

### Servlet “HelloWorld” com definição do header “Content- Type”

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Quarta implementação do Servlet para página HelloWorld
public class HelloWorld extends HttpServlet {

   // Atendimento de requisições HTTP com método GET
   public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response) {
      try {
         // Passo 1 - Definindo headers
         if(!p_responde.containsHeader(“Content- Type”))
            p_response.setHeader(“Content- Type”, “text/html”);

         // Passo 2 – Gerando a página de resposta
         PrintWriter l_pw = p_response.getWriter ();
         l_pw.print(“<HTML><BODY>”);
         l_pw.print(“Hello World!”);
         l_pw.println(“</BODY></HTML>”);
         l_pw.flush ();
      }catch( IOException p_e ) {
         ServletContext l_context = getServletContext ();
```

```
            l_context.log("Erro: não foi possível obter referência ao
objeto PrintWriter");
        }
    }

}
```

A função desse header Content-Type é informar o tipo do conteúdo que está contido na resposta, para que o browser (ou dispositivo cliente que está fazendo o acesso) saiba como esse conteúdo deve ser interpretado e apresentado.

Nos exemplos anteriores esse header não foi definido explicitamente em código: nesses casos, o ServletContainer automaticamente define seu valor como text/html. Esse valor indica que o conteúdo deve ser interpretado pelo browser como uma página HTML.

Outro header que é definido automaticamente pelo ServletContainer, caso o desenvolvedor não o defina explicitamente, é o header Content-Length. Esse header indica o tamanho em bytes do conteúdo contido na resposta.

Podem haver casos em que seja necessário se alterar esse comportamento "default": nesses casos, o desenvolvedor pode utilizar os métodos setXXXHeader () apresentados anteriormente, ou o método setContentLength () diretamente.

**Assinatura do método "setContentLength()"**

```
public void setContentLength( int p_contentLength );
```

# 4.2 Geração de outros tipos de saídas

Conforme mencionado no na seção anterior, o header Content-Type serve para indicar o tipo do conteúdo contido na resposta do Servlet. Dessa maneira, o valor text/html indica uma página HTML, o valor image/jpeg indica uma imagem JPEG, e assim por diante.

Devido a sua importância, existe uma função especial utilizada para a definição do valor desse header.

### Assinatura do método "setContentType()"

```
public void setContentType(java.lang.String p_contentType);
```

Embora a geração de uma saída HTML seja a situação mais comum, podem haver situações em que você deseje gerar outros tipos de saída. Nesses casos, você deve utilizar o método setContentType para especificar o tipo do conteúdo desejado.

Assim, podemos usar o seguinte trecho de código para retornar uma imagem JPEG contida em um arquivo:

### Servlet "ImageServlet"

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Servlet para geração de saída com imagem JPEG
public class ImageServlet extends HttpServlet {

    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response) {
        FileInputStream l_imageFile = null;
        try {
            // Definindo o tipo do conteúdo
            p_response.setContentType("image/jpeg");

            // Obtendo o arquivo com a imagem JPEG a partir dos
parâmetros
            // de inicialização do Servlet
            l_imageFile = new FileInputStream(getServletConfig
                ().getInitParameter("JpegFilename"));

            // Lendo o conteúdo de um arquivo contendo uma imagem
e
            // retornando este conteúdo como resposta
            ServletOutputStream l_os = p_response.getOutputStream ();
            byte [] l_buffer = new byte[1024];
            int l_bytes = 0;
```

```
            while((l_bytes = l_imageFile.read(l_buffer)) != -1)
                  l_os.write(l_buffer, 0, l_bytes);

            // Finalizando processo de geração de saída e fechando o
arquivo
            l_os.flush ();
            l_imageFile.close ();

        }catch(IOException p_e) {
            try {
                if(l_imageFile != null) l_imageFile.close ();
            }catch(Exception p_e2) {}
            ServletContext l_context = getServletContext ();
            l_context.log("Erro: não foi possível ler imagem de
arquivo");
        }
    }

}
```

No exemplo anterior, você poderá observar também a utilização do método getOutputStream (). Esse método deve ser utilizado, em substituição ao método getWriter () que foi usado nos exemplos anteriores, para obter uma referência a um stream de saída binária do Servlet.

Assim, quando você desejar que seu Servlet retorne um conteúdo binário (não-texto), como uma imagem JPEG no código anterior, você deverá utilizar esse método getOutputStream (), e obter uma referência a um objeto da classe javax.servlet.ServletOutputStream.

Esse stream "ServletOutputStream" extende, na verdade, a classe java.io.OutputStream e, sendo assim, herda os diversos métodos write() e o método flush() da classe-mãe.

### Assinaturas dos métodos "write()" e "flush ()"

```
public void write(byte [] p_byteArray);
public void write( byte [] p_byteArray, int p_offset, int p_length );
public void write( int p_byte );
public void flush();
```

Obviamente, a utilização dos métodos anteriores e até mesmo a obtenção desse stream de saída binária do Servlet podem gerar exceções do tipo java.io.IOException, em situações em que o usuário que está operando o browser aperta o botão Stop, antes de todo o conteúdo gerado pelo seu Servlet seja enviado, por exemplo. Por isso é importante que você se preocupe em capturar essas exceções, e fazer os tratamentos necessários caso isso ocorra.

## 4.3 Gerando conteúdo XML

Dentre os diversos tipos de conteúdos que podem ser gerados como resposta de um Servlet, apresentamos aqui, como mais um exemplo, a geração de documentos XML.

Documentos XML são normalmente usados para estruturar um conjunto de dados de maneira simples, bem-definida e eficiente. Um exemplo de documento XML já foi apresentado no *Capítulo 2 – Instalação e Configuração*: o "Deployment Descriptor" é um arquivo XML que descreve a maneira como o ServletContainer deve carregar e gerenciar uma "Aplicação Web".

Um outro tipo de documento XML bastante utilizado atualmente é o WML (definido dentro do padrão WAP, ou Wireless Application Protocol): através desse formato, um servidor pode formatar um conteúdo que será visualizado na tela de telefones celulares.

Embora a explicação detalhada do formato de um documento WML fuja ao escopo deste livro (mais detalhes podem ser obtidos no site do WapForum, http://www.wapforum.org), apresentamos abaixo o código de um Servlet que gera uma página WML apenas para exemplificar a geração de um documento XML.

**Servlet "HelloWorldWML"**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
// Quarta implementação do Servlet para página HelloWorld, desta
vez com a geração da resposta em formato WML
public class HelloWorldWML extends HttpServlet {


    // Atendimento de requisições HTTP com método GET
    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response) {
        try {
            // Passo 1 – Definindo headers
            // Em um documento XML normal, o "Content- Type" é
normalmente definido
            // como "text/xml"; no caso específico do WML este header
deve ser
            // definido como "text/vnd.wap.wml".
            p_response.setContentType("text/vnd.wap.wml");


            // Passo 2 – Gerando a página de resposta
            PrintWriter l_pw = p_response.getWriter ();
            l_pw.println("<?xml version=\"1.0\"?>");
            l_pw.println("<!DOCTYPE xml PUBLIC \"- //
WAPFORUM//DTD WML 1.1//EN\
                "\"http://www.wapforum.org/DTD/wml_1.1.xml\">");
            l_pw.print("<wml><card>");
            l_pw.print("<p align=\"center\">Hello World</p>");
            l_pw.print("</card></wml>");
            l_pw.flush ();
        }catch(IOException p_e) {
            ServletContext l_context = getServletContext ();
            l_context.log("Erro: " + p_e.getMessage ());
        }
    }
}
```

É interessante observar no exemplo anterior a utilização do método setContentType () para indicar o tipo do conteúdo na resposta; embora para documentos XML esse tipo seja geralmente o text/xml, no caso específico do WML ele deve ser definido como text/vnd.wap.wml.

## 4.4 Status HTTP

Como explicamos na seção 3.2 deste livro, além dos headers e do conteúdo propriamente dito, a resposta a uma requisição HTTP deve conter também a informação do status da resposta, sendo que essa informação é composta por um código numérico mais um string com uma mensagem.

Esse status é utilizado não só para indicar se o processamento da requisição recebida foi bem-sucedida ou não, mas também para indicar algumas outras situações possíveis, como por exemplo que o documento solicitado encontra-se disponível em uma outra URL.

Um Servlet pode definir esse status através do método setStatus () da classe HttpServletResponse, sendo que, conforme explicado anteriormente, para preservar a ordem dos elementos da resposta HTTP, a chamada a esse método deve ser feita antes de qualquer definição de headers ou início da geração do conteúdo da resposta.

### Assinatura do método "setStatus()"

```
public void setStatus(int p_statusCode);
```

Nos exemplos apresentados anteriormente, como esse status não foi atribuído explicitamente em código, automaticamente o ServletContainer o definiu como sendo 200, ou status OK. Vemos a seguir, por exemplo, a resposta do Servlet HelloWorld:

### Exemplo de resposta HTTP do Servlet "HelloWorld"

```
HTTP/1.1 200 OK
Server: Apache/1.3.26 (Unix)
Last-Modified: Sun, 22 Dec 2002 17:47:59 GMT
Content-Type: text/html
Content-Length: 40

<HTML><BODY>Hello World!</BODY></HTML>
```

Temos, a seguir, uma tabela com alguns dos códigos de status HTTP existentes e seus respectivos significados:

| Código de Status | Mensagem | Significado |
|---|---|---|

| | | |
|---|---|---|
| 200 | OK | Requisição foi processada com sucesso. |
| 302 | Moved Temporarily | O documento solicitado encontra-se disponível em outra URL. |
| 404 | Page Not Found | O documento solicitado não foi encontrado. |
| 500 | Internal Server Error | Erro no processamento / obtenção do documento requisitado. |
| 503 | Service Unavailable | O serviço não se encontra disponível. |

Assim, no caso de nosso Servlet de geração de imagem JPEG "ImageServlet", podemos modificá-lo de forma que ele retorne um código 404 caso o arquivo com a imagem a ser gerada não exista.

**Servlet "ImageServlet" com possível uso do código de status HTTP 404**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Servlet para geração de saída com imagem JPEG; gera código de status 404
// caso o arquivo com a imagem não exista.
public class ImageServlet extends HttpServlet {

    public void doGet(HttpServletRequest p_request, HttpServletResponse p_response) {
        FileInputStream l_imageFile = null;
        try {
            // Verificando se o arquivo com a imagem existe
            String l_filename = getServletConfig ().getInitParameter ("JpegFilename");
            if(l_filename != null) {
                File l_file = new File(l_filename);
                if(!l_file.exists ()) l_filename = null;
            }
            if(l_filename == null) {
                // Como o arquivo não existe, retornamos o código 404
                p_response.setStatus(p_response.SC_NOT_FOUND);
```

```
                return;
            }

            // Definindo o tipo do conteúdo
            p_response.setContentType("image/jpeg");

            // Obtendo o arquivo com a imagem JPEG a partir dos
parâmetros
            // de inicialização do Servlet
            l_imageFile = new FileInputStream(l_filename);

            // Lendo o conteúdo de um arquivo contendo uma imagem
e
            // retornando este conteúdo como resposta
            ServletOutputStream l_os = p_response.getOutputStream ();
            byte [] l_buffer = new byte[1024];
            int l_bytes = 0;

            while((l_bytes = l_imageFile.read(l_buffer)) != -1)
                    l_os.write(l_buffer, 0, l_bytes);

            // Finalizando processo de geração de saída e fechando o
arquivo
            l_os.flush ();
            l_imageFile.close ();

        } catch( IOException p_e ) {
            try {
                if(l_imageFile != null) l_imageFile.close ();
            } catch(Exception p_e2) {}
            ServletContext l_context = getServletContext ();
            l_context.log("Erro: não foi possível ler imagem de
arquivo");
        }
    }
}
```

Podemos ver, nessa implementação, que o código de status é definido pela constante numérica SC_NOT_FOUND da classe HttpServletResponse.

**Utilização de constante SC_NOT_FOUND da classe "javax.servlet.HttpServletResponse"**

```
...
p_response.setStatus(p_response.SC_NOT_FOUND);
...
```

Para cada código de status HTTP existente, a classe HttpServletResponse define uma constante correspondente, sendo que você deve dar preferência ao uso dessas constantes (em vez do número em si) para aumentar a legibilidade de seu código.

| Código de Status | Mensagem | Constante |
|---|---|---|
| 200 | OK | SC_OK |
| 302 | Moved Temporarily | SC_MOVED_TEMPORARILY |
| 404 | Page Not Found | SC_NOT_FOUND |
| 500 | Internal Server Error | SC_INTERNAL_SERVER_ERROR |
| 503 | Service Unavailable | SC_SERVICE_UNAVAILABLE |

Vale observar também que, para alguns códigos de status, existem headers auxiliares que contém indicações sobre o que o cliente deve fazer ao receber aquele código de status.

Assim, no caso do código 503 (SC_SERVICE_UNAVAILABLE), você pode definir o valor do header Retry-After para indicar o tempo estimado em segundos até que o serviço volte a ficar ativo.

Outro exemplo é o código de status 302 (SC_MOVED_TEMPORARILY): ao definir esse código de status, você deve definir também o valor do header Location para indicar a URL onde o novo documento pode ser encontrado.

Temos a seguir o exemplo de um Servlet que, dependendo do endereço IP do cliente que envia a requisição, redireciona o acesso para uma nova página contendo uma mensagem de acesso negado.

## Servlet “CheckIPAccess”

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Servlet para verificação do IP do cliente que faz a requisição;
// se o prefixo do IP não for reconhecido, direciona para uma página
// com uma mensagem de acesso negado.
public class CheckIPAccess extends HttpServlet {

    // Prefixo dos IPs permitidos (rede local)
    private static final String _AllowedIPPrefix = “192.168.0.”;

    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response) {
        try {
            // Verificando o prefixo do cliente que fez a requisição
            String l_ip = p_request.getRemoteAddr ();
            if((l_ip != null) && (l_ip.startsWith(_AllowedIPPrefix))) l_ip =
null;
            if(l_ip == null) {
                p_response.setStatus
(p_response.SC_MOVED_TEMPORARILY);
                p_response.setHeader(“Location”, “/accessdenied.html”);
                return;
            }

            // Acesso permitido; imprimindo página de sucesso
            PrintWriter l_pw = p_response.getWriter ();
            l_pw.println(“<HTML><BODY>”);
            l_pw.println(“Parabéns, seu acesso foi permitido!”);
            l_pw.println(“</BODY></HTML>”);
            l_pw.flush ();

        } catch(IOException p_e) {
            ServletContext l_context = getServletContext ();
            l_context.log(“Erro: “ + p_e.getMessage ());
        }
    }
```

```
}
```

Essa situação de redirecionamento do acesso mostra-se tão frequente que um outro método, chamado sendRedirect (), é disponibilizado pela API de Servlets para a obtenção desse mesmo efeito.

### Assinaturas do método "sendRedirect()"

```
public void sendRedirect(String p_location);
```

Desta forma, poderíamos trocar, no Servlet anterior, o trecho de código:

```
...
if(l_ip == null) {
    p_response.setStatus(p_response.SC_MOVED_TEMPORARILY);
    p_response.setHeader("Location", "/accessdenied.html");
    return;
}
...
```

Pelo código:

```
...
if(l_ip == null) {
    p_response.sendRedirect("/accessdenied.html");
    return;
}
...
```

Veremos a seguir uma variação do ServletContador, apresentado anteriormente neste livro, que utiliza o método sendRedirect() para redirecionar o milésimo acesso para uma página especial:

### Variação do ServletContador: redireciona milésimo acesso

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


// Servlet simples que retorna página HTML com o número de
requisições
// recebidas até o momento; redireciona milésimo acesso
```

```
public class ServletContador extends HttpServlet {

    // Contador das requisições recebidas
    private int _Contador = 0;

    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {

        // Armazenando valor do contador em uma variável local
usando sincronização
        int l_contador = 0;
        synchronized(this) {
            l_contador = ++ _Contador;
        }

        // Se este for o milésimo acesso, redireciona para uma página
especial
        if(l_contador == 1000) {
            p_response.sendRedirect("/premioespecial.html");
            return;
        }

        // ... caso contrário, imprime número de requisições recebidas
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");
        l_pw.println("Requisições recebidas: " + Integer.toString
(l_contador));
        l_pw.flush ();
    }
}
```

## 4.5 Código de Status de erro

Além dos métodos setStatus () e sendRedirect () que permitem a manipulação do status da resposta do Servlet, é possível também utilizar o método sendError () para indicar erros no processamento da requisição recebida.

**Assinaturas do método "sendError()"**

```
public void sendError(int p_statusCode);
public void sendError(int p_statusCode, java.lang.String
p_statusMessage);
```

Esse método deve ser utilizado para a definição de códigos de status nas faixas dos 400 e 500, como por exemplo, os códigos SC_NOT_FOUND (404) e SC_SERVICE_UNAVAILABLE (503), sendo que, se o segundo parâmetro for utilizado, este deve conter a mensagem descritiva que acompanha o código de status.

Vale lembrar que de acordo com a configuração das páginas de erro da Aplicação Web (descrita no "Deployment Descriptor", vide seção 2.2), a definição de um código de status de erro pode levar à apresentação de uma página alternativa de erro.

## 4.6 "Buffering" da resposta

Um último recurso interessante relacionado ao processo de geração de resposta do Servlet é o buffering do conteúdo dessa resposta.

Esse buffering é controlado através dos seguintes métodos da classe HttpServletResponse:

**Assinaturas dos métodos relacionados ao buffering de resposta do Servlet**

```
public int getBufferSize();
public void setBufferSize( int p_size );
public boolean isCommitted();
public void reset();
public void flushBuffer();
```

Ao invés de simplesmente repassar para o cliente conteúdos parciais a cada chamada aos métodos de escrita da classe HttpServletResponse (como por exemplo, os métodos println() e write()), para otimizar o desempenho da aplicação o ServletContainer armazena esse conteúdo em um buffer, sendo que esse conteúdo é repassado para o cliente somente quando o buffer enche ou quando é encerrada a execução do Servlet.

Dessa forma, embora tenhamos dito anteriormente que há uma ordem de definição dos elementos HTTP de resposta de um Servlet a ser obedecida, utilizando essas funções de manipulação do buffer de resposta, é possível contornar essa regra.

No exemplo a seguir, utilizamos essa funcionalidade para, dependendo do processamento realizado, redefinir os elementos da resposta do Servlet.

### Servlet com manipulação dos buffers da resposta

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Servlet que realiza um processamento qualquer, e, conforme o
caso,
// manipula o buffer de resposta
public class ServletBufferResposta extends HttpServlet {

    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
          throws IOException {

       // Imprimindo o tamanho do buffer de resposta no log da
aplicação
       getServletContext ().log("O tamanho default do buffer de
resposta é \"" +
          Integer.toString(p_response.getBufferSize ()) + "\"");

       // Definindo novo tamanho de buffer de resposta de 1k
       p_response.setBufferSize(1024);

       // Imprimindo cabeçalho HTML
       PrintWriter l_pw = p_response.getWriter ();
       l_pw.println("<HTML><BODY>");

       // Executando um processamento qualquer
       boolean l_sucessoProcessamento = ExecutaProcessamento();
```

```
        // Se ocorreu um erro no processamento anterior,
        // modificamos o status da resposta
        if(!l_sucessoProcessamento) {

            // Apesar de termos definido explicitamente o tamanho do
buffer, é sempre
            // bom verificar se o seu conteúdo já não foi repassado
para o cliente
            // por meio do método isCommited ()
            if(!p_response.isCommitted ()) {

                // Limpamos o conteúdo do buffer através do método
reset e redefinimos
                // o status da resposta para INTERNAL_SERVER_ERROR
(usado para
                // indicar que ocorreu um erro no processamento da
requisição)
                p_response.reset ();
                p_response.sendError
( p_response.SC_INTERNAL_SERVER_ERROR );
                return;
            }
        }

        // Imprimindo final da resposta HTML
        l_pw.println("Sucesso no
processamento!</BODY></HTML>");
        l_pw.flush ();
    }
}
```

# Capítulo 5

# Servlets – Captura de parâmetros da requisição

Além de gerar uma resposta para cada requisição recebida, outro trabalho importante que deve ser realizado por um Servlet é o de capturar e tratar os parâmetros da requisição (gerados, por exemplo, a partir de um formulário HTML). Conforme o resultado desse tratamento, um Servlet pode gerar respostas diferentes.

Esse capítulo explora exatamente esse processo de captura dos parâmetros da requisição recebida.

## 5.1 Informações sobre o servidor

No *Capítulo 3 – Servlets – características básicas* desse livro (seções 3.5 e 3.6), apresentamos algumas das funções para a obtenção de parâmetros e atributos do Servlet e de seu contexto.

Além dos métodos apresentados anteriormente, é possível obter algumas informações adicionais relevantes, como o nome do Servlet corrente, e os dados referentes a versão da API de Servlets suportada pelo ServletContainer (disponível somente para ServletContainer's que implementem a especificação de Servlets 2.1 ou superior).

**Assinaturas dos métodos "getServletName ()", "getMajorVersion()" e "getMinorVersion()"**

```
public String ServletConfig.getServletName ();
public int ServletContext.getMajorVersion ();
public int ServletContext.getMinorVersion ();
```

Existem também métodos para a obtenção de informações sobre o servidor em si, tais como: nome do servidor, que pode ser o próprio nome da máquina na rede (dependendo de como o servidor foi instalado); a porta onde o servidor recebe as requisições; e um texto de identificação do servidor, que normalmente contém o nome e versão do software que implementa o ServletContainer.

### Assinaturas dos métodos "getServerInfo ()", "getServerName()" e "getServerPort ()"

```
public String ServletContext.getServerInfo ();
public String HttpServletRequest.getServerName ();
public int HttpServletRequest.getServerPort ();
```

Em particular, os métodos getServerName () e getServerPort () devem ser chamados a partir do objeto da classe HttpServletRequest: como cada servidor pode definir diversos servidores virtuais, através de uma técnica chamada de "virtual hosts", é necessário se utilizar o próprio objeto da requisição para distinguir qual o servidor (virtual) que gerou a chamada ao servlet. Mais informações sobre esse recurso de "virtual hosts" podem ser encontradas, geralmente, na documentação de servidores Web.

No próximo exemplo, utilizamos os métodos apresentados anteriormente para construir uma página HTML com todas as informações conhecidas sobre o servidor, o Servlet e seu contexto.

### Servlet para apresentação de parâmetros do servidor

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


// Apresenta informações do servidor
public class ServletInfoServidor extends HttpServlet {
```

```
    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {

        // Imprimindo cabeçalho HTML
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");

        // Imprimindo informações sobre o servidor
        l_pw.println("Texto de Identificação do Servidor: " +
getServletContext
            ().getServerInfo () + "<BR>");
        l_pw.println("Nome do Servidor: " + p_request.getServerName
() + "<BR>");
        l_pw.println("Porta do Servidor: " + p_request.getServerPort ()
+ "<BR>");

        // Imprimindo informações sobre o Servlet e seu contexto
        l_pw.println("Nome do Servlet: " + getServletConfig ().
getServletName
            () + "<BR>");
        l_pw.println("Versão da API suportada pelo ServletContainer :"

            + Integer.toString( getServletContext ().getMajorVersion ())
+
            "." + Integer.toString(getServletContext ().getMinorVersion
()) + "<BR>");

        // Imprimindo rodapé HTML
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }

}
```

# 5.2 Informações sobre a requisição:

Conforme mostramos na seção 3.2 desse livro, uma requisição HTTP é recebida por um Servlet é composta de alguns elementos básicos: método, path, informações sobre a versão do protocolo e headers.

A API de Servlets permite que o desenvolvedor acesse todas essas informações associadas a cada requisição recebida. Assim, temos de início as seguintes funções:

**Assinaturas dos métodos associados a captura de informações de requisição**

```
public String HttpServletRequest.getRemoteAddr ();
public String HttpServletRequest.getRemoteHost ();
public String HttpServletRequest.getMethod ();
public String HttpServletRequest.getRequestURI ();
public String HttpServletRequest.getScheme ();
public String HttpServletRequest.getProtocol ();
```

Essas funções permitem recuperar o endereço IP do cliente que gerou a requisição, o nome da máquina associada a esse endereço IP (se for possível obter esse nome), e o método, path, e protocolo e sua versão associados a essa requisição. O protocolo da requisição pode também ser chamado de esquema (= “scheme”).

Temos, a seguir, um Servlet que imprime essas informações para cada requisição recebida.

**Servlet que imprime informações sobre cada requisição recebida**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


// Imprime informações sobre cada requisição recebida
public class ServletInfoReq extends HttpServlet {


    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
          throws IOException {


       // Imprimindo cabeçalho HTML
```

```
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");


        // Imprimindo informações sobre a requisição
        l_pw.println("Endereço de sua máquina: " +
p_request.getRemoteAddr ()
            + "<BR>");
        l_pw.println("Nome de sua máquina: " +
p_request.getRemoteHost () + "<BR>");
        l_pw.println("Método da requisição: " + p_request.getMethod ()
+ "<BR>");
        l_pw.println("Path da requisição: " + p_request.getRequestURI
() + "<BR>");
        l_pw.println("Protocolo da requisição: " + p_request.getScheme
() + "<BR>");
        l_pw.println("Versão do protocolo da requisição: "
            + p_request.getProtocol  () + "<BR>");


        // Imprimindo rodapé HTML
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }


    // Nesse Servlet, os métodos GET e POST são tratados da mesma
maneira
    // (pelo mesmo trecho de código)
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
        throws IOException {
        doGet(p_request, p_response);
    }


}
```

Assim, uma saída possível para esse Servlet poderia ser:

### Exemplo de página retornada pelo Servlet ServletInfoReq

```
<HTML><BODY>
Endereço de sua máquina: 127.0.0.1<BR>
Nome de sua máquina: localhost<BR>
Método da requisição: GET<BR>
```

```
Path da requisição: /livroservlets/ServetInfoReq/a.html<BR>
Protocolo da requisição: http<BR>
Versão do protocolo da requisição: HTTP/1.1<BR>
</BODY></HTML>
```

Além do método getRequestURI () utilizado em nosso exemplo, a API de Servlets disponibiliza algumas funções adicionais para a obtenção de informação relacionadas ao path da requisição.

### Assinaturas de métodos adicionais para a obtenção de informações de path

```
public String HttpServletRequest.getContextPath ();
public String HttpServletRequest.getServletPath ();
public String HttpServletRequest.getPathInfo ();
public String HttpServletRequest.getPathTranslated ();
public static StringBuffer HttpUtils.getRequestURL
(HttpServletRequest p_request);
```

Os métodos getContextPath (), getServletPath () e getPathInfo () retornam, respectivamente, o caminho do contexto, o caminho do Servlet abaixo do contexto, e o restante do "path" da requisição (se excluírmos as duas primeiras informações).

Em particular, no ambiente apresentado no *Capítulo 2 – Instalação e Configuração* desse livro (utilizando o Apache Tomcat), o caminho do contexto equivale ao subdiretório criado abaixo da pasta webapps, ou seja, o subdiretório da aplicação Web. Por outro lado, o caminho do Servlet, corresponde ao mapeamento ("servlet- mapping") que originou a chamada ao Servlet.

Se o Servlet anterior, ServletInfoReq, estivesse instalado junto a uma aplicação Web livroservlets, por exemplo, e, no "Deployment Descriptor" dessa aplicação Web, houvesse um mapeamento da URL /ServletInfoReq/ para esse Servlet, teríamos os seguintes resultados para cada requisição recebida:

Entra tabela

A função seguinte, getPathTranslated (), tenta traduzir a informação de path da requisição para o caminho real do

recurso no disco. É importante observar que essa função só funciona caso a aplicação web não tenha sido implantada como um arquivo WAR (explicado mais adiante no *Capítulo 9 – Tópicos Adicionais*).

A última função, getRequestURL(), serve para construir a URL que gerou a requisição ao Servlet, incluindo as informações de esquema, protocolo, método etc. Essa função deve ser chamada diretamente a partir da classe HttpUtils do pacote javax.servlet.http, pois trata-se de um método estático dessa classe.

Agregando, então, essas novas funções ao Servlet ServletInfoReq, temos o código a seguir:

### Servlet que imprime informações sobre cada requisição recebida

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Imprime informações sobre cada requisição recebida
public class ServletInfoReq extends HttpServlet {

    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {

        // Imprimindo cabeçalho HTML
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");

        // Imprimindo informações sobre a requisição
        l_pw.println("Endereço de sua máquina: " +
p_request.getRemoteAddr ()
            + "<BR>");
        l_pw.println("Nome de sua máquina: " +
p_request.getRemoteHost () + "<BR>");
        l_pw.println("Método da requisição: " + p_request.getMethod ()
+ "<BR>");
        l_pw.println("Path da requisição: " + p_request.getRequestURI
() + "<BR>");
```

```
        l_pw.println("Protocolo da requisição: " + p_request.getScheme
() + "<BR>");
        l_pw.println("Versão do protocolo da requisição: " +
p_request.getProtocol ()
            + "<BR>");


        l_pw.println("<BR>");
        l_pw.println("Caminho do contexto: " +
p_request.getContextPath () + "<BR>");
        l_pw.println("Caminho do Servlet: " + p_request.getServletPath
() + "<BR>");
        l_pw.println("Informações adicionais de caminho: " +
p_request.getPathInfo ()
            + "<BR>");
        l_pw.println("Caminho traduzido: " +
p_request.getPathTranslated ()
            + "<BR>");
        l_pw.println("URL completa da requisição: " +
HttpUtils.getRequestURL(p_request)
            + "<BR>");


        // Imprimindo rodapé HTML
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }


    // Nesse Servlet, os métodos GET e POST são tratados da mesma
maneira
    // (pelo mesmo trecho de código)
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        doGet(p_request, p_response);
    }


}
```

Com essas novas funções, uma possível saída para esse Servlet passaria a ser:

### Exemplo de página retornada pelo Servlet ServletInfoReq

```
<HTML><BODY>
```

```
Endereço de sua máquina: 127.0.0.1<BR>
Nome de sua máquina: localhost<BR>
Método da requisição: GET<BR>
Path da requisição: /livroservlets/ServetInfoReq/a.html<BR>
Esquema da requisição: http<BR>
Versão do protocolo da requisição: HTTP/1.1<BR>
<BR>
Caminho do contexto: /livroservlets<BR>
Caminho do Servlet: /ServletInfoReq<BR>
Informações adicionais de caminho: /a.html<BR>
Caminho traduzido: C:\Program Files\Apache Tomcat
4.0\webapps\livroservlets\a.html<BR>
URL completa da requisição:
http://localhost:8080/livroservlets/ServletInfoReq/a.html<BR>
</BODY></HTML>
```

## 5.3 Formulários HTML e parâmetros da requisição:

Se você já construiu páginas HTML, ou até mesmo navegou na Internet, você já deve ter se deparado com formulários HTML.

Formulários HTML possibilitam, por exemplo, que digitemos textos ou selecionemos valores em campos de uma página HTML, sendo que, ao final, normalmente clicamos em um botão para que essas informações sejam enviadas para o servidor.

Ao fazermos isso, há, na verdade, uma requisição que é gerada pelo browser, sendo que as informações digitadas e selecionadas por nós são “anexadas” como parâmetros dessa requisição.

Temos, a seguir, um exemplo de página HTML com um formulário de cadastro.

**Exemplo de página HTML com formulário de cadastro**

```
<HTML><BODY>
```

```
<FORM METHOD=”POST”
ACTION=”/livroservlets/ServletParamsReq”>
Digite seu nome:<BR><INPUT TYPE=”text” NAME=”nome”
SIZE=”20”><BR><BR>
Selecione sua(s) cor(es) preferida(s):<BR>
<SELECT NAME=”cores” MULTIPLE>
<OPTION VALUE=”azul”>azul</OPTION><OPTION
VALUE=”vermelho”>vermelho</OPTION>
<OPTION VALUE=”verde”>verde</OPTION><OPTION
VALUE=”amarelo”>amarelo</OPTION>
<OPTION VALUE=”branco”>branco</OPTION><OPTION
VALUE=”preto”>preto</OPTION>
</SELECT> <BR><BR>
<INPUT TYPE=”submit” NAME=”env” VALUE=”Enviar”>
</FORM>
</BODY></HTML>
```

Nesse formulário, existem 3 parâmetros que serão enviados junto com a requisição ao servidor quando clicamos no botão “Enviar”: nome, cores e env. Em particular, o parâmetro cores pode estar associado a mais de um valor, dependendo de quantas cores forem selecionados pelo usuário que visitar essa página.

Uma outra forma de enviar parâmetros é a sua concatenação diretamente no path da requisição, bastando para isso acrescentar o caractere ? ao final do path, seguido de pares <nome do parâmetro>=<valor do parâmetro> , separados pelo caractere &. Poderiamos, por exemplo, simular um envio de informações do formulário anterior através do seguinte path de requisição:

“/livroservlets/ServletParamsReq/?nome=meunome&cores
=azul&cores=preto&env=Enviar”

Nesse caso, todo o texto que segue o caractere ? é chamado de texto da “query”. É importante observar que os nomes e valores dos parâmetros contidos nesse texto devem estar codificados no formato MIME “x-www-form-urlencoded”.

Infelizmente, um estudo mais aprofundado da linguagem HTML e do protocolo HTTP foge ao escopo desse livro,

sendo deixado para o leitor a opção de consultar a literatura específica para maiores esclarescimentos.

## 5.4 Captura de parâmetros da requisição:

A API de Servlets disponibiliza diversos métodos para capturar os parâmetros de uma requisição:

**Métodos para a captura de parâmetros de formulários HTML**

```
public String HttpServletRequest.getParameter(String
p_parameterName);
public String [] HttpServletRequest.getParameterValues(String
p_parameterName);
public java.util.Enumeration HttpServletRequest.getParameterNames
();
public String HttpServletRequest.getQueryString ();
```

O primeiro método para a obtenção de parâmetros da requisição é o método getParameter(): ele recebe um nome de parâmetro e retorna o valor associado a esse nome de parâmetro.

Por outro lado, conforme vimos no formulário de exemplo da seção anterior, podem existir campos com diversos valores associados a um mesmo nome (vide campo cores do exemplo). Nesses casos, devemos utilizar o método getParameterValues (), que irá retornar não um único valor, mas sim um array de valores.

O método seguinte, getParameterNames (), retorna uma lista enumerada com todos os nomes de parâmetros recebidos. Dessa forma, é possível iterar nessa lista e obter os respectivos valores de parâmetros.

Por fim, o último método, getQueryString (), retorna justamente o texto da “query” explicado na seção anterior, ou seja, a parte do path de requisição que vem após o caractere ? (se existir).

O Servlet ServletParamsReq usa os métodos apresentados anteriormente para capturar os parâmetros enviados pelo usuário e apresentá- los em sua saída

**Servlet que imprime parâmetros de cada requisição recebida**

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Imprime parâmetros de cada requisição recebida
public class ServletParamsReq extends HttpServlet {

    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {

        // Imprimindo cabeçalho HTML
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");

        // Imprimindo o texto da "Query"
        if( p_request.getQueryString () != null)
            l_pw.println("Texto da Query: " + p_request.getQueryString
() + "<BR>");

        // Iterando nos nomes dos parâmetros
        Enumeration l_paramNames = p_request.getParameterNames
();
        while(l_paramNames.hasMoreElements ()) {

            // Imprimindo par nome / valor
            String l_paramName = (String) l_paramNames.nextElement
();
            String l_paramValue = p_request.getParameter
(l_paramName);
            l_pw.println("Parâmetro: " + l_paramName + "<BR>");
            l_pw.println(" - Valor: " + l_paramValue + "<BR>");
```

```
            // Imprimindo todos os valores para o parâmetro corrente
            String [] l_paramValues = p_request.getParameterValues
(l_paramName);
            l_pw.print(" – Todos os valores: ");
            if(l_paramValues != null)
                for(int I=0; I<l_paramValues.length; I++) {
                    if( I == 0 ) l_pw.print(l_paramValues[ I ]);
                    else l_pw.print(", " + l_paramValues[ I ]);
                }
            l_pw.println("<BR>");
        }

        // Imprimindo rodapé HTML
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }

    // Nesse Servlet, os métodos GET e POST são tratados da mesma
maneira
    // (pelo mesmo trecho de código)
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        doGet(p_request, p_response);
    }

}
```

Se utilizarmos o formulário apresentado na seção anterior para enviar informações para esse Servlet, uma saída possível seria:

### Exemplo de página retornada pelo Servlet ServletInfoReq

```
<HTML><BODY>
Parâmetro: nome<BR>
- Valor: meunome<BR>
- Todos os valores: meunome<BR>
Parâmetro: cores<BR>
- Valor: azul<BR>
- Todos os valores: azul, verde<BR>
```

```
Parâmetro: env<BR>
- Valor: Enviar<BR>
- Todos os valores: Enviar<BR>
</BODY></HTML>
```

Frequentemente, quando você quiser adicionar interatividade a sua aplicação Web, permitindo que o usuário envie dados ao servidor, você irá utilizar formulários HTML, e por isso, é muito importante que você domine os métodos apresentados nessa seção.

## 5.5 Headers da requisição HTTP

Assim como acontece na resposta de um Servlet, a requisição recebida também pode conter headers. Esses headers podem servir para indicar, por exemplo, características sobre o cliente que gerou a requisição.

### Métodos para a captura de informações de headers da requisição

```
public String HttpServletRequest.getHeader(String p_headerName);
public long HttpServletRequest.getDateHeader(String p_headerName);
public int HttpServletRequest.getIntHeader(String p_headerName);
public java.util.Enumeration HttpServletRequest.getHeaders(String
p_headerName);
public java.util.Enumeration HttpServletRequest.getHeaderNames();
```

Os métodos getHeader (), getDateHeader () e getIntHeader () servem para recuperar o conteúdo de um header e se comportam de maneira análoga aos respectivos métodos set...Header() apresentados na seção 4.2 desse livro: o método getDateHeader () recupera o conteúdo de um header contendo uma data, e o método getIntHeader () recupera o conteúdo de um header contendo um número inteiro.

O método getHeaders (), por sua vez, permite obter todos os valores associados a um determinado header: caso você saiba de antemão que um header irá ter diversos valores associados, você deve utilizar esse método para garantir a recuperação de todos esses valores.

Por fim, o método getHeaderNames () retorna uma lista enumerada de todos os nomes de headers recebidos. Com isso, você pode usar essa função para iterar na lista de todos os headers recebidos.

Um uso frequente para esses métodos é o de verificação do header user-agent, que indica o tipo e versão do software do cliente que enviou a requisição. Por meio desse header, você pode gerar um documento diferenciado para cada tipo de browser que acessa a sua aplicação.

O header accept também é bastante importante: ele indica os tipos de conteúdos aceitos pelo cliente. O Servlet ServletVerAccept a seguir verifica o conteúdo desse header e, dependendo de seu valor, retorna uma página de saudação em formato HTML ou WML; dessa forma, ele pode ser usado tanto para atender requisições de browsers normais da Internet, como também para atender requisições de browsers de telefones celulares (vide seção 4.3).

### Servlet "ServletVerAccept"

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Verifica o valor do header "accept" e retorna página HTML
// ou WML dependendo de seu conteúdo
public class ServletVerAccept extends HttpServlet {

    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response) throws IOException {

        PrintWriter l_pw = p_response.getWriter ();

        // Verificando conteúdo do header accept
        String l_acceptHeader = p_request.getHeader("accept");
        if((l_acceptHeader != null) && l_acceptHeader.equals
("text/vnd.wap.wml")) {
            // O cliente aceita conteúdo WML; primeiro temos que
definir o content-type
```

```
            p_response.setContentType("text/vnd.wap.wml");

            // Depois geramos o conteúdo WML
            l_pw.println("<?xml version=\"1.0\"?>");
            l_pw.println("<!DOCTYPE xml PUBLIC \"- //
WAPFORUM//DTD WML 1.1//EN\
                "\"http://www.wapforum.org/DTD/wml_1.1.xml\">");
            l_pw.print("<wml><card>");
            l_pw.print("<p align=\"center\">Olá</p>");
            l_pw.print("</card></wml>");
        }
        else {
            // O cliente não aceita conteúdo WML; geramos uma página
HTML
            l_pw.println("<HTML><BODY>");
            l_pw.println("<P>Olá</P>");
            l_pw.println("</BODY></HTML>");
        }
        l_pw.flush ();
    }

}
```

## 5.6 Upload de arquivos

Além dos campos normais de input, seleção e botões de um formulário HTML, existe um tipo de campo especial que permite o upload (ou transferência) de arquivos da máquina do usuário para o servidor.

### Exemplo de página HTML com formulário para upload de arquivo

```
<HTML><BODY>
    <FORM METHOD="POST"
ACTION="/livroservlets/ServletUploadArq/"
            ENCTYPE="multipart/form- data">
        Selecione o arquivo:<BR><INPUT TYPE="file"
NAME="arquivo"><BR><BR>
        <INPUT TYPE="submit" NAME="envio" VALUE="Enviar">
    </FORM>
```

```
</BODY></HTML>
```

Nesse formulário de exemplo, o campo arquivo é do tipo file; esse campo permite que seja selecionado um arquivo do disco local que será enviado junto com a requisição. Vale a pena observar também que existe um atributo adicional ENCTYPE, com conteúdo multipart/form-data, que deve ser especificado junto ao elemento FORM.

A maneira mais fácil de se tratar esse tipo de parâmetro de requisição em um Servlet é utilizando classes prontas de terceiros. Um exemplo de uma biblioteca que pode ser utilizada para esse tipo de tratamento pode ser encontrada no endereço http://sourceforge.net/projects/multpartrequest/ .

O Servlet ServletUploadArq a serguir utiliza essa biblioteca para tratar arquivos carregados pelo formulário HTML do exemplo anterior.

### Servlet "ServletUploadArq"

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

import http.utils.multipartrequest.ServletMultipartRequest;
import http.utils.multipartrequest.MultipartRequest;

// Trata arquivos transferidos do cliente para o servidor
public class ServletUploadArq extends HttpServlet {
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        PrintWriter l_pw = p_response.getWriter ();

        // Obtendo informações sobre o arquivo carregado
```

```
        // O primeiro parâmetro é a requisição, o segundo é o nome
de um diretório
        // temporário onde deverão ser armazenados os arquivos
carregados no servidor,
        // o terceiro é o tamanho máximo em bytes permitidos, e o
quarto pode
        // definir o encoding a ser utilizado pelo parser (o default é
“ISO-8859-1”)
        MultipartRequest l_parser = new ServletMultipartRequest
(p_request,
            “C:\\temp”, MultipartRequest.MAX_READ_BYTES, null);


        String l_nomeArq = l_parser.getBaseFilename(“arquivo”);//
Nome do arquivo


        // Tamanho do arquivo
        String l_tamArq = Long.toString(l_parser.getFileSize
(“arquivo”));


        // Nome local do arquivo: o conteúdo do arquivo carregado é
gravado em um
        // arquivo temporário local abaixo do diretório especificado
no construtor da
        // classe ServletMultipartRequest
        File l_arqLocal = l_parser.getFile(“arquivo”);
        String l_nomeArqLocal = l_arqLocal.getName ();


        // Imprimindo header HTML
        l_pw.println(“<HTML><BODY>”);


        // Imprimindo informações sobre o arquivo recebido
        l_pw.println(“Nome do arquivo: ” + l_nomeArq + “<BR>”);
        l_pw.println(“Tamanho do arquivo: ” + l_tamArq + “<BR>”);
        l_pw.println(“Nome do arquivo temporário local: ” +
l_nomeArqLocal + “<BR>” );


        // Imprimindo rodapé HTML
        l_pw.println(“</BODY></HTML”);
        l_pw.flush ();
    }
}
```

No Servlet anterior, usamos alguns dos métodos disponíveis na biblioteca MultipartRequest. Para conhecer todos as suas funcionalidades, você deve fazer o download da biblioteca, e observar a documentação de sua API.

## 5.7 Atributos da requisição

Finalmente, além de todos os métodos apresentados anteriormente, existem métodos para definir e recuperar atributos da requisição. Esses atributos funcionam como os atributos do contexto do Servlet apresentados na seção 3.6, exceto que seu escopo está limitado à requisição.

**Métodos para definir / recuperar atributos da requisição**

```
public String HttpServletRequest.getAttribute( String
p_attributeName );
public util.Enumeration HttpServletRequest.getAttributeNames ();
public void HttpServletRequest.setAttribute( String p_attributeName,
Object p_attributeValue );
```

Embora estejamos falando desses métodos pela primeira vez nesse capítulo, exemplos e uma explicação mais detalhada de seu uso serão apresentados no *Capítulo 8 – Modelo MVC*.

# Capítulo 6

# Servlets – Cookies e Sessões

Nesse capítulo são explorados os conceitos de Cookies e Sessões: através dessas técnicas, veremos como é possível armazenar informações sobre os usuários que utilizam nossas aplicações.

## 6.1 Armazenando informações do usuário

Como explicamos na seção 3.2, o HTTP é um protoloco stateless, ou seja, ele não mantém um histórico das requisições recebidas de um mesmo cliente.

Assim, imaginando uma aplicação simples como um carrinho de compras de uma livraria virtual, por exemplo, como devemos proceder para manter o histórico dos livros já selecionados pelo nosso cliente? Se a seleção de cada livro gera 1 ou mais requisições, no momento do fechamento da compra, como fazemos para saber quais foram todos os livros selecionados?

Existem algumas maneiras diferentes de resolver esse problema, de maneira a contornar essa limitação do protocolo HTTP, como veremos a seguir.

## 6.2 Campos escondidos de formulários HTML

Além dos tipos de campos de formulários HTML apresentados na seção 5.3 desse livro, existe também um tipo, chamado “hidden”, que pode ser usado para esconder parâmetros da requisição.

Na página HTML seguinte, o campo contador é do tipo “hidden”.

**Exemplo de formulário HTML com campo “hidden”**

```
<HTML><BODY>
   <FORM METHOD=”POST”
ACTION=”/livroservlets/ServletContadorReq/”>
      Contador de requisições: 0<BR>
      <INPUT TYPE=”hidden” NAME=”contador”
VALUE=”0”><BR><BR>
      <INPUT TYPE=”submit” NAME=”Atualizar”
VALUE=”Atualizar”>
   </FORM>
</BODY></HTML>
```

Se você tentar abrir essa página em um browser, você poderá observar que o campo contador não é visível; por outro lado, o nome e valor desse campo serão recebidos normalmente como um parâmetro de requisição pelo seu Servlet.

Assim, podemos construir o Servlet a seguir para contar o número de requisições recebidas de cada cliente:

**Servlet “ServletContadorReq”**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


// Conta o número de requisições recebidas de cada cliente
public class ServletContadorReq extends HttpServlet {
   public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
         throws IOException {
```

```
        PrintWriter l_pw = p_response.getWriter ();

        // Atualizando o contador para o cliente corrente (dessa
requisição)
        // Em particular, se essa for a primeira requisição recebida,
        // o valor do parâmetro "contador" é nulo, e, nesse caso
inicializamos
        // o contador com o valor 0.
        int l_contador = 0;
        try {
            l_contador = Integer.parseInt(p_request.getParameter
("contador"));
            ++ l_contador;
        }catch(NumberFormatException p_e) {
            l_contador = 0;
        }

        // Apresentando o valor do contador e o formulário HTML que
irá
        // permitir o incremento do mesmo
        l_pw.println("<HTML><BODY>");
        l_pw.println("<FORM METHOD=\"POST\"
ACTION=\"/livroservlets
            ServletContadorReq/\">");
        l_pw.println("Contador de requisições: "+Integer.toString
(l_contador)
            +"<BR>");
        l_pw.println("<INPUT TYPE=\"hidden\" NAME=\"contador\"
VALUE=\""
            + Integer.toString(l_contador) + "\"><BR><BR>");
        l_pw.println("<INPUT TYPE=\"submit\" NAME=\"Atualizar\"
            VALUE=\"Atualizar\">");
        l_pw.println("</FORM>");
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }

    // Nesse Servlet, os métodos GET e POST são tratados da mesma
maneira
    // (pelo mesmo trecho de código)
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
```

```
            doGet(p_request, p_response);
        }
    }
```

O Servlet ServletContadorReq anterior usa o campo escondido contador para armazenar o valor do contador na última requisição recebida, sendo que esse valor é repassado de volta para o Servlet cada vez que o cliente pressiona o botão Atualizar.

Assim como acontece nesse Servlet, você pode utilizar campos escondidos de formulários HTML para armazenar informações e fazer com que sejam passadas para seus Servlets junto com as requisições futuras recebidas.

Por outro lado, esse método tem uma grande desvantagem: para utilizá-lo, você precisa garantir que, uma vez armazenado um valor em um campo escondido, todas as requisições recebidas carregarão esse parâmetro junto.

Isso pode gerar um grande trabalho na construção de sua aplicação, além de não funcionar em todos os casos: se o usuário de sua aplicação sair de sua página, ou até mesmo usar os botões de navegação de seu browser (como o botão de voltar, por exemplo), ele automaticamente irá perder os últimos parâmetros definidos.

## 6.3 Informações adicionais de caminho

Outra forma de armazenar / passar informações entre requisições subsequentes de um mesmo cliente é utilizando as informações adicionais de caminho.

O Servlet ServletContadorReq da seção anterior, por exemplo, poderia ser modificado para passar o valor atual do contador como uma informação adicional do caminho.

### Servlet “ServletContadorReqMod”

```
import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;

// Conta o número de requisições recebidas de cada cliente
public class ServletContadorReqMod extends HttpServlet {
    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        PrintWriter l_pw = p_response.getWriter ();

        // Extraindo informações adicionais de caminho
        String l_infoPath = p_request.getPathInfo ();
        if( (l_infoPath != null) && l_infoPath.startsWith("/") &&
            (l_infoPath.length () > 1)) l_infoPath = l_infoPath.substring
(1);

        // Atualizando o contador para o cliente corrente (dessa
requisição)
        // Em particular, se essa for a primeira requisição recebida, o
valor do
        // parâmetro "contador" é nulo, e, nesse caso inicializamos o
contador
        // com o valor 0.
        int l_contador = 0;
        try {
            l_contador = Integer.parseInt(l_infoPath);
            ++ l_contador;
        }catch(NumberFormatException p_e) {
            l_contador = 0;
        }

        // Apresentando o valor do contador e o formulário HTML que
irá
        // permitir o incremento do mesmo
        l_pw.println("<HTML><BODY>");
        l_pw.println("Contador de requisições: " + Integer.toString
(l_contador)
            + "<BR><BR>");
        l_pw.println("<A
HREF=\"/livroservlets/ServletContadorReqMod/"
            + Integer.toString(l_contador) + "\">Atualizar</A>");
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
```

```
    }

    // Nesse Servlet, os métodos GET e POST são tratados da mesma
maneira
    // (pelo mesmo trecho de código)
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        doGet(p_request, p_response);
    }
}
```

Em relação a utilização de campos escondidos, essa técnica possui a vantagem de simplificar um pouco a construção de seus Servlets. Por outro lado, ela não resolve os problemas apontados anteriormente com relação a perda dos parâmetros definidos.

## 6.4 Cookies

Para resolver esse problema da persistência das informações associadas a um cliente, é necessária a utilização de “Cookies”.

Cookies são pacotes de dados, gerados pelo servidor, e que são enviados junto com a resposta de uma requisição, ficando armazenadas pelo browser do usuário. Posteriormente, a cada requisição enviada, o browser anexa também as informações desses Cookies, permitindo que o servidor recupere os valores definidos anteriormente.

Um estudo completo de Cookies foge ao escopo desse livro; para obter mais detalhes sobre eles, deve-se consultar a RFC 2109 (mais informações no site http://www.apps.ietf.org/rfc/rfc2109.html), que contém sua especificação.

Por outro lado, vale a pena conhecer os principais atributos de um Cookie; além de um nome, que identifica o Cookie para um determinado servidor e path nesse servidor, e o valor, que contém os dados em si, um Cookie possui também alguns atributos adicionais:

Comentário (Comment): deve conter um texto descrevendo o propósito e função do Cookie;

Período de expiração (MaxAge): determina por quanto tempo (em segundos) o Cookie será válido;

Domínio (Domain): por default, uma vez definido, um Cookie só é retornado junto com requisições para o mesmo servidor que o gerou. Esse atributo permite, por outro lado, que esse Cookie seja enviado para todos os servidores abaixo de um mesmo domínio (conforme especificado na RFC 2109);

Caminho (Path): por default, uma vez definido, um Cookie só é passado junto com as requisições para os recursos abaixo do diretório virtual do recurso que gerou o Cookie. Se esse caminho for definido, por outro lado, o Cookie passa a ser enviado junto com as requisições para qualquer recurso abaixo caminho especificado;

A API de Servlets disponibiliza uma classe especial para representar um Cookie, a classe javax.servlet.http.Cookie, sendo que os métodos dessa classe permitem definir / recuperar os valores de seus diversos atributos.

### Métodos para definir / recuperar os principais atributos do Cookie

```
public String getName ();
public String getValue ();
public void setValue(String p_value);
public String getComment ();
public void setComment(String p_comment);
public String getDomain ();
public void setDomain(String p_domain);
public String getPath ();
public void setPath(String p_path);
```

```
public int getMaxAge ();
public void setMaxAge(int p_maxAge);
```

Para usar um Cookie, primeiro devemos criá-lo, passando como parâmetros de seu construtor o nome e valor do Cookie, e, em seguida, devemos definir seus atributos. Finalmente, o Cookie deve ser adicionado ao objeto HttpServletResponse para que seja enviado para o browser do usuário.

### Exemplo de código com definição de Cookie

```
...
Cookie l_cookie = new Cookie("Contador", "0");
l_cookie.setComment("Cookie de exemplo / teste");
l_cookie.setMaxAge(60);
p_response.addCookie(l_cookie);
...
```

No código anterior, estamos gerando um Cookie de nome Contador, com o valor 0, e que terá validade de 60 segundos a partir do instante em que for recebido pelo usuário; passado esse período, o Cookie é automaticamente descartado pelo browser. Um valor negativo para o período de expiração indica que o Cookie deve ser válido enquanto o browser não for encerrado, e o valor 0 indica que o Cookie deve ser removido imediatamente pelo browser.

É importante observar também que, a menos que você esteja fazendo o controle do buffer de resposta conforme descrito na seção 4.6, esse procedimento de criação e adição de Cookies a resposta do Servlet deve acontecer antes de gerar o conteúdo da resposta propriamente dito, de forma a preservar a ordem dos elementos HTTP da resposta do Servlet.

Após gerarmos todos os Cookies desejados, podemos recuperá-los nas requisições seguintes através do método getCookies() da classe HttpServletRequest. Esse método retorna null se não houver nenhum Cookie na requisição recebida, ou um Array com os Cookies recebidos.

Infelizmente, não existe um método para a recuperação direta de um Cookie específico; é necessário utilizar esse método para recuperar todos os Cookies, e, então, iterar nessa lista até acharmos o Cookie que estamos procurando.

Assim, podemos utilizar o trecho de código a seguir para obter o valor do Cookie Contador definido anteriormente.

**Exemplo de código para obtenção de um Cookie**

```
...
Cookie [] l_cookies = p_request.getCookies ();
If(l_cookies != null)
    for(int i=0; i<l_cookies.length; i++)
        if((l_cookies[i].getName () != null) && (l_cookies[i].getName
            ().equals("Contador")))
            try {
                l_contador = Integer.parseInt(l_cookies[ i ].getValue ());
            } catch(NumberFormatException p_e) {
                l_contador = 0;
            }
...
```

Com os métodos e recursos apresentados, podemos reimplementar nosso Servlet ServletContadorReq da seguinte forma:

**Reimplementação do Servlet "ServletContadorReq"**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Conta o número de requisições recebidas de cada cliente
public class ServletContadorReq extends HttpServlet {
    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {

        // Definindo headers auxiliares para evitar cacheamento da
página
        p_response.setHeader("Cache- Control", "no- cache, must-
revalidate");
```

```
        p_response.setHeader("Pragma", "no-cache");
        p_response.setHeader("Expires", "Mon, 26 Jul 1997 05:00:00
GMT");
        p_response.setDateHeader("Last-Modified",
System.currentTimeMillis());

        // Atualizando o contador para o cliente corrente (dessa
requisição)
        // Em particular, se essa for a primeira requisição recebida,
        // o valor do parâmetro "contador" é nulo, e, nesse caso
inicializamos
        // o contador com o valor 0.
        // OBS: procuramos o valor do contador entre os cookies
        // recebidos com a requisição
        Cookie [] l_cookies = p_request.getCookies();
        int l_contador = 0;
        if(l_cookies != null)
            for(int i=0; i<l_cookies.length; i++)
                if((l_cookies[i].getName() != null) && (l_cookies[i].
getName
                    ().equals("Contador")))
                try {
                    l_contador = Integer.parseInt(l_cookies[i].getValue
());
                    l_contador ++;
                } catch(NumberFormatException p_e) {
                    l_contador = 0;
                }

        // Guardando o valor corrente do contador em um Cookie
        Cookie l_cookie = new Cookie("Contador", Integer.toString
(l_contador));
        l_cookie.setComment("Cookie com contador para
ServletContadorReq");
        l_cookie.setMaxAge(60 * 60 * 24);  // Validade = 60segundos *
60minutos * 24horas = 1 dia
        p_response.addCookie(l_cookie);

        // Apresentando o valor do contador
        PrintWriter l_pw = p_response.getWriter();
        l_pw.println("<HTML><BODY>");
```

```
        l_pw.println("Contador de requisições: " + Integer.toString
(l_contador)
            + "<BR><BR>");
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }

    // Nesse Servlet, os métodos GET e POST são tratados da mesma
maneira
    // (pelo mesmo trecho de código)
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        doGet(p_request, p_response);
    }
}
```

Nessa nova implementação do Servlet, vale a pena observar que não é necessário que o usuário de nossa aplicação clique em nada para que o contador seja atualizado, basta que ele recarregue a página. Até mesmo se o usuário visitar outra página e depois voltar para o endereço desse Servlet, o contador é incrementado.

Infelizmente, existem situações em que Cookies não funcionam: geralmente isso acontece quando o browser do usuário está configurado explicitamente para não aceitar Cookies. Com as crescentes preocupações em torno da privacidade de quem navega na Internet, alguns usuário passaram a bloquear o armazenamento de Cookies em seus browsers.

## 6.5 Gerenciamento de sessões

A API de Servlets disponibiliza um módulo extremamente útil no controle de informações associadas ao usuário que acessa nossa aplicação: o módulo de gerenciamento de sessões de usuários.

Basicamente, esse módulo funciona da seguinte forma: a primeira vez que o usuário acessa nossa aplicação, é criado para ele um identificador de sessão, ao qual é

associado um objeto da classe javax.servlet.http.HttpSession na memória do servidor de aplicações. A partir daí, o servidor de aplicações procura fazer com que todas as requisições vindas daquele usuário carreguem esse identificador de sessão, seja através da definição de Cookies, ou através da reescrita das URLs (com informações adicionais de caminho) para que incorporem essa informação. Recebido esse identificador, a API automaticamente disponibiliza para o Servlet o objeto HttpSession criado anteriormente.

Dessa forma, um Servlet pode utilizar esse objeto HttpSession em memória para armazenar informações que queira associar a um usuário específico que acessa a aplicação. Essas informações estarão disponíveis em todas as requisições posteriores recebidas não só pelo Servlet que as armazenou, mas também para todos os Servlets que compartilharem do mesmo contexto.

Para obter uma referência ao objeto dessa classe HttpSession, um Servlet deve utilizar o método getSession () da classe HttpServletRequest; esse método possui, na verdade, duas assinaturas.

**Assinaturas para o método "getSession ()" da classe "HttpServletRequest"**

```
public javax.servlet.http.HttpSession getSession();
public javax.servlet.http.HttpSession getSession(boolean
p_createSession);
```

A primeira assinatura retorna sempre um objeto da sessão, sendo que esse objeto pode ter sido criado nesse instante, ou ter sido criado em uma requisição anterior (ou seja, já havia uma sessão criada).

A segunda assinatura, por sua vez, admite um parâmetro que, se for true, faz com que o método tenha o mesmo comportamento da primeira assinatura. Caso contrário, o método pode retornar null: se ainda não houver uma sessão criada, ao invés de criar uma nova sessão, o método retorna null.

Podemos, então, utilizar o seguinte trecho de código para obter o objeto com a sessão corrente:

**Primeira alternativa de código para obtenção do objeto "HttpSession" para a sessão corrente**

```
...
HttpSession l_session = p_request.getSession( false );
if(l_session != null)
{
    // Tratamento do objeto com sessão
    ...
}
else
{
    // Sessão ainda não foi criada, é necessário se criar uma nova
sessão.
    ...
}
...
```

O código anterior utiliza a segunda assinatura do método getSession() para obter a sessão corrente. Uma alternativa melhor de código é apresentada a seguir, utilizando a primeira assinatura do método getSession ():

**Código para obtenção do objeto "HttpSession" para a sessão corrente**

```
...
HttpSession l_session = p_request.getSession();
if(l_session.isNew ())
{
    // Uma nova sessão acabou de ser criada; código para tratar esse
caso
    ...
}
else
{
    // A sessão já havia sido criada em uma requisição anterior;
    // código para tratar esse outro caso
    ...
}
```

```
...
```

No código anterior, podemos observar também a utilização de um método isNew() da classe HttpSession: esse método indica se o identificador da sessão já foi enviado para o browser cliente ou não (ou seja, se a sessão acabou de ser criada). Além desse método, a classe HttpSession oferece diversos outros métodos:

**Métodos da classe "HttpSession"**

```
public void setAttribute(String p_attributeName, Object
p_attributeValue);
public Object getAttribute(String p_attributeName);
public void removeAttribute(String p_attributeName);
public java.util.Enumeration getAttributeNames ();
public String getId ();
public long getCreationTime ();
public long getLastAccessedTime ();
```

Os primeiros quatro métodos apresentados permitem gerenciar os objetos que queremos associar a sessão do usuário de forma similar a manipulação de uma Hashtable. Dessa forma, para cada objeto que queremos armazenar na sessão HTTP, nós devemos associar um nome de atributo, e manipular o objeto como valor desse atributo.

Assim, se quisermos definir um atributo de sessão de nome Contador, contendo o valor do contador de requisições usados nos Servlets ServletContadorReq, podemos utilizar o seguinte trecho de código:

**Código com exemplo de definição de atributo de sessão**

```
...
HttpSession l_sessao = p_request.getSession(true);
l_sessao.setAttribute("Contador", new Integer(l_contador));
...
```

Para recuperar o valor desse atributo posteriormente, podemos utilizar esse outro trecho de código:

**Código com exemplo de recuperação de atributo de sessão**

```
...
HttpSession l_sessao = p_request.getSession(true);
Integer l_contadorInt = (Integer) l_sessao.getAttribute("Contador");
...
```

Os três métodos seguintes, getId (), getCreationTime () e getLastAccessedTime (), retornam, respectivamente, o identificador dessa sessão, o instante de criação da sessão e o instante de último acesso do usuário com a sessão corrente (esses dois últimos valores em milisegundos, desde 1º de janeiro de 1970 GMT).

O Servlet a seguir utiliza os métodos apresentados anteriormente para construir uma página HTML com todas as informações sobre a sessão corrente do usuário:

**Servlet "ServletInfoSessao"**

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Imprime informações sobre a sessão corrente do usuário
public class ServletInfoSessao extends HttpServlet {
    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
          throws IOException {

       // Definindo headers auxiliares para evitar cacheamento da
página
       p_response.setHeader("Cache- Control", "no- cache, must-
revalidate");
       p_response.setHeader("Pragma", "no- cache");
       p_response.setHeader("Expires", "Mon, 26 Jul 1997 05:00:00
GMT");
       p_response.setDateHeader("Last- Modified",
System.currentTimeMillis ());

       // Imprimindo header HTML
       PrintWriter l_pw = p_response.getWriter ();
       l_pw.println("<HTML><BODY>");
```

```
        // Obtendo informações sobre a sessão corrente do usuário
        HttpSession l_sessao = p_request.getSession(true);
        if(!l_sessao.isNew ())
        {
            l_pw.println("ID da sessão: " + l_sessao.getId () + "<BR>");
            l_pw.println("Instante de criação da sessão (em
milisegundos, desde epoch): "
                + Long.toString(l_sessao.getCreationTime ()) + "<BR>");
            l_pw.println("Instante de último acesso (em milisegundos,
desde epoch): "
                + Long.toString( l_sessao.getLastAccessedTime ()) +
"<BR>");
            Enumeration l_nomesAtribs = l_sessao.getAttributeNames
();
            while( l_nomesAtribs.hasMoreElements ())
            {
                String l_nomeAtrib = (String) l_nomesAtribs.nextElement
();
                Object l_objSessao = l_sessao.getAttribute(l_nomeAtrib);
                if(l_objSessao instanceof java.lang.String)
                    l_pw.println("Nome de Atributo: (" + l_nomeAtrib +
") => Valor de
                        Atributo: (" + (String) l_objSessao + ")<BR>");
                else l_pw.println("Nome de Atributo: (" + l_nomeAtrib +
") => Valor de
                        Atributo (Objeto não String)<BR>");
            }
        }
        else l_pw.println("Nova sessão criada!<BR>");

        // Footer HTML
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }

    // Nesse Servlet, os métodos GET e POST são tratados da mesma
maneira
    // (pelo mesmo trecho de código)
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        doGet(p_request, p_response);
```

```
        }
    }
```

A outra característica importante desse módulo de gerenciamento de sessões é que ele permite que sessões criadas automaticamente expirem após um determinado tempo de inatividade. Obviamente, o desenvolvedor também pode expirar uma sessão explicitamente através de sua programação.

Essa funcionalidade, presente em diversas tecnologias de desenvolvimento de aplicações para a Web, pode ser observada, por exemplo, em sites que exigem a autenticação do usuário, como por exemplo, em um site de Internet Banking.

Nesses sites, após o login, se o usuário ficar um longo período de tempo sem interagir com o site, sua sessão é "expirada", de forma que ele não consiga utilizar mais nenhuma função do site antes de efetuar o login novamente. O mesmo ocorre se o usuário clicar no botão ou link de sair / logoff.

Isso ocorre dessa maneira para reforçar a segurança do sistema; uma segunda pessoa que utilize o mesmo browser do usuário que acessou a aplicação terá que efetuar o login para conseguir ter acesso as funções do sistema. Caso contrário esse segundo usuário poderia simplesmente usar o sistema em nome do primeiro usuário.

A API de Servlets oferece as seguintes funções para controlar a expiração da sessão do usuário.

### Métodos para controle da expiração da sessão

```
public int getMaxInactiveInterval ();
public void setMaxInactiveInterval(int p_interval);
public void invalidate ();
```

As duas primeiras funções permitem, respectivamente, obter e definir o período máximo de inatividade em segundos antes que o ServletContainer invalide a sessão do usuário, e o terceiro método permite que essa sessão

seja expirada (ou invalidada) explicitamente pela aplicação.

Para exemplificar o uso de todas as funções apresentadas nessa seção, vamos construir um “esqueleto” de uma aplicação que requer a autenticação de usuários. Mais tarde, você poderá utilizar os códigos apresentados a seguir para construir sua própria aplicação “autenticada”.

Primeiro devemos construir um Servlet que apresenta a tela de login propriamente dita:

### Servlet “FormLogin”

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Apresenta página HTML com formulário de login
public class FormLogin extends HttpServlet {
    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {

        // Definindo headers auxiliares para evitar cacheamento da
página
        p_response.setHeader(“Cache- Control”, “no- cache, must-
revalidate”);
        p_response.setHeader(“Pragma”, “no- cache”);
        p_response.setHeader(“Expires”, “Mon, 26 Jul 1997 05:00:00
GMT”);
        p_response.setDateHeader(“Last- Modified”,
System.currentTimeMillis ());

        // Imprimindo header HTML
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println(“<HTML><BODY>”);

        // A sessão é criada nesse Servlet; se os demais Servlets da
aplicação
        // receberem requisições sem o objeto de sessão criado,
devem redirecionar
        // a requisição para esse Servlet
        HttpSession l_sessao = p_request.getSession(true);
```

```
        if(!l_sessao.isNew ()) {
            // A sessão já havia sido criada em outra requisição:
devemos verificar
            // se há alguma mensagem a ser apresentada para o
usuário e zerar a
            // informação de login se estiver presente na sessão (afinal,
se o usuário
            // chegou até essa página, é porque deseja fazer o login
novamente)
            String l_msg = (String) l_sessao.getAttribute("MSGLOGIN");
            if(l_msg != null) l_pw.println(l_msg + "<BR>");
            l_sessao.removeAttribute("LOGIN");
        }

        //  Imprimindo o restante da página HTML
        l_pw.println("<FORM ACTION=\"Login.html\"
METHOD=\"POST\">");
        l_pw.println("Login: <INPUT TYPE=\"TEXT\"
NAME=\"LOGIN\"><BR>");
        l_pw.println("Senha: <INPUT TYPE=\"PASSWORD\"
NAME=\"SENHA\"><BR>");
        l_pw.println("<INPUT TYPE=\"SUBMIT\" NAME=\"ENTRAR\"
VALUE=\"Entrar\"><BR>");
        l_pw.println("</FORM>");
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }

    // Nesse Servlet, os métodos GET e POST são tratados da mesma
maneira
    // (pelo mesmo trecho de código)
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        doGet(p_request, p_response);
    }
}
```

Em seguida, temos um Servlet que faz, efetivamente, a autenticação. Se a autenticação for bem sucedida, esse Servlet armazena, como atributo da sessão, o nome do usuário, e redireciona a requisição para o menu principal de nossa aplicação.

## Servlet “Login”

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Efetua a autenticação do usuário / senha preenchidos no formulário de login
public class Login extends HttpServlet {

    // Verifica se existe usuário com o login e senha passados como parâmetro
    private boolean ValidaLoginSenha(String p_login, String p_senha) {
        boolean l_sucesso = false;
        if((p_login != null) && (p_senha != null)) {
            // Você deve implementar aqui o código para validar o login e senha do usuário
            // (por exemplo, consultando uma base de dados)
        }
        return l_sucesso;
    }

    public void doGet(HttpServletRequest p_request, HttpServletResponse p_response)
            throws IOException {

        // Definindo headers auxiliares para evitar cacheamento da página
        p_response.setHeader(“Cache-Control”, “no-cache, must-revalidate”);
        p_response.setHeader(“Pragma”, “no-cache”);
        p_response.setHeader(“Expires”, “Mon, 26 Jul 1997 05:00:00 GMT”);
        p_response.setDateHeader(“Last-Modified”, System.currentTimeMillis());

        // Recuperando dados da sessão: se a sessão não tiver sido criada ainda,
        // redirecionamos a requisição para o Servlet “LoginForm”, que é o ponto de
        // entrada da aplicação; caso contrário, removemos o atributo “LOGIN” da
        // sessão, pois ele só deverá estar definido caso o processo de
```

```
autenticação
        // seja bem sucedido.
        HttpSession l_sessao = p_request.getSession(false);
        if(l_sessao == null) {
            p_response.sendRedirect("LoginForm.html");
            return;
        }
        l_sessao.removeAttribute( "LOGIN" );


        // Recuperando os parâmetros com login e senha e validando
esses dados
        String l_login = p_request.getParameter("LOGIN");
        String l_senha = p_request.getParameter("SENHA");
        if(ValidaLoginSenha(l_login, l_senha)) {
            // Login e senha são válidos, podemos definir o atributo de
    sessão "LOGIN"
            // (contendo o login do usuário) e podemos redirecionar a
    requisição para          // o menu principal da aplicação
            l_sessao.setAttribute("LOGIN", l_login);
            p_response.sendRedirect("MenuPrincipal.html");
        }
        else {
            // Se o login e senha não forem válidos, colocamos uma
mensagem de falha
            // na autenticação junto a sessão do usuário corrente, e
redirecionamos a
            // requisição para o Servlet "LoginForm"
            l_sessao.setAttribute("MSGLOGIN", "Login e/ou senha
inválidos!");
            p_response.sendRedirect("LoginForm.html");
        }


    }


    // Nesse Servlet, os métodos GET e POST são tratados da mesma
maneira
    // (pelo mesmo trecho de código)
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        doGet( p_request, p_response );
    }
```

```
}
```

Um detalhe interessante com relação ao Servlet anterior é que ele não retorna nenhum conteúdo HTML por si, ele simplesmente efetua um processamento e, conforme seu resultado, redireciona a requisição para Servlets que apresentarão um conteúdo. Esse modelo de funcionamento será explorado com mais detalhes no *Capítulo 8 – Modelo MVC*.

Finalmente, teremos o Servlet que apresenta o menu principal da aplicação (caso a autenticação tenha sido bem sucedida).

### Servlet "MenuPrincipal"

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Apresenta o menu principal da aplicação
public class MenuPrincipal extends HttpServlet {

    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response) throws IOException {

        // Definindo headers auxiliares para evitar cacheamento da
página
        p_response.setHeader("Cache- Control", "no- cache, must-
revalidate");
        p_response.setHeader("Pragma", "no- cache");
        p_response.setHeader("Expires", "Mon, 26 Jul 1997 05:00:00
GMT");
        p_response.setDateHeader("Last- Modified",
System.currentTimeMillis ());

        // Recuperando dados da sessão: o acesso a esse Servlet só é
permitido se a
        // sessão já tiver sido criada e contiver o login do usuário
como atributo de
        // sessão; se uma dessas condições falhar, é porque o usuário
corrente não
        // efetuou a autenticação
        String l_login = null;
```

```
        HttpSession l_sessao = p_request.getSession(false);
        if(l_sessao != null) l_login = (String) l_sessao.getAttribute
("LOGIN");
        if(l_login == null) {
            p_response.sendRedirect("LoginForm.html");
            return;
        }

        // Imprimindo o menu principal da aplicação
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");
        l_pw.println("Olá, " + l_login + ", este é o menu principal de
sua
            aplicação<BR>");
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }

    // Nesse Servlet, os métodos GET e POST são tratados da mesma
maneira
    // (pelo mesmo trecho de código)
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        doGet(p_request, p_response);
    }
}
```

Caso a aplicação venha a ter mais Servlets "autenticados" (que só poderão ser acessados após o login do usuário), pode valer a pena criar uma hierarquia de classes para facilitar o processo de desenvolvimento.

Assim, podemos criar uma classe abstrata ServletLogado que faz a validação da sessão do usuário, verificando se o usuário já efetuou o login, e, se isso não tiver acontecido, redirecionando a requisição para o Servlet FormLogin.

### Servlet "ServletLogado"

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
// Classe abstrata para verificar se usuário já efetuou o login
public abstract class ServletLogado extends HttpServlet {

    // Login do usuário: se a autenticação já tiver sido feita,
    // colocamos nessa variável o login do usuário que fez a
autenticação
    protected String m_login = null;

    // Servlets que necessitarem da autenticação devem extender
    // essa classe e implementar o método abaixo
    public abstract void doLoggedGet(HttpServletRequest p_request,
            HttpServletResponse p_response) throws IOException;

    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {

        // Definindo headers auxiliares para evitar cacheamento da
página
        p_response.setHeader("Cache- Control", "no- cache, must-
revalidate");
        p_response.setHeader("Pragma", "no- cache");
        p_response.setHeader("Expires", "Mon, 26 Jul 1997 05:00:00
GMT");
        p_response.setDateHeader("Last- Modified",
System.currentTimeMillis ());

        // Recuperando dados da sessão: o acesso a esse Servlet só é
permitido se a
        // sessão já tiver sido criada e contiver o login do usuário
como atributo de
        // sessão; se uma dessas condições falhar, é porque o usuário
corrente não
        // efetuou a autenticação
        HttpSession l_sessao = p_request.getSession(false);
        if(l_sessao != null) m_login = (String) l_sessao.getAttribute
("LOGIN");
        if(m_login == null) {
            p_response.sendRedirect("LoginForm.html");
            return;
        }
```

```
        doLoggedGet(p_request, p_response);
    }

    // Nesse Servlet, os métodos GET e POST são tratados da mesma
maneira
    // (pelo mesmo trecho de código)
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        doGet(p_request, p_response);
    }
}
```

Utilizando a classe anterior, podemos criar tantos Servlets "autenticados" quanto quisermos: basta que esses Servlets estendam a classe ServletLogado e implementem o método doLoggedGet(), em vez do método doGet(). Poderíamos, por exemplo, implementar o Servlet MenuPrincipal da seguinte forma:

### Servlet "MenuPrincipal"

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Apresenta o menu principal da aplicação
public class MenuPrincipal extends ServletLogado {

    public void doLoggedGet(HttpServletRequest p_request,
HttpServletResponse
            p_response) throws IOException {

        // Como esse Servlet estende a classe "ServletLogado", ao
chegar nesse trecho          // de código já foi feita a validação e já
foram definidos os headers
        // principais, fica restando só imprimir a página com o menu
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");
        l_pw.println("Olá, " + m_login + ", este é o menu principal de
sua
            aplicação<BR>");
        l_pw.println("</BODY></HTML>");
```

```
        l_pw.flush ();
    }

    // Nesse Servlet, os métodos GET e POST são tratados da mesma
maneira
    // (pelo mesmo trecho de código)
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        doGet(p_request, p_response);
    }
}
```

# Capítulo 7

# Páginas JSP

Nos quatro capítulos anteriores, exploramos com detalhes como um Servlet pode tratar uma requisição HTTP e gerar sua resposta.

Infelizmente, as aplicações que pudemos desenvolver até agora tem uma séria limitação inerente a própria tecnologia de Servlets: a formatação do conteúdo da resposta está totalmente integrada a programação da lógica da aplicação. A tecnologia de páginas JSP, que estudaremos nesse capítulo, nos ajudará a transpor essa limitação.

## 7.1 Formatação do conteúdo da resposta com Servlets

Em nossos Servlets de exemplo apresentados nos capítulos anteriores, geramos, como resposta, páginas HTML simples, sem nenhuma formatação.

O primeiro Servlet apresentado (seção 1.2), por exemplo, pode gerar a seguinte página HTML ao receber uma requisição:

```
<HTML><BODY>
   O seu endereço IP é ”127.0.0.1”
</BODY></HTML>
```

Essa página HTML não tem, praticamente, nenhuma formatação; um “Webdesigner” sugeriria, com certeza,

diversas alterações no código HTML para tornar o layout da página mais atrativo. Assim, poderíamos alterar esse código minimamente para:

```
<HTML>
   <HEAD>
      <META HTTP-EQUIV="Content-Type" CONTENT="text/html;
charset=iso-8859-1">
      <TITLE>Endereço IP</TITLE>
   </HEAD>
   <BODY BGCOLOR="#FFFFFF">
      <IMG NAME="LOGO" SRC="/icons/logopagina.jpg"
BORDER="0"><BR>
      <P><FONT FACE="Arial" SIZE="2" COLOR="BLUE">
         <B><U>Página de exemplo – Apresentando o endereço IP
do cliente</U></B>
      </FONT></P>
      <P><FONT FACE="Arial" SIZE="2" COLOR="BLUE">O seu
endereço IP é
         <B>"127.0.0.1"</B></FONT></P>
      </BODY>
</HTML>
```

Apesar da página apresentar exatamente as mesmas informações, a formatação pode tornar o conteúdo muito mais atraente para o usuário que estiver visitando nossa aplicação.

Por outro lado, essa modificação faz com que a codificação do Servlet fique muito mais trabalhosa, pois agora a resposta do Servlet precisa conter todos esses dados de formatação adicionais.

Assim, se originalmente o código do método doGet() do Servlet era

```
   public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
         throws IOException {
      PrintWriter l_pw = p_response.getWriter ();
      l_pw.println("<HTML><BODY>");
      l_pw.println("O seu endereço IP é \"" +
p_request.getRemoteAddr () + "\"");
      l_pw.println("</BODY></HTML>");
```

```
        l_pw.flush ();
    }
```

, agora esse código passa a ser

```
    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
        throws IOException {

        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML>");
        l_pw.println("<HEAD>");
        l_pw.println("<META HTTP-EQUIV=\"Content-Type\"
CONTENT=\"text/html;
            charset=iso-8859-1\">");
        l_pw.println("<TITLE>Endereço IP</TITLE>");
        l_pw.println("</HEAD>");
        l_pw.println("<BODY BGCOLOR=\"#FFFFFF\">");
        l_pw.println("<IMG NAME=\"LOGO\"
SRC=\"/icons/logopagina.jpg\"
            BORDER=\"0\"><BR>");
        l_pw.println("<P><FONT FACE=\"Arial\" SIZE=\"2\"
COLOR=\"BLUE\">");
        l_pw.println("<B><U>Página de exemplo – Apresentando o
endereço IP do
            cliente</U></B>" );
        l_pw.println("</FONT></P>");
        l_pw.println("<P><FONT FACE=\"Arial\" SIZE=\"2\"
COLOR=\"BLUE\">O seu
            endereço IP é <B>\"127.0.0.1\"</B></FONT></P>");
        l_pw.println( "</BODY></HTML>" );
    }
```

Além disso, qualquer modificação na formatação dessa página torna necessário o envolvimento do programador, já que ele precisa incorporar essa modificação ao código do Servlet e recompilar seu código. Por exemplo, se nosso "Webdesigner" decidir que a cor do texto agora precisa ser verde, precisamos alterar o código anterior para

```
    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
        throws IOException {
        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML>");
        l_pw.println("<HEAD>");
```

```
        l_pw.println("<META HTTP-EQUIV=\"Content-Type\"
CONTENT=\"text/html; charset=iso-8859-1\">");
        l_pw.println("<TITLE>Endereço IP</TITLE>");
        l_pw.println("</HEAD>");
        l_pw.println("<BODY BGCOLOR=\"#FFFFFF\">");
        l_pw.println("<IMG NAME=\"LOGO\"
SRC=\"/icons/logopagina.jpg\"
            BORDER=\"0\"><BR>");
        l_pw.println("<P><FONT FACE=\"Arial\" SIZE=\"2\"
COLOR=\"BLUE\">");
        l_pw.println("<B><U>Página de exemplo – Apresentando o
endereço IP do
            cliente</U></B>");
        l_pw.println("</FONT></P>");
        l_pw.println("<P><FONT FACE=\"Arial\" SIZE=\"2\"
COLOR=\"GREEN\">O seu
            endereço IP é <B>\"127.0.0.1\"</B></FONT></P>");
        l_pw.println("</BODY></HTML>");
    }
```

Todo esse processo faz com que o desenvolvedor tenha um grande trabalho cada vez que a formatação da página precise ser modificada, tornando esse processo lento e pouco flexível.

## 7.2 Formatação do conteúdo da resposta com páginas JSP

Se em um Servlet a inclusão e a modificação da formatação da resposta precisam ser feitas pelo desenvolvedor diretamente no código da aplicação, em uma página JSP a idéia é que esse processo possa ser feito diretamente pelo responsável pelo "look'n'feel" (layout) da aplicação.

Assim, uma página JSP nada mais é, na verdade, que uma página HTML, com elementos especiais onde o desenvolvedor pode programar o conteúdo dinâmico da aplicação. Porém, ao contrário de uma página HTML cujo nome de arquivo tem extensão ".htm" ou ".html", arquivos com páginas JSP devem ser nomeados com extensão ".jsp".

A primeira versão do Servlet apresentado na seção anterior (sem nenhuma formatação), por exemplo, poderia ser reescrito como a seguinte página JSP "PrimPag.jsp"

**Primeiro exemplo de página JSP**

```
<HTML><BODY>
   O seu endereço IP é "<%= request.getRemoteAddr () %>"
</BODY></HTML>
```

Nessa página JSP, o conteúdo dinâmico está contido no elemento

```
<%= request.getRemoteAddr () %>
```

Desde que esse elemento não seja alterado, podemos incluir / alterar livremente a formatação da página sem afetar o funcionamento da aplicação. Podemos, portanto, modificar essa página de maneira a ter a segunda versão do Servlet apresentado na seção anterior:

**Segundo exemplo de página JSP**

```
<HTML>
   <HEAD>
      <META HTTP-EQUIV="Content-Type" CONTENT="text/html;
charset=iso-8859-1">
      <TITLE>Endereço IP</TITLE>
   </HEAD>
   <BODY BGCOLOR="#FFFFFF">
   <IMG NAME="LOGO" SRC="/icons/logopagina.jpg"
BORDER="0"><BR>
   <P><FONT FACE="Arial" SIZE="2" COLOR="BLUE">
   <B><U>Página de exemplo – Apresentando o endereço IP do
cliente</U></B>
   </FONT></P>
   <P><FONT FACE="Arial" SIZE="2" COLOR="BLUE">O seu
endereço IP é <B>"
      <%= request.getRemoteAddr () %>"</B></FONT></P>
</BODY></HTML>
```

Após fazer essa modificação, ao contrário do que acontece quando se trabalha diretamente com Servlets, você não precisa recompilar a aplicação, essa modificação já passa a ser automaticamente visível para os usuários.

## 7.3 Funcionamento interno

A mágica por trás de uma página JSP é a seguinte: existe um Servlet especial, chamado Page Compiler, que intercepta requisições direcionadas a recursos com extensão ".jsp".

No instante em que é recebida uma requisição para uma página JSP, o Page Compiler transforma essa página em um Servlet e o compila, sendo que o resultado dessa compilação é carregado em memória para evitar que esse processo tenha que ser repetido para todas as requisições recebidas.

A primeira versão de nossa página de exemplo PrimPag.jsp, por exemplo, é transformada no seguinte Servlet

**Primeira versão da página JSP "PrimPag.jsp" transformada em Servlet**

```
package org.apache.jsp;


import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;

public class pag1$jsp extends HttpJspBase {

    static {
    }
    public pag1$jsp( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws
org.apache.jasper.runtime.JspException {
    }
```

```
    public void _jspService(HttpServletRequest request,
HttpServletResponse
            response) throws java.io.IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String _value = null;
        try {
            if (_jspx_inited == false) {
                synchronized (this) {
                    if (_jspx_inited == false) {
                        _jspx_init();
                        _jspx_inited = true;
                    }
                }
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;ISO-8859-1");
            pageContext = _jspxFactory.getPageContext(this, request,
response,
                "", true, 8192, true);
            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            // HTML // begin [file="/pag1.jsp";from=(0,0);to=(1,21)]
            out.write("<HTML><BODY>\r\nO seu endereÃ§o IP Ã©
\"");

            // end
            // begin [file="/pag1.jsp";from=(1,24);to=(1,51)]
            out.print(request.getRemoteAddr ());
            // end
            // HTML // begin [file="/pag1.jsp";from=(1,53);to=(2,14)]
```

```
            out.write("\"\r\n</BODY></HTML>");

            // end

        } catch (Throwable t) {
            if (out != null && out.getBufferSize() != 0)
                out.clearBuffer();
            if (pageContext != null) pageContext.handlePageException
(t);
        } finally {
            if (_jspxFactory != null) _jspxFactory.releasePageContext
(pageContext);
        }
    }
}
```

O Page Compiler também verifica a data de alteração do arquivo que contém a página JSP: caso essa data se modifique, o processo de compilação é executado novamente para garantir que modificações feitas na página sejam visíveis para os usuários da aplicação.

Devido a todo esse processo de compilação / recompilação, você poderá observar que o primeiro acesso após a criação ou modificação de uma página JSP é sempre mais lento que os acessos seguintes (até que haja uma modificação no conteúdo da página).

## 7.4 Ciclo de vida

O fato de uma página JSP ser convertida para um Servlet faz com que ela tenha o mesmo ciclo de vida apresentado na seção 3.4 desse livro: existe uma etapa de inicialização, uma etapa de atendimento de requisições, e finalmente, uma etapa de finalização.

Não existem métodos equivalentes ao doGet() ou doPost() de um Servlet para a etapa de atendimento de requisições, já que o próprio conteúdo da página contém o código a ser executado e retornado para o browser a cada requisição.

Por outro lado, existem os métodos jspInit () e jspDestroy () que possibilitam a implementação de códigos de inicialização e finalização, respectivamente, da página JSP. A maneira pela qual esses dois métodos podem ser declarados para uma página JSP será apresentada na seção "Declarações" mais adiante nesse mesmo capítulo.

## 7.5 Elementos dinâmicos

Já vimos nas páginas JSP apresentadas até agora um exemplo de um elemento dinâmico; na página PrimPag.jsp apresentada na seção 7.2 desse capítulo, por exemplo, temos a seguinte linha contendo um elemento dinâmico:

```
O seu endereço IP é "<%= request.getRemoteAddr () %>"
```

, onde o conteúdo do elemento dinâmico está delimitado pelos caracteres <%= e %>.

Este é apenas um tipo de elemento dinâmico, chamado comumente de "expressão". Além desse, existem outros 4 tipos principais de elementos dinâmicos que podem estar presentes em uma página JSP: diretivas, scriptlets, declarações e JavaBeans.

As seções seguintes irão apresentar, de maneira mais detalhada, cada um desses tipos de elementos dinâmicos que poderão ser utilizados por você em suas páginas JSP.

## 7.6 Diretivas

O primeiro tipo de elemento dinâmico que iremos estudar será a diretiva.

O formato básico de uma diretiva é o seguinte:

**Formato básico de uma diretiva de uma página JSP**

```
<%@ diretiva nomeAtributo1="valorAtributo1"
nomeAtributo2="valorAtributo2" ... %>
```

, onde a palavra diretiva deve ser substituída por page, include ou taglib. Para cada um desses tipos de diretivas,

existem conjuntos de atributos específicos utilizados para parametrizar a diretiva.

Conforme o próprio nome indica, a diretiva page serve para se definir diretivas da página; embora existam diversos atributos possíveis para essa diretiva, os atributos mais comuns são os seguintes: info, contentType, import, errorPage e isErrorPage.

O atributo "info" deve ser utilizado para se definir um texto informativo sobre a página sendo construída; seu valor é retornado pelo método getServletInfo() do Servlet (veja seção 3.10).

**Exemplo de diretiva "page" com atributo "info"**

```
<%@ page info="Escrito por nome_do_autor" %>
```

Da mesma forma que o método setContentType () apresentado na seção 4.2 desse livro, o atributo contentType serve para indicar o tipo de conteúdo sendo gerado pela página JSP. Assim, podemos utilizar a seguinte diretiva no início de uma página JSP para indicar que seu conteúdo é uma página HTML.

**Exemplo de diretiva "page" com atributo "contentType"**

```
<%@ page contentType="text/html" %>
```

O atributo seguinte, import deve ser utilizado para indicar pacotes a serem importados no Servlet que será gerado (via declaração import). Assim, devemos indicar por meio desse atributo todos os pacotes que estaremos utilizando na programação de nossa página JSP. Se quisermos utilizar a classe Vector do pacote java.util, e as classes que estão no pacote java.io, por exemplo, poderíamos declarar a diretiva page com os seguintes atributos:

**Exemplo de diretiva "page" com atributo "import"**

```
<%@ page import="java.io.*" %>
<%@ page import="java.util.Vector" %>
```

Finalmente, o atributo errorPage serve para indicar a página JSP a ser exibida em caso de erro no

processamento da página corrente. A página JSP que for exibir o erro deve, por sua vez, declarar a diretiva page com o atributo isErrorPage definido explicitamente como true.

Usando todos esses atributos da diretiva page, poderíamos implementar uma página JSP simples de impressão da data corrente da seguinte forma:

**Página JSP que imprime data corrente (datacorrente.jsp)**

```
<%@ page errorPage="ErroPag.jsp" %>
<%@ page info="Escrito por nome_do_autor" %>
<%@ page contentType="text/html" %>
<%@ page import="java.util.*" %>
<%@ page import="java.text.SimpleDateFormat" %>
<HTML>
<BODY BGCOLOR="#FFFFFF">
A data corrente é <%= new SimpleDateFormat("dd/MM/yyyy").format
(new Date ()) %>
</BODY></HTML>
```

Além do tipo de diretiva page ao qual todos os atributos apresentados anteriormente se referem, existe também um outro tipo de diretiva, a diretiva include. Essa diretiva admite um único atributo file.

Essa diretiva deve ser utilizada para incluir o conteúdo de outro arquivo na página JSP corrente, sendo que esse arquivo tanto pode conter um conteúdo estático, como uma página HTML (ou pedaço de uma página HTML), como um conteúdo dinâmico, ou seja, uma outra página JSP.

Sendo assim, podemos reescrever a página JSP anterior da seguinte forma

**Segunda versão da página JSP que imprime data corrente**

```
<%@ include file="cabecalho.jsp" %>
A data corrente é <%= new SimpleDateFormat("dd/MM/yyyy").format
(new Date ()) %>
<%@ include file="rodape.html" %>
```

, sendo que o conteúdo do arquivo cabecalho.jsp é

**Cabeçalho para página JSP que imprime data corrente (cabecalho.jsp)**

```
<%@ page errorPage="ErroPag.jsp" %>
<%@ page info="Escrito por nome_do_autor" %>
<%@ page contentType="text/html" %>
<%@ page import="java.util.*" %>
<%@ page import="java.text.SimpleDateFormat" %>
<HTML>
<BODY BGCOLOR="#FFFFFF">
```

, e o conteúdo do arquivo rodape.html é

**Rodapé para página JSP que imprime data corrente (rodape.html)**

```
</BODY></HTML>
```

A vantagem de utilizar essa diretiva está no fato de que você pode manter conteúdo estático ou dinâmico comum a diversas páginas JSP em arquivos separados, incluídos, através dessa diretiva, conforme a necessidade. Podemos, por exemplo, construir novas páginas JSP que incluem o arquivo de cabeçalho cabecalho.jsp e rodape.html: se for necessário mudar o conteúdo do cabeçalho ou do rodapé, não precisaremos editar todas as nossas páginas, apenas o conteúdo desses arquivos.

## 7.7 Expressões

Em todas as páginas JSP construídas até agora utilizamos um elemento dinâmico chamado de Expressão: esse elemento serve para imprimir o resultado String de uma expressão Java.

Sua sintaxe básica é a seguinte:

**Sintaxe de uma expressão JSP**

```
<%= <expressão Java> %>
```

Obviamente, esse elemento pode ser utilizado para imprimir o conteúdo de uma variável do tipo String, ou até mesmo de uma constante String. Supondo que l_texto seja uma variável String, por exemplo, poderíamos incluir o seguinte elemento em uma página JSP para imprimir o conteúdo da variável:

**Exemplo de inclusão de uma expressão JSP para a impressão do conteúdo de uma variável do tipo "String"**

```
<%= l_texto %>
```

Por outro lado, podemos formar expressões mais complexas, como na página JSP que imprime a data corrente, desde que o resultado dessas expressões sejam String's.

**Expressão que imprime a data corrente na página JSP "datacorrente.jsp"**

```
<%= new SimpleDateFormat("dd/MM/yyyy").format(new Date ()) %>
```

## 7.8 Scriptlets

Uma expressão JSP possibilita o processamento de uma expressão Java, e a impressão de seu "resultado" junto com o conteúdo da página JSP. Embora esse recurso seja bastante poderoso, ele não serve para situações quando precisamos efetuar um processamento mais complexo, utilizando, por exemplo, diversos blocos de código Java.

Um "Scriptlet" permite a inserção de um bloco de código Java diretamente no corpo da página JSP. Sua sintaxe é a seguinte:

**Sintaxe de um Scriptlet JSP**

```
<% <bloco de código Java> %>
```

Nós podemos utilizar um Scriptlet para incrementar um pouco nossa página JSP datacorrente.jsp:

### Terceira versão da página JSP que imprime data corrente

```
<%@ include file="cabecalho.jsp" %>
<%
    String l_textoSaudacao = "";
    String l_data = new SimpleDateFormat( "dd/MM/yyyy" ).format
( new Date ());
    if( l_data.startsWith( "01/01" )) l_textoSaudacao = "Feliz Ano
Novo!";
    else if( l_data.startsWith( "25/12" )) l_textoSaudacao = "Feliz
Natal!";
%>
A data corrente é <%= l_data %><BR><%= l_textoSaudacao %>
<%@ include file="rodape.html" %>
```

Nesse exemplo de página JSP, utilizamos um Scriptet para definir variáveis cujos conteúdos são posteriormente impressos por expressões JSP. De fato, podemos combinar o uso de Scriptlets e expressões, desde que, obviamente, seja preservada a semântica Java (ou seja, variáveis sejam utilizadas somente após serem declaradas, por exemplo).

Construímos, a seguir, um exemplo um pouco mais elaborado de página JSP utilizando "Scriptlets": trata-se de um formulário HTML para input do mês do aniversário do usuário, sendo que Scriptlets são utilizados para construir os meses possíveis da caixa de seleção do formulário.

### Página JSP com formulário HTML de input do mês de aniversário do usuário ("formmesaniv.jsp")

```
<%@ include file="cabecalho.jsp" %>
<FORM ACTION="procmesaniv.jsp" METHOD="POST">
Entre com o mês de seu aniversário:
<SELECT NAME="MES">
<%
    // Scriptlet: fazendo um "for" para iterar sobre os 12 meses do
ano, ...
    for( int i=1; i<=12; i++ )
    {
        // ... saindo do Scriptlet para imprimir os elementos estáticos
        // da página e as expressões JSP
```

```
%><OPTION VALUE="<%= Integer.toString( i ) %>"><%=
Integer.toString( i ) %></OPTION>
<%
        // ... e voltando para um Scriptlet para fechar o "for"
    }
    // Finalizando o "for"; saindo do scriptlet para imprimir
elementos
    // estáticos da página
%>
</SELECT><BR>
<INPUT TYPE="SUBMIT" NAME="ENVIAR" VALUE="Enviar">
</FORM>
<%@ include file="rodape.html" %>
```

É interessante observar nesse exemplo a combinação de Scriptlets, expressões e elementos estáticos da página: o bloco de código Java iniciado no primeiro Scriptlet da página não é finalizado nesse próprio Scriptlet, mas sim em um outro Scriptlet, inserido após a impressão de um conteúdo estático e de 2 expressões JSP.

## 7.9 Objetos implícitos

No nosso primeiro exemplo de página JSP (veja seção 7.2), fizemos referência a um objeto que não foi declarado em nenhum ponto da página: na expressão JSP dessa página, existe uma referência a um objeto request.

**Primeiro exemplo de página JSP**

```
<HTML><BODY>
O seu endereço IP é "<%= request.getRemoteAddr () %>"
</BODY></HTML>
```

Esse objeto equivale, na verdade, a instância da classe HttpServletRequest passada como parâmetro para o Servlet quando esse recebe uma requisição, conforme estudamos no *Capítulo 5 – Captura de parâmetros da requisição* desse livro.

Além desse objeto, que representa a requisição recebida pela página, a API de páginas JSP disponibiliza outros

objetos implícitos que podem ser utilizados nos elementos dinâmicos programados pelo desenvolvedor.

Segue uma lista dos principais objetos implícitos, com suas respectivas classes:

| Objeto | Classe |
|---|---|
| request | javax.servlet.http.HttpServletRequest |
| response | javax.servlet.http.HttpServletResponse |
| out | javax.servlet.jsp.JspWriter |
| session | javax.servlet.http.HttpSession |
| application | javax.servlet.ServletContext |
| config | javax.servlet.ServletConfig |

Você deve utilizar esses objetos implícitos da mesma maneira como você utilizaria os objetos das respectivas classes na programação de seus Servlets. Em particular, o objeto implícito out, da classe JspWriter, provê funcionalidade semelhante a da classe PrintWriter que utilizamos na maior parte dos exemplos com Servlets desse livro.

No exemplo seguinte, reescrevemos o formulário de input do mês de aniversário do usuário utilizando o objeto implícito out:

**Segunda versão para página JSP de input do mês de aniversário do usuário (“formmesaniv.jsp”)**

```
<%@ include file=“cabecalho.jsp” %>
<FORM ACTION=”procmesaniv.jsp” METHOD=”POST”>
Entre com o mês de seu aniversário:
<SELECT NAME=”MES”>
<%
   // Scriptlet: fazendo um “for” para iterar sobre os 12 meses do
ano
   for( int i=1; i<=12; i++ )
      out.println( “<OPTION VALUE=\”” + Integer.toString( i ) +
“\”>”
         + Integer.toString( i ) + “</OPTION>” );
%>
</SELECT><BR>
<INPUT TYPE=”SUBMIT” NAME=”ENVIAR” VALUE=”Enviar”>
```

```
</FORM>
<%@ include file="rodape.html" %>
```

Podemos também construir a página JSP procmesaniv.jsp que fará o tratamento dos dados submetidos pelo formulário da página formmesaniv.jsp, utilizando os objetos implícitos request e response:

**Página JSP que faz o tratamento dos dados submetidos pelo formulário "formmesaniv.jsp" ("procmesaniv.jsp")**

```
<%
    Integer l_mes = null;
    try {
        l_mes = new Integer(request.getParameter("MES"));
    } catch(Exception p_e) {
        l_mes = null;
    }
    if((l_mes == null) || (l_mes.intValue () < 1) || (l_mes.intValue () >
12))
        response.sendRedirect("formmesaniv.jsp");
%><%@ include file="cabecalho.jsp" %>
<%
    if(l_mes.intValue () == Calendar.getInstance ().get
(Calendar.MONTH) + 1)
        out.println("Parabéns, esse é o mês de seu aniversário!");
    else out.println("Infelizmente, esse não é o mês de seu
aniversário; seu
        aniversário é no mês \"" + l_mes.toString () + "\".");
%>
<%@ include file="rodape.html" %>
```

Nesse exemplo, tomamos o cuidado de tratar o parâmetro recebido e, conforme o caso, gerar o redirecionamento da requisição logo no início da página, antes de escrever qualquer outro conteúdo estático ou dinâmico: da mesma forma que no desenvolvimento de Servlets, precisamos obedecer a ordem de geração dos elementos da resposta HTTP, ou seja, os headers do redirecionamento devem ser gerados antes da escrita de qualquer elemento do conteúdo da resposta (veja seção 4.2).

## 7.10 Declarações

Esse tipo de elemento dinâmico especial de uma página JSP serve para definir códigos Java que deverão ficar fora do método de atendimento das requisições (o método service (), no mais alto nível). Assim, esse elemento serve para declarar variáveis de classe (estáticas), variáveis de instância, ou até mesmo novos métodos.

Sua sintaxe é a seguinte:

**Sintaxe de uma declaração JSP**

```
<%! <declarações da página> %>
```

Em particular, esse elemento deve ser utilizado para definir os métodos jspInit () e jspDestroy () (mencionados na seção 7.4) caso você opte por incluir um código de inicialização ou finalização da página JSP.

A página JSP seguinte aperfeiçoa um pouco mais nossa página input do mês de aniversário do usuário, imprimindo o nome do mês por extenso na caixa de seleção do formulário. O array de "mapeamento" do nome do mês é declarado em um elemento de declaração JSP, para evitar a alocação de memória adicional a cada atendimento de requisição.

### Terceira versão para página JSP de input do mês de aniversário do usuário ("formmesaniv.jsp")

```
<%@ include file="cabecalho.jsp" %>
<%!
    // Array com os nomes dos meses
    private static String [] c_nomesMeses = { "Janeiro", "Fevereiro",
"Março",
            "Abril", "Maio", "Junho", "Julho", "Agosto", "Setembro",
"Outubro",
            "Novembro", "Dezembro" };


    // Função para retornar o nome do mês associado ao número
passado como parâmetro
    private String obtemNomeMes(int p_numMes) {
        if((p_numMes >= 1) && (p_numMes <= 12))
            return c_nomesMeses[p_numMes – 1];
```

```
        else return null;
    }
%>
<FORM ACTION="procmesaniv.jsp" METHOD="POST">
Entre com o mês de seu aniversário:
<SELECT NAME="MES">
<%
    // Scriptlet: fazendo um "for" para iterar sobre os 12 meses do
ano
    for( int i=1; i<=12; i++ )
        out.println("<OPTION VALUE=\"" + Integer.toString( i ) +
"\">"
            + obtemNomeMes( i ) + "</OPTION>");
%>
</SELECT><BR>
<INPUT TYPE="SUBMIT" NAME="ENVIAR" VALUE="Enviar">
</FORM>
<%@ include file="rodape.html" %>
```

## 7.11 Comentários

Existem, na verdade, dois tipos de comentários que você poderá utilizar em sua página JSP.

Um primeiro tipo é o comentário HTML: independente de ser uma página JSP (ou seja, mesmo sendo uma página HTML estática), você pode utilizar esse elemento para incluir um texto que não aparecerá diretamente para o usuário que estiver visualizando sua página. O usuário poderá, por outro lado, ler o comentário caso visualize o fonte da página.

A sintaxe de um comentário HTML é a seguinte:

**Sintaxe de um comentário HTML**

```
<!— <comentário> —>
```

O outro tipo de comentário que você poderá utilizar é um comentário JSP: ao contrário de um comentário HTML, o texto escrito por você não aparecerá para o usuário mesmo que ele visualize o fonte da página. A sintaxe de um comentário JSP é a seguinte:

**Sintaxe de um comentário JSP**

```
<%— <comentário> —%>
```

Exemplificando o uso dos dois tipos de comentários:

**Página JSP com exemplo de uso dos dois tipos de comentários (HTML e JSP)**

```
<HTML><BODY>
Exemplo de uso dos tipos de comentários de uma página JSP
<!—Esse comentário irá aparecer no fonte da página —>
<%— Esse comentário não irá aparecer no fonte da página —%>
</BODY></HTML>
```

## 7.12 JavaBeans

JavaBeans são, na verdade, classes Java reutilizáveis que seguem algumas regras bem definidas para nomeação de seus métodos e variáveis. A idéia por trás do uso desses JavaBeans em nossas páginas JSP, é que eles encapsulem a lógica de nossa aplicação, separando-a do restante da página.

Embora a definição exata de JavaBeans fuja ao escopo desse livro, para efeitos de uso do uso dessas classes em páginas JSP, é necessário que se siga algumas regras básicas no seu desenvolvimento:

1) O construtor da classe, se declarado, não deve receber nenhum argumento.
2) Podem existir um ou mais métodos públicos para a definição de valores de propriedades do Bean; esses métodos são chamados de métodos setter.
3) Podem existir um ou mais métodos públicos para a obtenção de valores de propriedades do Bean; esses métodos são chamados de métodos getter.

Temos, a seguir, uma exemplo de classe JavaBean que implementa uma lógica básica de nossa aplicação:

**Primeiro exemplo de JavaBean: encapsula lógica de cálculo de preços de um produto**

```
package com.minhaempresa;

// JavaBean simples para cálculo de preço de um produto
public class PrecoProdBean {

    // Valor unitário
    private int m_PrecoUnid = 10;

    // Método para cálculo do valor total de um lote: recebe como
parâmetro a quantidade de produtos no lote
    public int calcPrecoLote( int p_quantProds ){
        return p_quantProds * m_PrecoUnid;
    }

}
```

Uma vez construído o JavaBean, para referenciá-lo em nossa página JSP, devemos utilizar o elemento dinâmico <jsp:useBean>, que tem a seguinte sintaxe:

**Sintaxe de um elemento para inclusão de um JavaBean**

```
<jsp:useBean id="<id do Bean>" scope="<escopo do Bean>"
class="<classe do Bean>"/>
```

Na sintaxe anterior, <id do Bean> deve conter o nome do JavaBean como ele será referenciado na página e <classe do Bean> deve conter o nome da classe incluindo informações sobre seu pacote (como por exemplo, com.minhaempresa.PrecoProdBean).

O atributo scope, por outro lado, deve conter um dos seguintes valores: page (página; é o valor default caso esse atributo não seja explicitamente definido), request (requisição), session (sessão) ou application (aplicação); esse valor indica o escopo dentro do qual o JavaBean será visível. Assim, se o escopo de um JavaBean for de sessão, ele será armazenado como um atributo de sessão, podendo ser referenciado em todas as requisições dessa mesma sessão.

Se quiséssemos utilizar o JavaBean PrecoProdBean em uma página JSP, por exemplo, poderíamos incluir as seguintes linhas de código:

**Exemplo de página JSP utilizando o JavaBean "PrecoProdBean"**

```
<HTML>
<BODY>
<jsp:useBean id="PrecoProduto"
class="com.minhaempresa.PrecoProdBean"/>
<%= "O preço total do lote de 10 produtos é: "
    + Integer.toString(PrecoProduto.calcPrecoLote(10)) %>
</BODY>
</HTML>
```

Esse elemento <jsp:useBean> não é o único elemento disponibilizado pela API de páginas JSP para trabalhar com JavaBeans: existem elementos para referenciar diretamente os métodos getter e setter dos JavaBeans desenvolvidos.

Ainda no exemplo de JavaBean anterior (PrecoProdBean), vamos acrescentar métodos getter e setter para a propriedade com o preço de unidade do produto:

**Segundo exemplo de JavaBean: aperfeiçoa PrecoProdBean com métodos "getter" e "setter" para o preço de unidade**

```
package com.minhaempresa;

// JavaBean simples para cálculo de preço de um produto
public class PrecoProdBean {

    // Valor unitário default
    private int m_PrecoUnid = 10;

    // Método "getter" para o preço da unidade do produto
    public int getPrecoUnid() {
        return m_PrecoUnid;
    }
```

```
    // Método "setter" para o preço da unidade do produto
    public void setPrecoUnid(int p_precoUnid) {
        m_PrecoUnid = p_precoUnid;
    }

    // Método para cálculo do valor total de um lote: recebe como
    // parâmetro a quantidade de produtos no lote
    public int calcPrecoLote(int p_quantProds) {
        return p_quantProds * m_PrecoUnid;
    }

}
```

Para referenciar o método getter de um JavaBean, a API de páginas JSP disponibiliza o elemento <jsp:getProperty>, cuja sintaxe é:

**Sintaxe de um elemento para referenciar o método "getter" de um JavaBean**

```
<jsp:getProperty name="<nome do JavaBean>" property="<nome
da propriedade>"/>
```

Para imprimir o valor do preço unitário de um produto, por exemplo, poderíamos acrescentar as seguintes linhas em nossa página JSP:

**Segundo exemplo de página JSP utilizando o JavaBean "PrecoProdBean"**

```
<HTML>
<BODY>
<jsp:useBean id="PrecoProduto"
class="com.minhaempresa.PrecoProdBean"/>
O preço unitário do produto é <jsp:getProperty
name="PrecoProduto"
   property="precoUnid"/>
</BODY>
</HTML>
```

Da mesma forma como existe um elemento dinâmico para referenciar métodos getter de JavaBeans, também existe um elemento dinâmico para os métodos setter. A sintaxe desse elemento é:

**Sintaxe de um elemento para referenciar o método "setter" de um JavaBean**

```
<jsp:setProperty name="<nome do JavaBean>" property="<nome
da propriedade>"
   value="<valor da propriedade>" />
```

Podemos, então, modificar o exemplo de página JSP anterior para definir o valor do preço unitário do produto antes de exibi- lo:

**Terceiro exemplo de página JSP utilizando o JavaBean "PrecoProdBean"**

```
<HTML>
<BODY>
<jsp:useBean id="PrecoProduto"
class="com.minhaempresa.PrecoProdBean"/>
<jsp:setProperty name="PrecoProduto" property="precoUnid"
value="3" />
O preço unitário do produto é <jsp:getProperty
name="PrecoProduto"
   property="precoUnid"/>
</BODY>
</HTML>
```

Esse elemento de referência a métodos setter de JavaBeans admite, ainda, outra sintaxe de uso:

**Sintaxe alternativa de elemento para referenciar o método "setter" de um JavaBean**

```
<jsp:setProperty name="<nome do JavaBean>" property="<nome
da propriedade>" param="<nome do parâmetro>" />
```

Nessa sintaxe, o valor que será recebido pelo JavaBean para a propriedade em questão, será o valor do parâmetro <nome do parâmetro> , conforme recebido pela requisição HTTP.

# 7.13 Bibliotecas de Tags (Tag Libraries)

Existe um último tipo de elemento dinâmico que pode ser utilizado em uma página JSP: um elemento de referência a uma biblioteca de Tags.

Uma biblioteca de Tags permite que você também separe a lógica de programação de sua aplicação do conteúdo da página JSP, assim como JavaBeans se propõe a fazer.

No entanto, ao contrário de JavaBeans, uma biblioteca de Tags oferece um acesso nativo aos objetos da página JSP, como por exemplo, o objeto que encapsula a resposta do Servlet correspondente a página JSP.

Nessa seção estaremos explorando algumas características básicas de biblioteca de Tags. Uma explicação mais detalhada e aprofundada sobre TagLibs pode ser obtida em http://java.sun.com/products/jsp/tutorial/TagLibrariesTOC.html.

Existem 4 componentes principais por trás do uso de uma biblioteca de Tags.

O primeiro componente é um arquivo chamado de "Descritor de Bibliotecas de Tags" (Tag Library Descriptor): esse arquivo, nomeado com extensão ".tld" e colocado no diretório WEB-INF, contém as configurações das bibliotecas de Tags utilizadas pela aplicação.

### Exemplo de um Descritor de Bibliotecas de Tags ("minhabibliotags.tld")

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
    <tlibversion>1.0</tlibversion>
    <shortname>exemplotag</shortname>
    <info>Exemplo de biblioteca de Tags</info>
    <tag>
        <name>TagOlaMundo</name>
        <tagclass>com.minhaempresa.TagOlaMundo</tagclass>
        <bodycontent>empty</bodycontent>
```

```
    </tag>
</taglib>
```

Nesse exemplo de “Descritor de Biblioteca de Tags”, podemos observar diversos elementos: os elementos tlibversion, shortname e info contém, respectivamente, a versão da biblioteca de Tags, seu nome (conforme referenciado posteriormente na página JSP) e um texto informativo sobre a biblioteca.

O último elemento no descritor de nosso exemplo é um elemento “tag”: esse elemento serve para especificar um nome de uma Tag (TagOlaMundo) e a respectiva classe que irá tratar essa Tag quando utilizada na página JSP (com.minhaempresa.TagOlaMundo). O elemento bodycontent especifica o tipo de tratamento que o conteúdo da Tag deverá receber: no nosso exemplo, esse conteúdo deverá ser vazio, ou seja, não deverá haver texto nenhum entre o elemento de início e fim da Tag; outros valores possíveis para esse elemento são JSP e tagdependent.

Embora em nosso exemplo haja somente um elemento do tipo “tag”, pode haver mais de um elemento desse tipo no mesmo “Descritor de Biblioteca de Tags”.

O componente seguinte por trás do uso de uma biblioteca de Tags, é o “Deployment Descriptor”, já apresentado no *Capítulo 2 – Instalação e Configuração* desse livro. Para que se possa utilizar uma biblioteca de Tags, é necessário incluir um mapeamento de uma URI, referenciada na página JSP, ao arquivo “.tld” com a biblioteca de Tags correspondente.

Assim, poderíamos incluir as seguintes linhas no “Deployment Descriptor” de nossa aplicação:

**Exemplo de linhas incluídas no Deployment Descriptor para utilização de uma biblioteca de Tags**

```
<!DOCTYPE web- app
    PUBLIC “-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN”
```

```
        "http://java.sun.com/dtd/web- app_2_3.dtd">

<web- app>
...

<taglib>
    <taglib- uri>/minhabibliotags</taglib- uri>
    <taglib- location>/WEB- INF/minhabibliotags.tld</taglib-
location>
</taglib>

...
</web- app>
```

O componente seguinte de uma biblioteca de Tags é a classe que irá gerenciar a Tag (Tag Handler). Essa classe, especificada pelo elemento tagclass do Descritor da Biblioteca de Tags, é responsável por executar as rotinas pertinentes quando a Tag é encontrada na página JSP.

Essa classe que gerencia a Tag deve implementar a interface Tag ou BodyTag: a diferença básica entre essas duas interfaces diz respeito ao tratamento do conteúdo da Tag. Como em nosso exemplo não estamos interessandos no conteúdo da Tag, a classe que apresentamos implementa a interface Tag.

### Exemplo de classe para gerenciar uma Tag ("Tag Handler")

```
package com.minhaempresa;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

// Classe que gerencia Tag TagOlaMundo, utilizada em páginas JSP
de nossa aplicação
public class TagOlaMundo implements Tag {
    PageContext m_PageContext;

    // Implementação default
    public void setParent(Tag p_tag) {
```

```
    }

    // Implementação default
    public void setPageContext(PageContext p_pageContext) {
        m_PageContext = p_pageContext;
    }

    // Implementação default
    public void release () {
    }

    // Implementação default
    public Tag getParent () {
        return null;
    }

    // Imprimimos o texto "Ola Mundo!" quando a Tag é encontrada;
    // a função retorna SKIP_BODY de maneira que o conteúdo da
Tag
    // não seja processado (no nosso caso, a Tag é vazia).
    public int doStartTag (){
        try {
            m_PageContext.getOut ().println("Ola Mundo!");
        } catch (Exception p_e) {}
        return SKIP_BODY;
    }

    // Implementação default; a função retorna "EVAL_PAGE" de
maneira que o
    // restante da página JSP seja processado
    public int doEndTag () {
        return EVAL_PAGE;
    }

}
```

Os métodos mais importantes dessa classe são os métodos doStartTag (), chamado quando a Tag é encontrada, doEndTag (), chamado após o processamento do conteúdo da Tag, e release (), chamado ao término de todo o processamento devendo liberar quaisquer recursos

alocados durante o processo. É interessante observar que no método doStartTag ()" obtemos uma referência ao stream de saída da página, e utilizamos esse stream para imprimir nosso texto Ola Mundo!.

Finalmente, o componente final de nossa biblioteca de Tags é a página JSP em si. Para utilizar a biblioteca de Tags minhabibliotags.tld, podemos elaborar a seguinte página JSP de exemplo:

**Exemplo de página JSP que utiliza a biblioteca de Tags "minhabibliotags.tld" ("exemplotags.jsp")**

```
<HTML>
<BODY>
Processando TagOlaMundo ...<BR>
<%@ taglib uri="/minhabibliotags" prefix="minhaLib"%>
<minhaLib:TagOlaMundo/>
</BODY>
</HTML>
```

Nessa página, primeiro incluímos um elemento taglib, cujo atributo uri contém a URI especificada no "Deployment Descriptor" (que por sua vez faz o mapeamento com o arquivo ".tld" correto), e cujo atributo "prefix" especifica o prefixo a ser utilizado antes de cada Tag.

Na linha seguinte, temos a utilização da Tag em si: há um prefixo, conforme especificado pelo atributo prefix do elemento taglib, seguido da Tag, que deverá ter sido declarada no descritor de bibliotecas de Tags. Ao encontrar esse elemento, o container irá carregar a classe que gerencia esse Tag, e chamar os métodos pertinentes.

É importante observar que não existe nenhum conteúdo para a tag minhaLib:TagOlaMundo; a notação abreviada <minhaLib:TagOlaMundo/> é equivalente se escrever <minhaLib:TagOlaMundo></minhaLib:TagOlaMundo>.
Caso houvesse algum conteúdo para essa tag, teríamos que utilizar um valor diferente para o atributo bodyContent no descritor de bibliotecas de Tags, e poderíamos considerar a implementação da interface BodyTag em vez de Tag para a classe TagOlaMundo.

A página HTML resultante do processamento da página JSP apresentada anteriormente será então:

**Resultado do processamento da página "exemplotags.jsp"**

```
<HTML>
<BODY>
Processando TagOlaMundo ...<BR>
Ola Mundo!
</BODY>
</HTML>
```

Embora você possa construir suas próprias bibliotecas de Tags, normalmente é mais prático e fácil utilizar uma biblioteca já pronta, como o Apache Jakarta Struts, disponível no site http://jakarta.apache.org/struts/ .

# Capítulo 8

# Modelo MVC

Nesse capítulo apresentamos o modelo MVC: através desse modelo, procuramos mostrar como podemos separar o trabalho de desenvolvimento do trabalho de formatação e layout da aplicação.

## 8.1 Programação e layout

Normalmente, o desenvolvimento de uma aplicação Web envolve o trabalho de duas equipes distintas: os desenvolvedores são responsáveis pela programação, e os web-designers são responsáveis pela formatação e layout do front-end da aplicação.

Pelo que vimos até agora, existe uma intersecção entre esses dois mundos: na verdade, qualquer que seja a tecnologia que se utilize para desenvolver aplicações Web, sempre existe um ponto em que se mistura os trabalhos dessas duas equipes.

Agora imagine uma situação onde o início do trabalho de uma equipe dependa da outra equipe finalizar a sua parte; imagine também que, qualquer alteração feita por uma equipe precise necessariamente envolver o trabalho da outra equipe. Essas situações podem onerar tanto o cronograma quanto o custo de um projeto.

Assim, tanto para efeitos de construção, quanto da manutenção da aplicação desenvolvida, é muito importante que haja a maior separação possível entre

esses dois trabalhos, de maneira que as duas equipes possam trabalhar de forma independente, sem dependerem uma da outra.

Embora já tenhamos visto nesse livro algumas técnicas que auxiliam na separação entre lógica da aplicação e apresentação, apresentaremos nesse capítulo uma técnica muito mais eficaz e que pode ser utilizada de maneira complementar as apresentadas anteriormente.

## 8.2 Arquitetura básica

A arquitetura básica do modelo MVC, ou Model-View-Controller, se vale do uso de Servlets, JavaBeans e páginas JSP: os Servlets controlam as requisições recebidas (Controller), os JavaBeans implementam a lógica da aplicação (Model), e as páginas JSP se encarregam da apresentação do resultado (View).

Podemos representar melhor essa arquitetura através da seguinte figura:

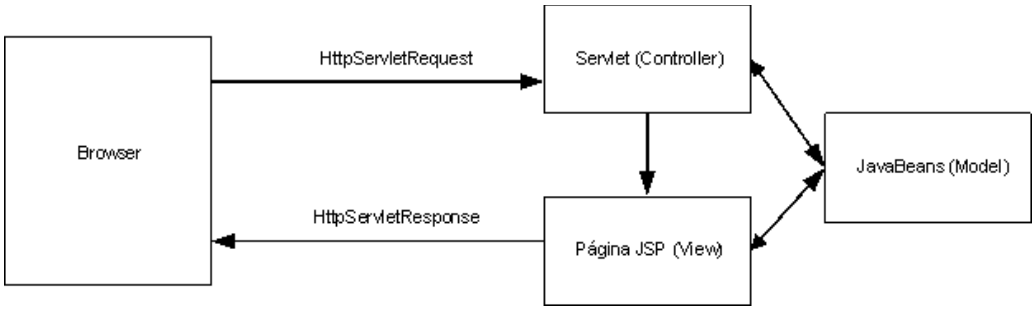


***Figura 8.1 – Arquitetura básica MVC.***

Toda vez que uma requisição é recebida, o Servlet de controle repassa a requisição para a página JSP responsável pela apresentação da resposta, sendo que JavaBeans são utilizados pela página JSP para obter os dados dinâmicos da aplicação.

## 8.3 Forward de requisições

Para que um Servlet Controller possa repassar a requisição recebida para uma página JSP, é necessário utilizar um método específico da classe javax.servlet.http.HttpServletRequest.

**Assinatura do método "getRequestDispatcher ()" da classe HttpServletRequest**

```
public javax.servlet.RequestDispatcher getRequestDispatcher
(java.lang.String p_path);
```

Esses método retorna uma referência para um objeto que implementa a interface javax.servlet.RequestDispatcher e que atua como um "wrapper" para o recurso indicado no path passado como parâmetro para a função. Assim, se você passar como parâmetro, por exemplo, o caminho relativo de uma página JSP, esse método retornará uma referência a um "wrapper" dessa página JSP.

Esse "wrapper", por sua vez, disponibiliza um método forward () que permite que você repasse a requisição para o recurso "encapsulado" pelo "wrapper".

**Assinatura do método "forward ()" da interface RequestDispatcher**

```
public void forward(HttpServletRequest p_request,
HttpServletResponse, p_response);
```

Podemos, dessa forma, implementar um Servlet de exemplo que não faz nada, apenas repassa todas as requisições recebidas para uma página JSP.

**Exemplo de Servlet que repassa requisições para uma página JSP**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


// Servlet simples que repassa requisições recebidas para a
// página JSP "OlaMundo.jsp"
public class ServletRepassaReqs extends HttpServlet {
```

```
    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        // Repassando a requisição para a página JSP OlaMundo.jsp
        try {
            p_request.getRequestDispatcher("/OlaMundo.jsp").forward
(p_request,
                p_response);
        }catch(ServletException p_e) {}
    }

}
```

No Servlet anterior, existe um ponto importante que deve ser observado: o path passado como parâmetro para a função getRequestDispatcher () referencia um recurso contido na mesma aplicação Web do Servlet, e, dessa forma, deve excluir a parte referente ao diretório virtual da aplicação (ou seja, esse path não deve ser escrito como / livroservlets/OlaMundo.jsp). Além disso, é importante observar que o método forward poderá lançar uma exceção: uma das causas pode ser uma exceção lançada pelo próprio recurso referenciado pelo RequestDispatcher.

Podemos também implementar a página JSP referenciada no Servlet anterior como:

### Exemplo de página JSP "OlaMundo.jsp"

```
<HTML>
<BODY>
<% out.println("Ola Mundo!"); %>
</BODY>
</HTML>
```

Assim, a cada requisição recebida, o Servlet ServletRepassaReqs repassa a requisição para a página JSP OlaMundo.jsp, que por sua vez retorna para o Browser a seguinte página HTML

### Resposta do Servlet "ServletRepassaReqs"

```
<HTML>
```

```
<BODY>
Ola Mundo!
</BODY>
</HTML>
```

## 8.4 Atributos de requisições

Existe ainda um outro recurso da API de Servlets que iremos utilizar no desenvolvimento de nossas aplicações Web no modelo MVC: esse recurso é a definição / obtenção de atributos da requisição.

A classe HttpServletRequest possui quatro métodos que podem ser utilizados para gerenciar os atributos de uma requisição.

**Assinatura dos métodos da classe "HttpServletRequest" que gerenciam os atributos de uma requisição**

```
public java.lang.Object getAttribute(java.lang.String p_attributeName);
public java.util.Enumeration getAttributeNames ();
public void removeAttribute(java.lang.String p_attributeName);
public void setAttribute(java.lang.String p_attributeName,
java.lang.Object
        p_attributeValue);
```

Esses métodos funcionam de maneira semelhante aos métodos da classe ServletContext apresentada na seção 3.6: eles permitem definir, remover ou obter valores de atributos de uma requisição. Esses valores de atributos não precisam necessariamente ser objetos String, eles podem ser objetos quaisquer Java.

Para exemplificar o uso dessas funções, vamos implementar novamente nosso Servlet ServletRepassaReqs da seguinte forma:

**Segunda versão para Servlet que repassa requisições para uma página JSP**

```
import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;

// Servlet simples que repassa requisições recebidas para a página
JSP
public class ServletRepassaReqs extends HttpServlet {

    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        // Definimos um atributo da requisição chamado "Mensagem"
        p_request.setAttribute("Mensagem", "Ola Mundo!");

        // Repassando a requisição para a página JSP
ApresentaMensagem.jsp
        try {
            p_request.getRequestDispatcher("/livroservlets/
                ApresentaMensagem.jsp").forward(p_request,
p_response);
        }catch(ServletException p_e) {}
    }

}
```

A página JSP ApresentaMensagem.jsp, por sua vez, pode ser implementada como:

### Página JSP "ApresentaMensagem.jsp"

```
<HTML>
<BODY>
<%
    String l_mensagem = (String) request.getAttribute("Mensagem");
    if(l_mensagem != null) out.println(l_mensagem);
%>
</BODY>
</HTML>
```

Dessa forma, além de repassar a requisição do Servlet para a página JSP, estamos também passando objetos, definidos no Servlet, para a página JSP. Esse mecanismo funciona graças aos atributos de requisição.

## 8.5 Juntando as partes

Juntando os recursos apresentados nas seções anteriores, podemos finalmente apresentar o modelo MVC com todos os seus componentes.

Para fazer essa apresentação, reimplementaremos e estenderemos a aplicação de login da seção 6.4 desse livro. Agora, a aplicação de login passará a prever também um nível de acesso associado a cada usuário que efetua a autenticação: se o usuário tiver nível de acesso administrativo, será apresentada a interface do administrador, caso contrário será apresentada a interface do usuário "comum".

Assim, o Servlet de Login será:

### Servlet "Login"

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Apresenta o formulário de login e faz a autenticação da aplicação
public class Login extends HttpServlet {

    // Verifica se existe usuário com o login e senha passados como
parâmetro: se
    // existir, retorna nível de acesso do usuário em questão ("1" para
nível de
    // usuário comum, "2" para nível de administrador); caso
contrário, retorna "-1".
    private int NivelAcessoUsuario(String p_login, String p_senha) {
        int l_nivelAcesso = -1;
        if((p_login != null) && (p_senha != null)) {
            // Você deve implementar aqui o código para validar o
login
            // e senha do usuário e definir a variável "l_nivelAcesso"
            // (por exemplo, consultando uma base de dados)
        }
        return l_nivelAcesso;
    }
```

```
    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
          throws IOException {

       // Definindo headers auxiliares para evitar cacheamento de
página
       p_response.setHeader("Cache- Control", "no- cache, must-
revalidate");
       p_response.setHeader("Pragma", "no- cache");
       p_response.setHeader("Expires", "Mon, 26 Jul 1997 05:00:00
GMT");
       p_response.setDateHeader("Last- Modified",
System.currentTimeMillis ());

       // Fazendo autenticação do usuário e obtendo seu nível de
acesso
       String l_login = p_request.getParameter("LOGIN"), l_senha =
          p_request.getParameter("SENHA");
       int l_nivelAcesso = NivelAcessoUsuario(l_login, l_senha);

       // Definindo atributos de sessão (se o usuário for válido)
       HttpSession l_sessao = p_request.getSession(true);
       if(l_nivelAcesso != - 1) l_sessao.setAttribute("LOGIN", l_login);
       else l_sessao.removeAttribute("LOGIN");

       try {
          if(l_nivelAcesso == - 1) {
             // Usuário não conseguiu se autenticar; existem duas
possibilidades:
             // login / senha incorretos ou usuário ainda não enviou
os dados do
             // formulário (simplesmente acessou a página); em
ambos os casos,
             // fazemos um "forward" para a página JSP
"FormLogin.jsp"
             p_request.setAttribute("Mensagem", "");
             if((l_login != null) || (l_senha != null))
p_request.setAttribute
                ("Mensagem", "Erro: login e/ou senha inválido(s)!" );
             p_request.getRequestDispatcher("/FormLogin.jsp").
forward(p_request,
                p_response);
          }
```

```
            else {
                // Usuário conseguiu efetuar autenticação;
apresentando a interface
                // correspondente ao seu nível de acesso
                p_request.setAttribute("Login", l_login);
                if(l_nivelAcesso == 1) p_request.getRequestDispatcher
                    ("/UsuarioComum.jsp").forward(p_request,
p_response);
                else p_request.getRequestDispatcher
                    ("/Administrador.jsp").forward(p_request,
p_response);
            }
        }catch( ServletException p_e ) {}
    }

    // Nesse Servlet, os métodos GET e POST são tratados da mesma
maneira
    // (pelo mesmo trecho de código)
    public void doPost(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {
        doGet(p_request, p_response);
    }
}
```

Conforme você pode observar no Servlet anterior, não existe nenhuma codificação feita que se relaciona a apresentação da interface para o usuário da aplicação. O Servlet simplesmente trata os parâmetros recebidos, repassando a requisição para as diversas páginas JSP da aplicação, que deverão ser responsáveis pela interface em si.

### Página JSP "FormLogin.jsp"

```
<HTML>
<BODY>
<%= (String) request.getAttribute("Mensagem") %><BR>
<FORM ACTION="Login" METHOD="POST">
Login: <INPUT TYPE="TEXT" NAME="LOGIN"><BR>
Senha: <INPUT TYPE="PASSWORD" NAME="SENHA"><BR>
<INPUT TYPE="SUBMIT" NAME="ENTRAR" VALUE="Entrar"><BR>
</FORM>
```

```
</BODY></HTML>
```

### Página JSP “UsuarioComum.jsp”

```
<HTML>
<BODY>
Olá, <%= request.getAttribute(“Login”) %>, você tem acesso como usuário comum!
</BODY>
</HTML>
```

### Página JSP “Administrador.jsp”

```
<HTML>
<BODY>
Olá, <%= request.getAttribute(“Login”) %>, você tem acesso como administrador!
</BODY>
</HTML>
```

Essas páginas JSP poderão ser trabalhadas pelos responsáveis pelo layout e formatação de nossa aplicação, sem afetar a lógica e o trabalho de desenvolvimento.

# Capítulo 9

# Tópicos adicionais

Já apresentamos nos capítulos anteriores as principais características e funcionalidades referentes ao desenvolvimento de aplicações Web com Servlets e páginas JSP.

Estaremos, nesse capítulo, apresentando alguns tópicos adicionais que complementarão todo o conhecimento que você obteve até agora: você irá conhecer arquivos WAR, mecanismos de autenticação HTTP e pools de conexões a uma base de dados.

## 9.1 Arquivos WAR

Já apresentamos, na seção 2.2, uma maneira através da qual você pode instalar uma aplicação Web em um servidor de aplicações: você pode fazer essa instalação criando, abaixo do diretório webapps, uma pasta com o nome de sua aplicação, com o conteúdo dessa pasta obedecendo a um formato específico (veja seção 2.2).

Embora essa forma de instalar a aplicação funcione, é considerado mais elegante distribuir sua aplicação no formato de um arquivo WAR, ou **W**eb **A**pplication A**r**chive.

Um arquivo WAR nada mais é que um arquivo ".jar", nomeado com a extensão ".war". Assim, você deve gerar o arquivo WAR utilizando o aplicativo jar para juntar todo o conteúdo do diretório de sua aplicação (retirando, obviamente, os arquivos fontes ".java").

Assim, poderíamos, por exemplo, remover os arquivos ".java" de dentro do diretório livroservlets (que contém a nossa aplicação), e, a partir de um PROMPT DOS e de dentro desse diretório, digitar a linha de comando jar cvf .. \livroservlets.war *.

Com um arquivo ".war", podemos instalar nossa aplicação Web em qualquer servidor de aplicações: basta copiar esse arquivo para a pasta webapps do servidor. Obviamente, devemos tomar cuidado para não instalar o arquivo ".war" mais o diretório com toda a nossa aplicação juntos em um mesmo servidor de aplicações.

## 9.2 Autenticação HTTP

Na seção 6.4 desse livro, apresentamos um exemplo de aplicação contendo uma autenticação baseada em formulários HTML. Essa não é a única forma de fazer a autenticação de usuários de uma aplicação.

O protocolo HTTP incorpora uma funcionalidade que pode auxiliar na implementação de um mecanismo de autenticação. Estaremos, nessa seção, mostrando o funcionamento de um tipo especial de autenticação, a partir do protocolo HTTP, chamada de "Basic Authentication" (autenticação básica).

Para implementar esse tipo de autenticação, precisamos utilizar o código de status de resposta HTTP 401 (HttpServletResponse.SC_UNAUTHORIZED) e o header de resposta WWW-Authenticate.

Ao retornar uma resposta com esse código de status e esse header de resposta contendo um valor BASIC REALM=<domínio> , onde <domínio> deve ser substituído por um nome do domínio no qual estamos fazendo a autenticação (cada domínio deve proteger um conjunto de recursos de sua aplicação), faremos com que o browser do usuário de nossa aplicação mostre uma caixa de diálogo pedindo um usuário e senha de acesso.

O usuário e senha digitados nessa caixa pelo usuário serão enviados, por outro lado, junto com uma nova requisição a nossa aplicação, através do header Authorization. O valor desse header estará codificado no formato Base64, sendo necessário utilizar um decodificador para ler, efetivamente, seu conteúdo: esse conteúdo será um texto no formato <usuário digitado>:<senha digitada>.

O Servlet a seguir exemplifica o uso desse tipo de autenticação HTTP:

### Exemplo de Servlet que utiliza autenticação HTTP para controlar acesso

```
import java.io.*;
import javax.servlet.http.*;

// Servlet que utiliza autenticação HTTP para controlar acesso de usuários da aplicação
public class ServletAutenticacaoHTTP extends HttpServlet {

    public void doGet(HttpServletRequest p_request, HttpServletResponse p_response)
            throws IOException {

        // Definindo headers auxiliares para evitar cacheamento de página
        p_response.setHeader("Cache-Control", "no-cache, must-revalidate");
        p_response.setHeader("Pragma", "no-cache");
        p_response.setHeader("Expires", "Mon, 26 Jul 1997 05:00:00 GMT");
        p_response.setDateHeader("Last-Modified", System.currentTimeMillis());

        // Obtendo o valor do header "Authorization"; se esse valor for nulo,
        // definimos o código de resposta HTTP SC_UNAUTHORIZED e o header
        // WWW-Authenticated" para que o browser do usuário apresente a caixa de
        // diálogo solicitando o preenchimento de login e senha
        String l_headerAuth = p_request.getHeader("Authorization");
        if(l_headerAuth == null)
```

```
        {
            p_response.setHeader("WWW-Authenticate",
                "BASIC REALM=\"Dominio de Autenticacao\"");
            p_response.sendError(p_response.SC_UNAUTHORIZED);
            return;
        }

        // Decodificando o conteúdo do header: para isso utilizamos a
classe
        // "Base64.Java" disponível em
http://iharder.sourceforge.net/base64/
        // OBS: é necessário desconsiderar os 6 primeiros caracteres
do header
        // (eles contém o String "Basic ")
        String l_headerAuthDec = new String(Base64.decode
(l_headerAuth.substring(6 )));

        // O login é separado da senha pelo caractere ':'
        String l_login = l_headerAuthDec.substring(0,
l_headerAuthDec.indexOf(':'));
        String l_senha = l_headerAuthDec.substring
(l_headerAuthDec.indexOf(':') + 1);

        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");
        l_pw.println("Login: \"" + l_login + "\"<BR>");
        l_pw.println("Senha: \"" + l_senha + "\"<BR>");
        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }

}
```

Existe também uma maneira de implementar a autenticação de usuários que estão acessando nossa aplicação baseada em elementos de segurança configurados no "Deployment Descriptor" de nossa aplicação (veja seção 2.2).

Para implementar essa autenticação dessa forma, você deve adicionar elementos security-constraint e um elemento login-config ao "Deployment Descriptor" de sua aplicação.

**Exemplo de configuração de segurança no "Deployment Descriptor" da aplicação**

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Area Restrita</web-resource-name>
        <url-pattern>/ServletAutenticacaoHTTP</url-pattern>
    </web-resource-collection>
    <auth-constraint><role-name>tomcat</role-name></auth-
constraint>
</security-constraint>
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Dominio de Autenticacao</realm-name>
</login-config>
```

Os elementos apresentados na listagem anterior, por exemplo, definem que os acessos a URL / livroservlets/ServletAutenticacaoHTTP devem ser autenticados. Em particular, o elemento role-name define o grupo de usuários que deve ter acesso a URL em questão: os diversos grupos, com seus respectivos usuários / senhas, devem, nesse caso, ser configurados no arquivo conf\tomcat-users.xml abaixo do diretório de instalação do Apache Tomcat.

Se você optar por utilizar esse método de autenticação, você deve chamar em seu código os métodos getRemoteUser () e getUserPrincipal (), da classe HttpServletRequest, para obter o nome do usuário e seu grupo.

Mais detalhes com relação a autenticação baseada em elementos de segurança do "Deployment Descriptor" da aplicação podem ser encontrados na especificação de Servlets.

## 9.3 Pools de conexões a base de dados

Uma das grandes vantagens no desenvolvimento de aplicações Web com Servlets e páginas JSPé a performance obtida em função da persistência dos objetos carregados em memória. Em particular, conforme mostramos em diversos exemplos ao longo desse livro, o estado de uma variável estática pode ser mantido ao longo das diversas requisições recebidas pela aplicação.

Um dos recursos que mais se beneficiam dessa persistência em memória são as conexões com o banco de dados: como o processo de abrir uma nova conexão com o banco de dados pode pesar significativamente na performance da aplicação, vale a pena manter as conexões abertas, em vez de abrir uma nova conexão a cada requisição recebida.

Esse tipo de persistência é, normalmente, implementado por meio de um gerenciador de conexões a base de dados: a cada requisição recebida, um Servlet chama um método desse gerenciador para alocar uma conexão; ao término de seu processamento, o Servlet chama outro método para liberar a conexão previamente alocada. O gerenciador, por sua vez, é responsável por estabelecer efetivamente novas conexões quando necessário, e armazenar essas conexões abertas.

A classe PoolConBD, a seguir, implementa um gerenciador de conexões a base de dados:

### Classe "PoolConBD": implementa gerenciador de pool de conexões com a base de dados

```
package com.livroservlets;

import java.io.*;
import java.sql.*;
import java.util.*;
import java.util.Date;

// Gerenciador de conexões ao banco de dados
// Mantém um pool de conexões ativas com o banco de dados de maneira que não
```

```
// seja necessário se estabelecer uma nova conexão a cada
requisição recebida
public class PoolConBD
{

    // Pool de conexões de uma aplicação específica
    // Assim, cada aplicação Web pode manter um conjunto distinto
de conexões
    // (caso contrário, todas as aplicações que forem executadas em
uma mesma
    // instância da máquina virtual terão que compartilhar o mesmo
Pool).
    private static class PoolConBDAplic
    {

        private String m_Aplic = null;  // Nome da aplicação
        private Vector m_Pool  = null;  // Pool de conexões dessa
aplicação

        // Url, usuario e senha para abrir novas conexões
        private String m_Url = null, m_Usuario = null, m_Senha = null;

        // Tamanho máximo desse pool: dessa forma evitamos que,
caso haja
        // um erro na aplicação, novas conexões sejam abertas
indefinidamente
        // até derrubar o servidor de banco de dados.
        private int m_TamMaxPool = 0;

        // Timeout para o estabelecimento de novas conexões
        private int m_Timeout = 0;

        private Driver m_DriverJdbc = null;  // Driver jdbc
        private int m_NumConsAlocadas = 0;   // Número corrente de
conexões alocadas

        // Inicializa estruturas locais: cadastra driver jdbc e
        // abre a primeira conexão com a base de dados
        public PoolConBDAplic(String p_aplic, String p_driverJdbc,
String p_nomeBD,
                String p_url, String p_usuario, String p_senha, int
p_tamMaxPool, int
                p_timeout) throws PoolConBDException
```

```
        {
            if(p_aplic != null) {
                if((p_driverJdbc != null) && (p_nomeBD != null) &&
(p_url != null) &&
                    (p_usuario != null) && (p_senha != null) &&
(p_tamMaxPool > 0)
                    && (p_timeout > 0)) {
                m_Aplic = p_aplic;
                m_Pool = new Vector ();

                m_Url = p_url;
                m_Usuario = p_usuario;
                m_Senha = p_senha;
                m_TamMaxPool = p_tamMaxPool;
                m_Timeout = p_timeout;

                // Carregando / cadastrando o driver Jdbc
                try {
                    m_DriverJdbc = (Driver) Class.forName(p_driverJdbc).
newInstance ();
                    DriverManager.registerDriver(m_DriverJdbc);
                } catch(Exception p_e) {
                    throw new PoolConBDException(p_aplic, "não foi
possível
                        carregar/cadastrar driver jdbc");
                }

                // Inicializando pool com uma conexão
                try {
                    m_Pool.addElement(alocaNovaConexao ());
                    m_NumConsAlocadas = 1;
                } catch(Exception p_e) {
                    destroiPool ();
                    throw new PoolConBDException(p_aplic, "não foi
possível
                        criar a primeira conexão; " + p_e.getMessage
());
                }
            }
            else throw new PoolConBDException(p_aplic,
                "parâmetros de criação do pool inválidos");
        }
```

```
    }

    // Aloca uma nova conexao com a base de dados
    // OBS: esse método precisa ser "synchronized" para evitar
    // problemas de concorrência no acesso ao Pool
    private synchronized Connection alocaNovaConexao () throws
PoolConBDException {
        Connection l_con = null;
        try {
            l_con = DriverManager.getConnection(m_Url,
m_Usuario, m_Senha);
        } catch (Exception p_e) {
            throw new PoolConBDException(m_Aplic, "não foi
possível abrir nova
                conexão; " + " a razão foi \"" + p_e.getMessage () +
"\"");
        }
        return l_con;
    }

    // Retorna, se possível, uma conexão com a base de dados
    // (ou diretamente do pool ou uma nova conexão)
    // OBS: esse método precisa ser "synchronized" para evitar
    // problemas de concorrência no acesso ao Pool
    public synchronized Connection alocaConexao () throws
PoolConBDException
    {
        Connection l_con = null;
        if(m_Pool != null)
        {
            if(m_Pool.size () > 0)
            {
                // Se o pool de conexões não estiver vazio, podemos
                // retirar dele uma conexão previamente
estabelecida
                l_con = (Connection) m_Pool.firstElement ();
                m_Pool.removeElementAt(0);

                try {
                    if (l_con.isClosed ()) l_con = null;
                } catch (Exception p_e) { l_con = null; }
```

```
                if(l_con == null) l_con = alocaConexao ();
            }
            else if (m_NumConsAlocadas < m_TamMaxPool)
            {
                // ... caso contrário, se ainda não tivermos atingido o
tamanho
                // máximo do pool, podemos estabelecer uma nova
conexão
                l_con = alocaNovaConexao ();
            }
            else throw new PoolConBDException(m_Aplic, «número
máximo
                    de conexões atingido»);
        }

        if(l_con != null) m_NumConsAlocadas ++;
        return l_con;
    }

    // Libera uma conexão previamente alocada (retorna essa
conexão para o pool)
    // OBS: esse método precisa ser “synchronized” para evitar
problemas de
    // concorrência no acesso ao Pool
    public synchronized void liberaConexao(Connection p_con) {
            if(m_Pool != null)
        {
            try {
                if(!p_con.isClosed ()) m_Pool.addElement(p_con);
            } catch (Exception p_e) {}
            m_NumConsAlocadas --;
        }
    }

    // Retorna o número corrente de conexões alocadas
    public synchronized int obtemNumConsAlocadas ()
    {
        return m_NumConsAlocadas;
    }
```

```
        // Destroi pool de conexões (fechando todas as conexões
abertas)
        public synchronized void destroiPool () throws
PoolConBDException
        {
            if(m_Pool != null)
            {
                for (int i = 0; i < m_Pool.size (); i++)
                {
                    Connection l_con = (Connection) m_Pool.elementAt
(i);
                    try {
                        if(l_con != null) l_con.close ();
                    } catch (Exception p_e) {}
                }

                m_Pool.removeAllElements ();

                try {
                    DriverManager.deregisterDriver(m_DriverJdbc);
                } catch(Exception p_e) {
                    throw new PoolConBDException(m_Aplic, "não foi
possível
                        descadastrar driver jdbc");
                }
            }
        }
    }

    // Hashtable com mapeamento entre nomes das aplicações e seus
respectivos pools
    private static Hashtable m_PoolsAplics = new Hashtable ();

    // Cria um novo pool a partir dos seguintes dados: nome da
aplicação,
    // nome do driver de conexão a base de dados, nome da base,
url / usuário / senha
    // de conexão, tamanho máximo do pool e timeout de conexão a
base de dados.
    public static synchronized void criaPoolAplic(String p_aplic, String

            p_driverJdbc, String p_nomeBD, String p_url, String
```

```
p_usuario, String
          p_senha, int p_tamMaxPool, int p_timeout) throws
PoolConBDException {
       if(p_aplic != null) {
          PoolConBDAplic l_pool = (PoolConBDAplic)
m_PoolsAplics.get(p_aplic);
          if(l_pool != null) l_pool.destroiPool ();

          l_pool = new PoolConBDAplic(p_aplic, p_driverJdbc,
p_nomeBD, p_url,
              p_usuario, p_senha, p_tamMaxPool, p_timeout);
          m_PoolsAplics.put(p_aplic, l_pool);
       }
   }

   // Veja PoolConBDAplic.alocaConexao
   public static Connection alocaConexao(String p_aplic) throws
PoolConBDException
   {
       PoolConBDAplic l_pool = (PoolConBDAplic) m_PoolsAplics.get
(p_aplic);
       if(l_pool != null) return l_pool.alocaConexao ();
       else return null;
   }

   // Veja PoolConBDAplic.obtemNumConsAlocadas
   public static int obtemNumConsAlocadas(String p_aplic)
   {
       PoolConBDAplic l_pool = (PoolConBDAplic) m_PoolsAplics.get
(p_aplic);

       if(l_pool != null) return l_pool.obtemNumConsAlocadas ();
       else return -1;
   }

   // Veja PoolConBDAplic.liberaConexao
   public static void liberaConexao(String p_aplic, Connection p_con)
   {
       PoolConBDAplic l_pool = (PoolConBDAplic) m_PoolsAplics.get
(p_aplic);
       if(l_pool != null) l_pool.liberaConexao(p_con);
```

```
    }

    // Vide PoolConDBAplic.destroiPool
    public static synchronized void destroiPool(String p_aplic)
            throws PoolConBDException
    {
        PoolConBDAplic l_pool = (PoolConBDAplic) m_PoolsAplics.get
(p_aplic);

        if (l_pool != null)
        {
            l_pool.destroiPool();
            m_PoolsAplics.remove(p_aplic);
        }
    }
}
```

É interessante observar que a classe anterior permite que sejam configurados / utilizados não apenas um único pool de conexões a base de dados, mas sim diversos pools: como são declaradas variáveis estáticas, todas as aplicações Web sendo executadas na mesma instância da máquina virtual Java irão compartilhar esses mesmos objetos. Sendo assim, é necessário prever a alocação e liberação de conexões por aplicação, de forma que o funcionamento de uma aplicação não interfira no funcionamento de outra.

Esse gerenciador de conexões ao banco de dados utiliza também uma classe do tipo Exception para indicar falhas em seu funcionamento. Segue a implementação dessa classe PoolConBDException:

### Classe "PoolConBDException": exceções associadas a classe "PoolConBD"

```
package com.livroservlets;
package com.livroservlets;


// Exceção para gerenciador de conexões a base de dados
"PoolConBD"
public class PoolConBDException extends Exception {
```

```
    public PoolConBDException(String p_msg) {super(p_msg); }

    public PoolConBDException(String p_aplic, String p_msg) {
        super("(" + p_aplic + ") " + p_msg);
    }
}
```

Finalmente, temos a seguir um Servlet para exemplificar o uso de nosso gerenciador de pool de conexões:

### Servlet "ServletTestePool": exemplifica o uso de nosso gerenciado de pool de conexões a base de dados

```
package com.livroservlets;
import java.io.*;
import java.sql.*;

import javax.servlet.*;
import javax.servlet.http.*;

import com.livroservlets.*;

// Servlet para teste do pool de conexões a base de dados
public class ServletTestePool extends HttpServlet {

    // Inicializa pool de conexões
    public void init(ServletConfig p_servletConfig)
            throws UnavailableException, ServletException {
        super.init(p_servletConfig);

        if (p_servletConfig != null)
            try {
                PoolConBD.criaPoolAplic("LivroServlets",
"com.mysql.jdbc.Driver",
                    "livroservlets",
"jdbc:mysql://127.0.0.1:3306/livroservlets",
                    "admin", "321", 10, 60);
            }catch(Exception p_e) {
```

```
                throw new UnavailableException("Erro: não foi possível
criar pool; \""
                        + p_e.getMessage () + "\"");
            }
    }

    public void doGet(HttpServletRequest p_request,
HttpServletResponse p_response)
            throws IOException {

        // Definindo headers auxiliares para evitar cacheamento de
página
        p_response.setHeader("Cache- Control", "no- cache, must-
revalidate");
        p_response.setHeader("Pragma", "no- cache");
        p_response.setHeader("Expires", "Mon, 26 Jul 1997 05:00:00
GMT");
        p_response.setDateHeader("Last- Modified",
System.currentTimeMillis ());

        PrintWriter l_pw = p_response.getWriter ();
        l_pw.println("<HTML><BODY>");

        try {
            l_pw.println("Alocando conexão com a base de dados ...
<BR>");
            Connection l_con = PoolConBD.alocaConexao
("LivroServlets");

            // ... executa queries SQL com conexão alocada ...

            l_pw.println("Liberando conexão com a base de dados ...
<BR>");
            PoolConBD.liberaConexao("LivroServlets", l_con);
        } catch(Exception p_e) {
            l_pw.println("Erro na alocação / liberação de conexão a
base de dados!");
        }

        l_pw.println("</BODY></HTML>");
        l_pw.flush ();
    }
```

```
}
```

# Parte II

## Desenvolvimento de Aplicações Distribuídas Utilizando EJB

# Capítulo 10

# Novas Técnicas de Desenvolvimento

## 10.1 Desenvolvimento de Clássico de Aplicações

O mercado de desenvolvimento de aplicações na década de setenta e oitenta era baseado em sistemas centralizados executando sobre um único computador. O desenvolvimento contínuo da tecnologia diminuiu o preço de componentes de hardware. Com a queda no preço de componentes de hardware tornou- se viável a aquisição de computadores, agilizando, principalmente, o processo produtivo de empresas.

As empresas, que adquiriram muitos computadores, começaram a perceber a utilizade em criar canais de comunicação entre estas máquinas, este foi o desenvolvimento prático das redes de computadores. As redes deixaram de estar na teoria de livros e partiu para o cotidiano do mercado. A vantagem das redes de computadores estava ligada, principalmente, à possibilidade de acesso compartilhado de recursos tais como impressoras e arquivos.

Com o desenvolvimento das redes de computadores diversas empresas começaram a oferecer sistemas

operacionais que suportassem tal interconexão em rede. Desenvolveram-se, então, sistemas como o Novell, Windows for Workgroups, Linux que mais tarde foram evoluindo para os sistemas mais utilizados atualmente.

Com o desenvolvimento dos sistemas operacionais de rede, que permitiam o compartilhamento de recursos, os desenvolvedores se depararam com clientes desejando softwares capazes de executar em todos seus computadores, atualizando um mesmo banco de dados. Neste ponto houve o desenvolvimento da tecnologia cliente-servidor, utilizando linguagens tais como Clipper, Visual Basic e Delphi.

Nas implementações da tecnologia cliente-servidor era utilizado um computador central com o banco de dados. Cada um dos computadores que participavam da rede tinha instalado um software que acessava remotamente este banco de dados. Esta foi a forma encontrada para que todos pudessem executar o software com desempenho desejável.

Nos anos noventa, com o desenvolvimento da Internet, um novo mercado foi criado. Este mercado exigiria mais conhecimentos dos desenvolvedores, e especialização até mesmos dos vendedores de tecnologia. Estes sistemas de Internet executavam sobre um determinado computador, que tinha um Web Server (tal como Apache, Microsoft IIS, etc) e uma banco de dados (PostgreSQL, Oracle, etc). Quando um usuário acessasse um servidor via um navegador (Netscape, Opera, Microsoft Internet Explorer, etc), este servidor retornaria uma página, esta página conteria conteúdo em HTML, que mais tarde evoluiu para outros formatos.

No início os servidores de páginas na Internet, hospedavam apenas páginas em HTML com figuras (gifs, jpegs, etc). Páginas HTML não poderiam gerar dinamicamente dados, tal como obter, inserir e alterar informações em um banco de dados. Neste momento houve o desenvolvimento de CGIs (Common Gateway Interfaces). Os CGIs eram pequenos programas em

linguagens como por exemplo C. Estes CGIs eram chamados pelo servidor Web para acessar um banco de dados ou executar qualquer tipo de tarefa que deveria ser dinâmica, e não estática, tal como são as páginas HTML.

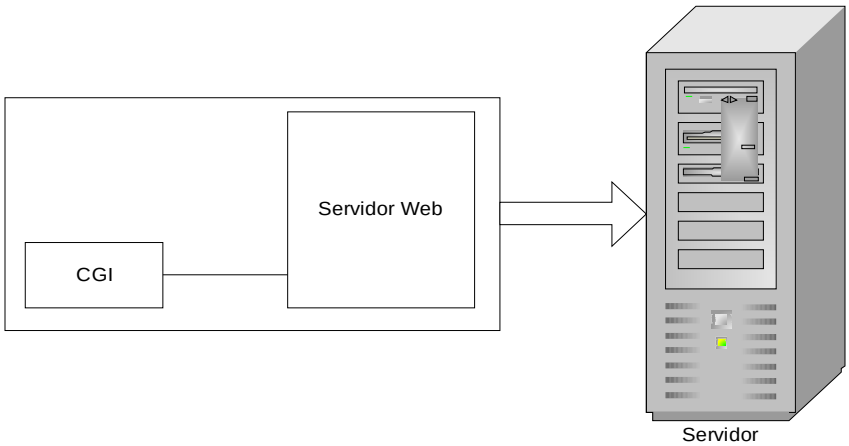


***Figura 10.1 – Servidor WEB com CGI.***

Os CGIs eram mais difíceis de se desenvolver (nesta época se utilizava linguagem C), então diversos desenvolvedores iniciaram a criação de suportes modulares nos sevidores Web para permitir a integração direta com linguagens de desenvolvimento. Os CGIs eram executados como processos à parte do Web Server, o que necessitava a alocação de mais memória e sobrecarregava o sistema, em casos com muito acesso.

Diversas linguagens de script se desenvolveram após os CGIs, tais como PHP, ASP, Perl, Phyton e JSP. A execução de scripts (pequenos programas) criados nestas linguagens era feita como observado na figura 10.1, o que diminuia o consumo de memória do servidor, gerando menos atrasos no tempo de resposta do cliente e minimizando a sobrecarga do servidor.

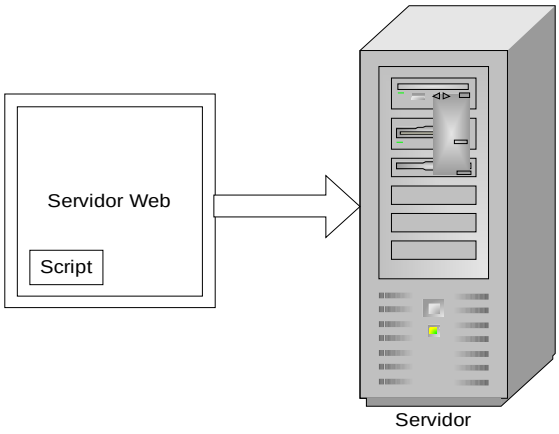


***Figura 10.2 – Suporte a Módulos.***

O desenvolvimento nas linguagens de script é o mais comum até hoje para sistemas na Internet. Contudo, desenvolver neste modelo tem diversas limitações tais como escalabilidade, comunicação entre processos, acesso simultâneo a bancos de dados, transações entre bancos de dados que estão localizados em diferentes computadores e distribuição de carga.

Para solucionar as limitações das linguagens de script diversos trabalhos estavam paralelamente em desenvolvimento. Estes projetos visavam a construção de suportes para a construção de sistemas distribuídos. Os sistemas distribuídos permitem que computadores executem computações de forma cooperativa, e além disto oferecem um grau de transparência para o usuário final. Contudo, desenvolver neste tipo de arquitetura distribuída não era uma tarefa simples, aliás, o desenvolver precisava ter altíssima especialização. Para isto empresas tais como a Sun Microsystems se uniram em torno de um padrão chamado J2EE (Java 2 Enterprise Edition) para o desenvolvimento de aplicações distribuídas, oferecendo aos desenvolvedores uma base sólida e mais simples (não que seja tão simples assim) para desenvolvimento.

## 10.2 Sistemas Distribuídos

Antes de aprofundar em qualquer arquitetura de sistema distribuído, deve-se aprender mais sobre sistemas distribuídos. A definição mais simples de sistemas distribuídos é um conjunto de computadores interligados em rede, que executam operações computacionais de forma cooperativa e transparência para o usuário final. Ter uma rede é o primeiro passo para a construção destes sistemas, o próximo passo é criar um sistema que seja modular, onde cada módulo execute em um computador distinto. Estes módulos trocam informações entre si para executar determinada operação computacional. O termo transparência se refere a criar um nível de abstração entre o usuário e o sistema, desta forma o usuário não sabe que seu sistema executa uma parte em cada computador, simplesmente para ele, o sistema esta sendo executado. Este usuário "enxerga" o conjunto de computadores interligados em rede para execução cooperada de computações tal como um único computador virtual (maiores informações no livro Distributed Systems, autor Andrew S. Tanembaum).

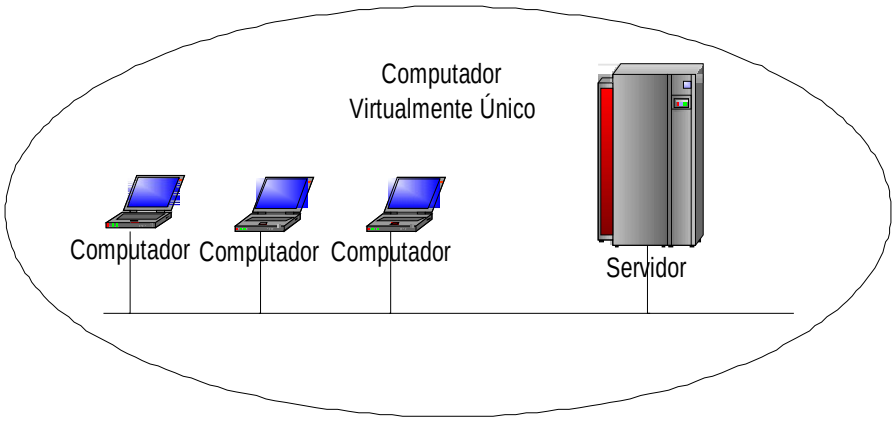


***Figura 10.3 – Computador Virtualmente Único***

Os benefícios de um Sistema Distribuído sobre um sistema que executa em um único computador compreendem a economia, velocidade, desenvolvimento de aplicações naturalmente distribuídas, confiabilidade, crescimento incremental.

Um caso prático da economia encontra-se em certa situação vivenciada por um dos autores que, na época, desenvolvia sistemas para a Internet. Determinado sistema havia sido contruído em PHP, uma linguagem de script, para acessar um banco de dados e realizar determinandas operações. Após certo tempo este sistema tornou-se um dos sites mais acessados do país, e o servidor que o executava ficou cada vez mais carregado. Havia então dois caminhos a se tomar: o primeiro seria comprar um computador maior e com mais capacidade, o outro desenvolver a aplicação novamente para executar como uma aplicação distribuída.

Para tratar da questão que envolvia esta aplicação em PHP foram realizados diversos estudos comparando custos e desempenho computacional. Foi observado que seria necessário adquirir uma workstation da Sun Microsystems para atender a aplicação, caso contrário seria necessário desenvolver novamente. Contudo, caso a aplicação fosse implementada para funcionar sobre um sistema distribuído seriam necessários três computadores pessoais para executá-la, o que era em torno de 12 vezes mais barato.

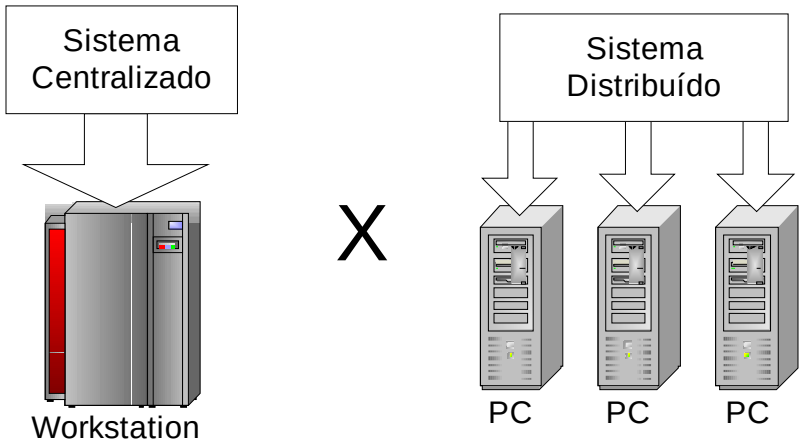


***Figura 10.4 – Economia de Recursos***

A velocidade é outra questão de interessante análise. A velocidade de um hardware chega a limites da própria

física, contudo como ultrapassar estes limites impostos pelos materiais existentes? Utilizar um ambiente distribuído pode colaborar neste sentido, pois subdividir uma aplicação em módulos, que executem em paralelo, cada um em um computador distinto, divide a carga e permite que a aplicação tenha maior desempenho final. Contudo, subdividir uma aplicação em módulos não é uma tarefa trivial.

Um exemplo de aplicação prática para atingir alta velocidade na execução da aplicação é o site de buscas Google (http://www.google.com). Imagine buscas muito complexas, existe um computador único que conseguiria atender a este sistema? Não. Construir um hardware para isto seria viável? Não, pois o custo seria proibitivo. Para resolver este tipo de problema, os envolvidos criaram uma aplicação que executa sobre diversos computadores, particionando as operações de busca e indexação das informações.

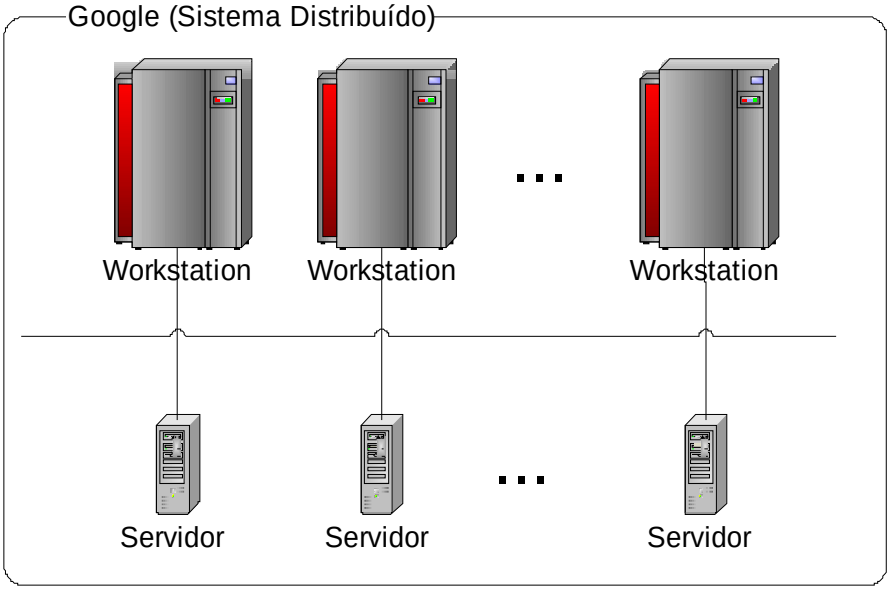


***Figura 10.5 – Google.***

Há aplicações que são naturalmente distribuídas, onde módulos precisam executar tarefas distintas, contudo em algum momento necessitam trocar mensagens para

sincronizar determinadas informações. Este tipo de aplicação é altamente privilegiada pelos sistemas distirbuídos.

A confiabilidade de um sistema pode ser atingida de duas formas: através da replicação de hardware e da replicação de software. Replicar hardware tem o intuito de não deixar o sistema cair em casos onde um dos componentes físicos venha a ter problemas, este tipo de solução é conhecida como tolerância a falhas. A réplica de software tem o intuito de copiar softwares para diferentes computadores, caso um dos computadores pare, outro poderá reiniciar a aplicação e o sistema continua disponível, este tipo de solução é conhecida como alta disponibilidade.

A alta disponibilidade é algo inerente de um sistema distribuído. Como existem vários computadores em uma rede, torna-se muito acessível desenvovler uma aplicação onde sejam criados módulos e réplicas destes módulos possam existir em outros computadores. Caso um dos computadores tenha problemas, outro poderá assumir.

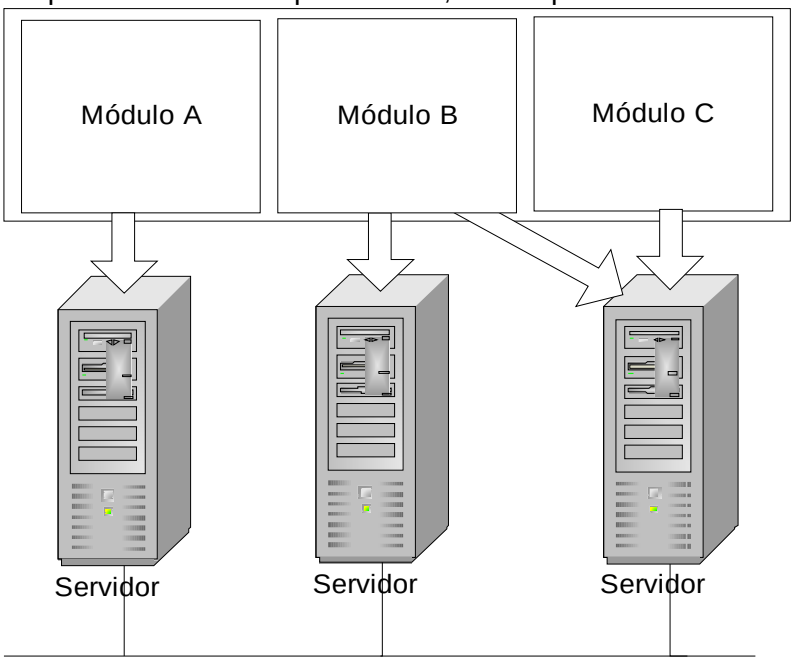


***Figura 10.6 – Alta Disponibilidade.***

O crescimento incremental está relacionado à necessidade do sistema de suportar maiores cargas. Em um sistema centralizado, quando este torna-se muito carregado (veja o exemplo citado sobre o aspecto econômico de sistema distribuídos) deve-se adquirir um novo hardware para executá-lo, e este com certeza será de maior custo. Contudo, numa aplicação distribuída bem subdividida em módulos, no caso do ambiente ficar muito carregado, pode-se adicionar novos computadores ao sistema, redistribuir os módulos entre os computadores de tal forma que atinja maior desempenho e atenda seus novos requisitos.

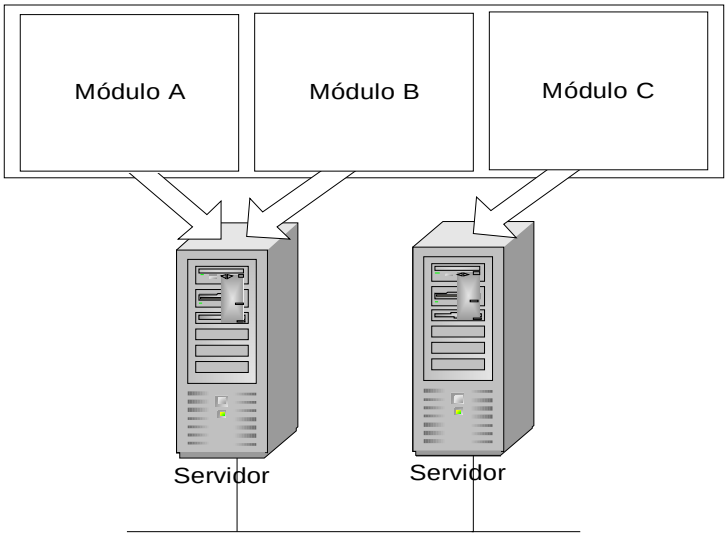


***Figura 10.7 – Módulos do Sistema Distribuído.***

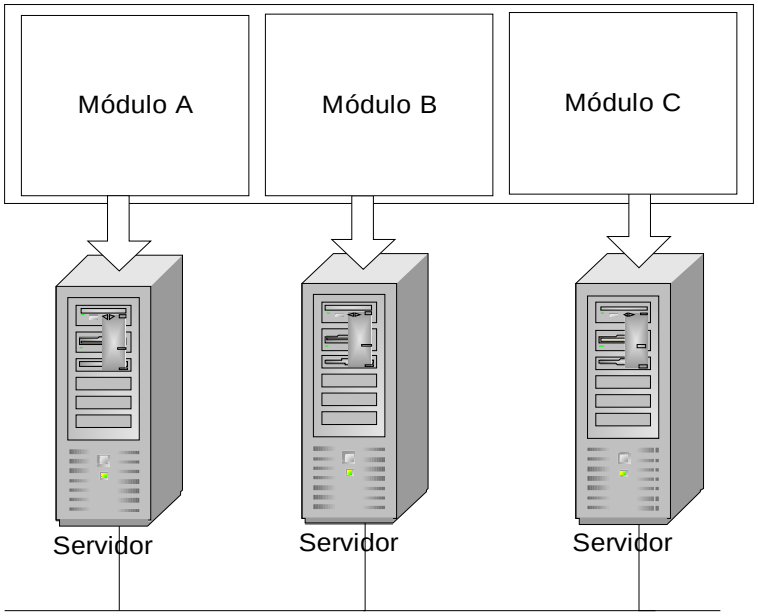


***Figura 10.8 – Módulos do Sistema Distribuído Redistribuídos.***

As desvantagens de um Sistema Distribuído compreendem o desenvolvimento de software, sobrecarga no meio de comunicação, segurança.

Para projetar um sistema distribuído o desenvolvedor precisa de conceitos adicionais e sempre ter em mente aspectos tais como multithreading, acesso compartilhado a recursos, comunicação em rede de computadores, acesso simultâneo a recursos e outros conceitos. Isto tona o desenvolvimento de aplicações distribuídas algo complexo para os desenvolvedores mais comuns.

Subdividir os módulos de uma aplicação distribuída não é uma tarefa simples. Para isto deve-se ter em mente a necessidade de criar módulos que tenham pouca comunicação entre si, sobrecarregando o mínimo possível o meio de comunicação. Tendo, por exemplo, cinco objetos, sendo que três deles comunicam-se em demasia, crie um módulo para estes três, caso os outros tenham

pouca comunicação, subdivida-os entre os demais computadores.

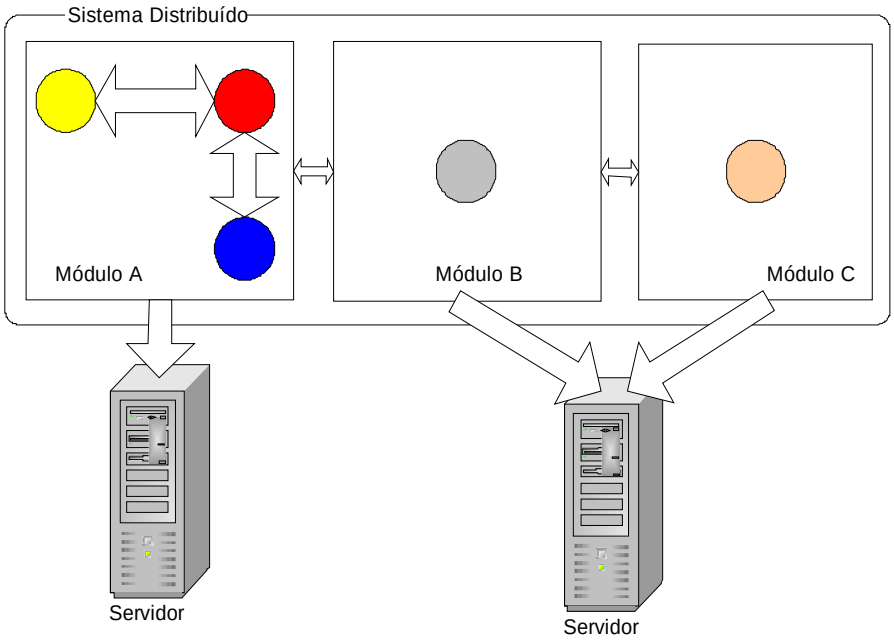


***Figura 10.9 – Subdivisão de Módulos.***

Observando a figura 10.9 pode-se pensar: caso a aplicação fique muito carregada posso redistribuir os módulos B e C, conforme a figura 10.10.

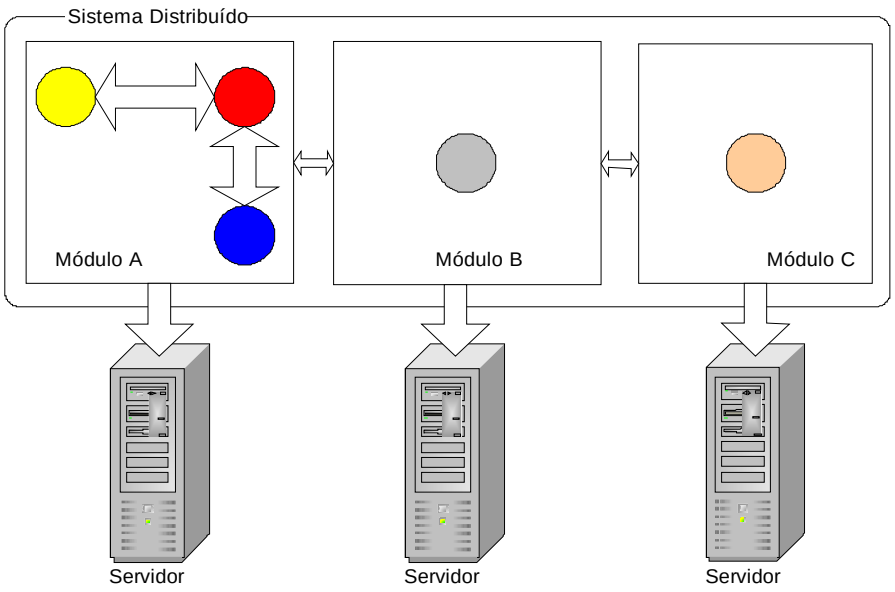


***Figura 10.10 – Redistribuição de Módulos.***

Contudo, se o módulo A tornar seu computador muito carregado como resolver tal questão? Pode-se tentar subdividí-lo, mas se a rede ficar muito sobrecarregada com tal divisão pode ser necessário comprar um computador de alto desempenho somente para executar este módulo. Os sistemas distribuídos minimizam em cerca de 95% dos casos que envolvem aquisição de hardware, contudo há problemas que se resolvidos de forma distribuída podem tornar a rede sobrecarregada e gerar um pior desempenho final.

## Deixando de Confundir um Sistema Operacional de Rede com um Sistema Distribuído

O principal objetivo de um sistema operacional de rede é o compartilhamento de recursos em uma rede de computadores. Suponha uma rede onde existe apenas uma impressora, é mais fácil compartilhá-la para todos os usuários do sistema do que comprar uma impressora para cada computador. Para solucionar este tipo de problema foram desenvolvidos os sistemas operacionais de rede. São

exemplos de sistemas operacionais de rede o Linux e Windows 95/98/NT/2000.

Em um sistema operacional de rede, os computadores são “enxergados” pelos usuários como máquinas distintas. Desta forma, para acessar um determinado recurso, deve-se saber em qual computador ele se localiza. Em um sistema operacional de rede a comunicação entre computadores é realizada através de arquivos compartilhados, isto ocorre tanto no acesso a diretório (ou pastas) compartilhadas, quanto no uso da impressora, que cria uma fila de impressão para recepção de arquivos compartilhados.

Um sistema distribuído oferece ao usuário a imagem de um único recurso computacional. O usuário final não sabe se parte de sua aplicação executa nos computadores A, B e C. Além disto, as partes em que sua aplicação foi subdividida comunicam-se entre si para sincronizar informações e, portanto, executar uma operação em conjunto.

Suponha uma aplicação que necessita realizar muitos cálculos, enquanto outra parte da aplicação utiliza os resultados destes cálculos. Pode-se subdividir esta aplicação em dois módulos. Cada um deles executando em um computador diferente. O projetista do sistema conhece estes aspectos do sistema, contudo o usuário final acredita que o sistema está executando em apenas um computador, pois ele desconhece o funcionamento do sistema e suas operações.

A comunicação em um sistema distribuído é feita através de mensagens que trafegam sobre a rede. Ao contrário dos sistemas operacionais de rede que podem trafegar arquivos completos.

Com as definições anteriores pode-se notar que executar um sistema operacional como Linux ou Windows em uma rede não é ter um sistema distribuído. Ter um sistema distribuído é construir uma aplicação que seja subdividida em módulos, cada um deles executando em um

computador distinto e trocando mensagens entre si para sincronizar todas as tarefas que estão sendo executadas.

## 10.3 Primeiros Ensaios de Arquiteturas para Sistemas Distribuídos no Mercado

### Corba

CORBA (Common Object Request Broker Architecture) é um padrão que definido pela OMG (Object Management Group), organização que reúne cerca de 800 empresas do mundo todo. Este padrão foi desenvolvido para a construção de aplicações distribuídas. Por ser um padrão, para aplicá-lo, deve-se ter acesso a um suporte ou ferramenta que o implemente. Diversas empresas e interessados desenvolveram suas próprias versões seguindo o padrão Corba, dentre estas pode-se destacar o Mico (http://www.mico.org), OmniOrb (), Visibroker (), Jacorb () entre outros.

Segundo o padrão Corba, aplicações para um ambiente distribuído podem estar sendo executadas em diferentes plataformas de hardware e sistemas operacionais. Além disto, podem ter sido construídas em diferentes linguagens de programação tais como C, C++, Java ou Delphi.

Uma das vantagens do padrão Corba é ser aberto, isto permitiu que várias empresas implementassem suas próprias versões, deixando de limitar o desenvolvedor tal como ocorre com soluções proprietárias. E o mais importante, fossem interoperáveis entre si. Isto de fato é o objetivo, contudo fazer duas versões de Corba diferentes interoperarem não é tão simples quanto parece.

Um desenvolvedor contrói sua aplicação sobre uma versão do Corba, e desta forma, pode distribuir as partes desta

aplicação em uma rede. Assim a aplicação irá funcionar como um sistema distribuído.

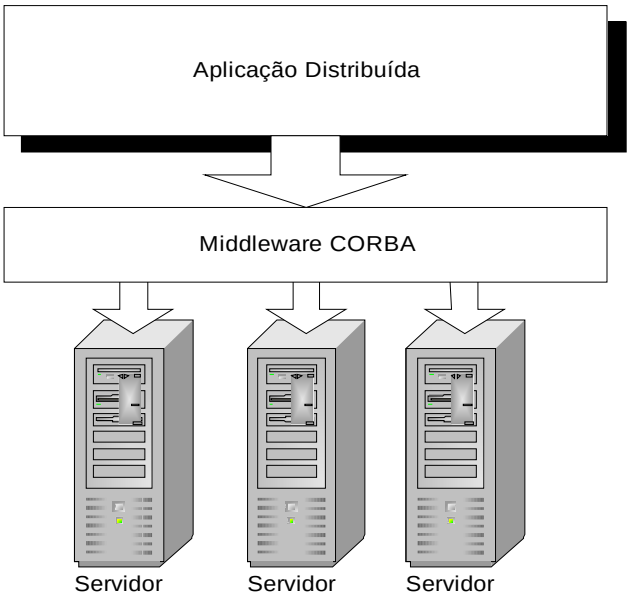


***Figura 10.11 – Corba.***

A principal limitante no crescimento do uso de Corba é a complexidade em desenvolver para esta arquitetura. Há muitas exigências para os desenvolvedores, que necessitando de produtividade acabaram deixando esta arquitetura.

## Java/RMI

Com o desenvolvimento da plataforma Java, a Sun Microsystems observou um grande horizonte no desenvolvimento de aplicações em rede de computadores e iniciou o desenvolvimento de um suporte para objetos distribuídos chamado Java/RMI. RMI significa Remote Method Invocation, ou seja, a invocação de métodos remotos. Este suporte simplificou a construção de aplicações distribuídas, contudo funciona apenas para a linguagem Java, ao contrário de Corba.

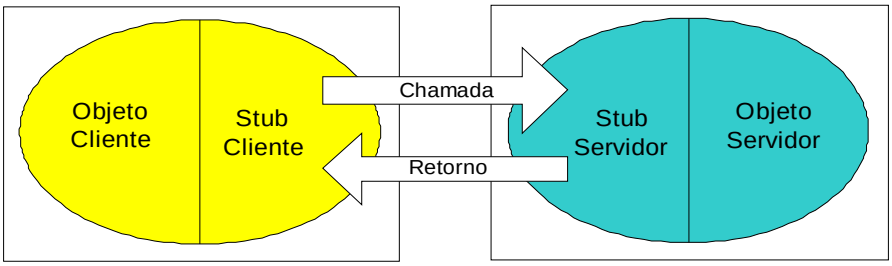


***Figura 10.12 – Java/RMI.***

RMI contou com o apoio de diversas empresas, que interessadas na plataforma Java, desenvolveram uma série de IDEs (Integrated Development Environments), aquelas ferramentas gráficas que simplificam o desenvolvimento, prontas para implementar usando a nova tecnologia. Isto agregou muito e diversos desenvolvedores voltaram sua atenção para Java.

RMI possibilitava as mesmas funcionalidades que Corba, contudo, com o apoio de fabricantes, em pouco tempo surgiram boas ferramentas e aplicações neste suporte. Além disto, a Sun Microsystems criou o suporte Java/IDL para comunicar Java com Corba, isto permitia que desenvolvedores utilizando Corba pudesse até mesmo migrar para Java e continuar o desenvolvimento em RMI.

## J2EE

Observando o lado promissor da tecnologia Java/RMI a Sun Microsystems desenvolveu um padrão chamado J2EE (Java 2 Enterprise Edition). Com isto, a empresa segmentou a tecnologia Java e começou cada vez mais a se preocupar com o mercado de aplicações distribuídas.

J2EE é uma arquitetura que utiliza a mesma pilha de protocolos de Java/RMI o que permite comunicação com Corba, e além disto, permite continuidade aos desenvolvedores Java/RMI. Nesta tecnologia uma série de suportes foram oferecidos. Programar para um ambiente distribuído tornou-se mais simples, pois uma base sólida de componentes haviam sido desenvolvidos para isto.

Diversos fabricantes se interessaram pela arquitetura J2EE, mais ferramentas de desenvolvimento foram lançadas e o mercado aumentou sua aceitação para esta tecnologia. Dentre os fabricantes destaca-se a Oracle, IBM, Sun Microsystems, Bea Systems, etc.

### Microsoft .NET

Observando a conquista do mercado pelo J2EE, a Microsoft não se agüentou e lança sua própria tecnologia para o desenvolvimento de aplicações distribuídas. Esta tecnologia é proprietária e não um padrão aberto tal como Corba e J2EE, por isto, somente a Microsoft a oferece.

## 10.4 Mercado Atual para Sistemas Distribuídos

O mercado atual tem se voltado cada vez mais para a tecnologia Java, e dentro deste enfoque a arquitetura para sistemas distribuídos é a J2EE. Apesar da grande evolução desta arquitetura, o projeto de aplicações distirbuídas ainda exige conhecimentos adicionais dos desenvolvedores. Cada vez mais desenvolver torna-se uma tarefa que exige estudos sobre conceitos e novas tecnologias, ao contrário de estudar uma linguagem comum.

O mercado tem optado por tecnologias baseadas em plataformas abertas e isto pode ser cada vez mais notado. Empresas como IBM e Oracle voltaram-se muito para este segmento de mercado, suportando sistemas operacionais livres como o Linux e partindo para a tecnologia Java. Aos poucos as empresas notam que tecnologias fechadas, tais como o Microsoft .NET, limitam o cliente, pois há a dependência única e exclusiva da Microsoft, ao contrário do J2EE que é oferecido por vários fabricantes.

Não se encontrar neste mercado promissor é algo que preocupa muitos projetistas e desenvolvedores. Portanto, conhecer uma tecnologia como J2EE é necessário e abre

caminhos, não só no cotidiano de escrita de código, mas sim nas técnicas e escolha das melhores arquiteturas e opções para desenvolvimento.

# Capítulo 11

# J2EE e Enterprise JavaBeans

## 11.1 O que é J2EE?

J2EE, ou Java 2 Enterprise Edition, é uma plataforma para desenvolvimento de aplicações distribuídas. Apresenta facilidades para a utilização dos recursos computacionais e distribuídos tais como acesso à banco de dados, componentes Web, utilização de mensagens assíncronas, execução de processos transacionais, persistentes ou não etc.

Apresenta uma API, especificada pela Sun MicroSystems, que proporciona um padrão para a implementação dos diversos serviços que oferece, sendo que isto pode ser feito diferentemente por várias empresas, de formas distintas mas ainda assim oferecendo as mesmas facilidades, por estarem de acordo com as especificações impostas para a sua construção.

Para um programador que já tenha tido contato com a linguagem Java e suas APIs na J2SE (Java 2 Standart Edition), este não terá muitas dificuldades no entendimento e na utilização de J2EE. O que precisa-se é entender os detalhes da arquitetura e onde se encontram cada componente e seus recursos, isto é, se faz necessário se ambientar neste contexto para se fazer o uso correto da plataforma.

## 11.2 Visão da plataforma

A arquitetura J2EE se apresenta em várias camadas, sendo que cada camada é composta por componentes e serviços que são providos por um container. A idéia de container e componentes pode ser facilmente entendida por meio de um exemplo.

Imagine uma colméia de abelhas, que contém abelhas obviamente, pulpas, zangões, a abelha rainha, o mel real etc. Podemos fazer um paralelo e entender como container a colméia, que fornece recursos para as abelhas sobreviverem. Por sua vez, as abelhas em suas diferentes funções, tais como as operárias e as reprodutoras, podem ser vistas como os componentes que sobrevivem dentro do container, isto é, a colméia. Podemos ainda expandir esse exemplo em um apiário, imaginando que cada colméia seja um container e todas as colméias juntas, ou seja, o apiário, represente o servidor J2EE.

Outro exemplo um pouco mais técnico, é o uso de páginas HTML em um Web Browser em uma simples navegação em um site qualquer. Podemos entender como container, o próprio navegador que fornece recursos e facilidades para o componente, neste caso as páginas HTML. O componente por sua vez, pode oferecer diversos serviços ao usuário, através do suporte do container, tais como facilidades visuais como botões, hiperlinks, figuras e tabelas, e o próprio serviço de navegação.

Em um servidor J2EE, podemos ter diversos containers interagindo entre si.

Veremos a seguir uma breve explicação de cada camada da arquitetura e de seus componentes.

Camada cliente: acesso por meio de interfaces stand-alone (aplicações Java), páginas HTML ou Applets. Nesta camada os componentes residem em um Applet Container, em um HTML container (Web browser) ou em um Application Client Container. O Applet container, fornece recursos para um componente Applet executar e se tornar funcional para o usuário. O Web Browser

apresenta recursos e funcionalidades para o uso de páginas HTML e por fim o Application Client container fornece recursos para a execução das classe stand-alone utilizadas pelos usuários para interagirem no sistema.

Camada Web: esta camada é implementada por JSPs e Servlets, que fornecem a lógica para a camada cliente ( ou de apresentação ) do negócio. JSPs e Servlets residem no Web Container. JSPs oferecem a facilidade de utilizar algumas lógicas de apresentação em uma página web sem muitas dificuldades tecnológicas. O Servlet apresenta-se como um controlador das ações executadas pelos usuários nas páginas de apresentação, e fornece recursos para obtêr dados dessas ações e realizar as operações desejadas. Os componentes Web residem no Web Container que pode ser um servidor TomCat ou outro similar.

Camada de Negócios: esta camada trata da lógica de negócio da aplicação. É nela que implementa-se todas as regras de negócio, alocação de recursos, persistëncia de dados, validação de dados, gerencia de transações e segurança, providos por componentes conhecidos por EJBs. Este por sua vez residem no EJB Container.

Camada EIS - Enterprise Information System, ou Sistema de informações empresariais: nesta camada é que se encontram os sistemas de banco de dados, sistemas legados, integração com outros sistemas não J2EE etc.

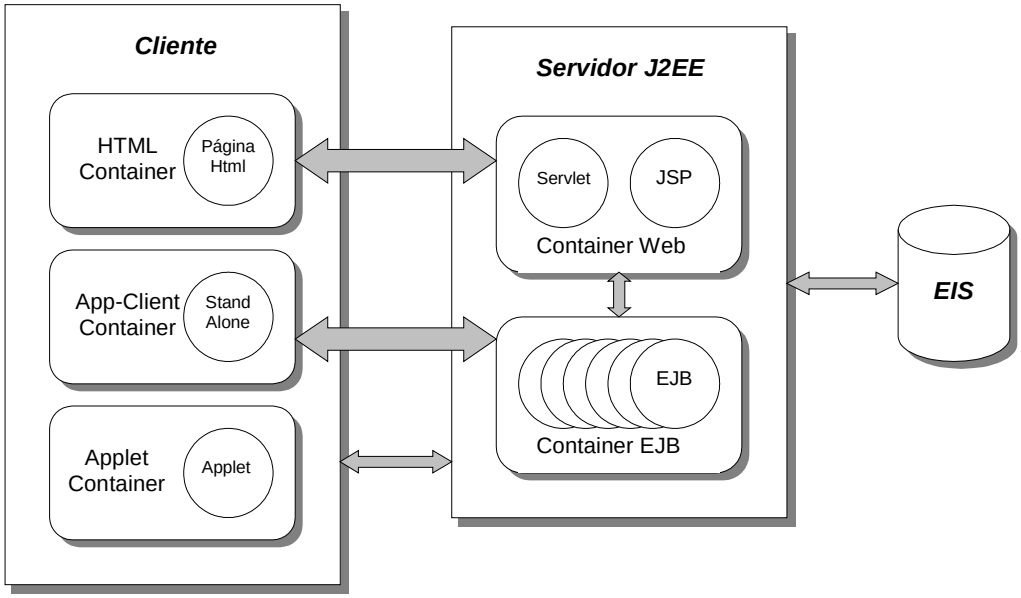


***Figura 11.1 – Camadas, seus respectivos componentes e containers e a interação entre eles.***

# 11.3 Instalando o J2SDKEE

Para obter o kit de desenvolvimento para J2EE, acesse o site da Sun MicroSystems em J2EE - java.sun.com/j2ee e faça o download do J2SDKEE e de suas documentações. A sua instalação segue praticamente o mesmo esquema da versão J2SE. Por fim se faz necessário a criação da variável de ambiente J2EE_HOME. Inclua também o diretório J2EE_HOME/bin na variável de ambiente PATH. Para acessar a biblioteca J2EE, aponte o classpath da sua aplicação para o diretório J2EE_HOME/lib/j2ee.jar. Os pacotes desta API tem o prefico javax e nele podem ser encontrados todos os recursos disponíveis para a especificação J2EE.

Utilizaremos para a execução dos nossos exemplos, o servidor de aplicações da Sun MicroSystems, que também faz parte do kit J2EE e pode ser acessado no diretório J2EE_HOME/bin/j2ee.bat. Um outro aplicativo muito útil a

ser utilizado para realizar a instalação dos componentes no servidor será o deploytool, também disponível no kit e acessado no mesmo diretório.

## 11.4 O que são Enterprise JavaBeans?

Enterprise JavaBeans são objetos distribuídos que apresentam uma estrutura bem definida, isto é, implementam interfaces específicas e que rodam no lado do servidor. Também são conhecidos como EJBs (Enterprise JavaBeans) e serão tratados dessa forma neste livro.

São nada mais do que simples objetos que devem seguir algumas regras. Estas regras foram definidas pela Sun MicroSystems através da especificação de EJBs na arquitetura J2EE.

Apresentam métodos para a lógica de negócio e métodos que tratam da criação (instanciação), remoção, atualização do EJB entre outros, dentro do ambiente aonde sobrevive. (Isto será abordado durante o livro no tema ciclo de vida do EJB).

Conforme definido pela Sun MicroSystems – "Enterprise JavaBean é uma arquitetura para computação distribuída baseada em componentes ..."

Devemos entender que EJBs, não são simples classes Java, mas sim componentes distribuídos que fornecem serviços e persistência de dados, além de processamento assíncrono e que podem ser invocados remotamente.

## 11.5 Para que servem e por que utilizá-los?

EJBs são normalmente utilizados para executarem a lógica de negócio do lado do servidor de forma distribuída. Podemos ter EJBs sobrevivendo em ambientes distintos, em máquinas diferentes, em locais geograficamente diversos e ainda assim utilizando de serviços eficientes.

Utilizando EJBs, sua aplicação irá se beneficiar de serviços como transações, segurança, tolerância a falhas, clustering, distribuição, controle de sessão entre outros. Estes serviços são fornecidos pelo ambiente que o EJB sobrevive, o container EJB, que será visto em mais detalhes nos capítulos seguintes.

EJBs residem em um mundo chamado container. Este local conhece muito bem a interface implementada pelos EJBs e sendo assim, consegue tratar cada tipo de EJB diferente um do outro e de forma correta.

Veremos mais adiante que o cliente que deseja utilizar um EJB, não acessa-o diretamente, mas sim utiliza-o através do container, que encaminha as chamadas de método ao EJB e retorna a chamada ao cliente quando for necessário.

Vejamos a seguir como isto é feito pelo container, implementando um exemplo de acesso remoto a um serviço, utilizando a API de Sockets e o recurso de serialização de objetos.

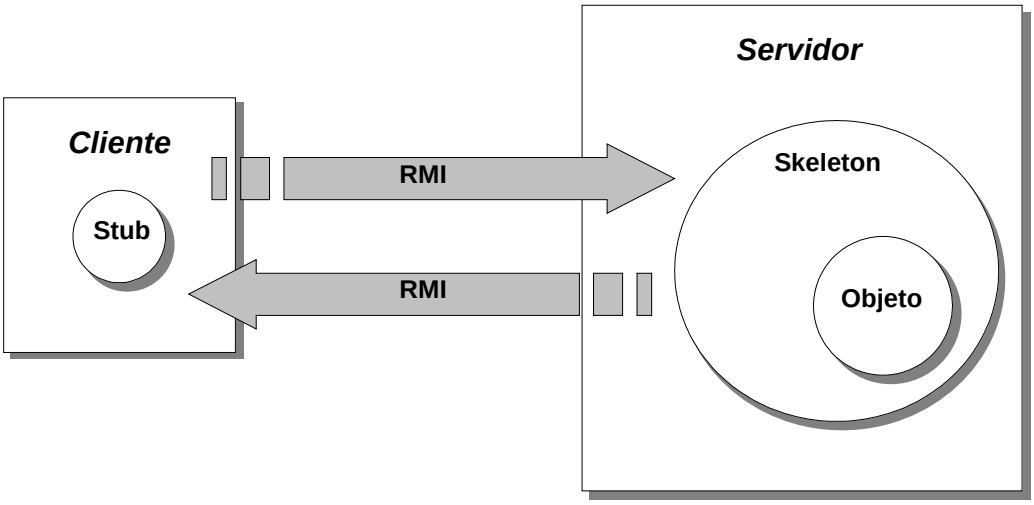


***Figura 11.2 – Exemplo de serviço e troca de objetos utilizando Sockets e Serialização.***

# 11.6 Componentes EJB

Os tipos de Enterprise JavaBeans especificados até a edição deste livro são:

**Session Bean** : Stateless e Stateful

**Entity Bean** : Bean-Managed Persistence e Container-Managed Persistence

**Message-Driven Bean**

Um EJB Session Bean provê serviços, isto é, define métodos de negócio que podem ser acessados remotamente e que disponibilizam operações relevantes à aplicação. O tipo Session Bean Stateless não apresenta um estado como o próprio nome já diz, e fornece serviços para clientes locais e remotos. O EJB Session Bean Stateful, também fornece serviços localmente ou remotamente, mas apresenta uma relação forte com um cliente, isto é, mantém o estado que um cliente define, assim este cliente pode configurar e recuperar dados deste EJB.

Os EJBs Entity Beans representam entidades, objetos que são persistidos. Podem apresentar a manipulação e persistência do objeto de duas formas: BMP ou CMP. No tipo BMP (Bean-Managed-Persistence), o código de persistência e manipulação do objeto deve ser fornecido pelo Bean, isto é, deve ser programado. Já o tipo CMP (Container-Bean-Managed) é provido pelo próprio container, não tendo a necessidade de escrever linhas de código para estas operações.

Message-Driven-Bean são EJBs que fornecem serviços assíncronos e podem ser comparados a Session Beans Stateless, que também fornecem serviços aos clientes locais e remotos, mas de forma assíncrona. Um EJB do tipo Message-Driven-Bean se comporta como um listener que aguarda o recebimento de mensagens através de um MOM (Middleware Oriented Message).

Detalhes de cada tipo de EJB serão vistos na *Parte II – Tipos de Enterprise JavaBeans*.

## 11.7 Classes e interfaces

Cada tipo de EJB deve implementar interfaces diferentes e definidas pela API J2EE. Estas interfaces definem o comportamento que o EJB deve apresentar.

Além de implementar uma interface definida pela API, devemos criar duas interfaces que serão utilizadas pelos clientes para acessarem os EJBs. Estas interfaces são conhecidas como Local e Remote e devem ser definidas para os EJBs do tipo Session Bean (Stateless e Stateful) e Entity Bean (BMP e CMP).

Para EJBs do tipo Message-Driven Bean não precisamos definir nenhuma interface e conforme veremos em um próximo capítulo.

Não se preocupe com detalhes destas classes e interfaces neste momento, pois logo adiante detalharemos cada uma delas, nos tipos específicos de EJB.

## 11.8 Acesso local e/ou remoto

O acesso remoto é utilizado quando o EJB e o cliente estão em máquinas diferentes. Se o cliente e o Enterprise JavaBean estiverem na mesma máquina o acesso remoto será realizado igualmente. Acessamos um Enterprise JavaBean na mesma máquina ou em outra máquina da mesma forma (transparência).

Para criarmos um Enterprise JavaBean com acesso remoto, devemos implementar a interface Remote e a interface Home. A interface Remote define os métodos de negócio específicos do EJB e a interface Home define os métodos do ciclo de vida do EJB. Para os Entity Beans a interface Home também define os métodos de busca (create e finders).

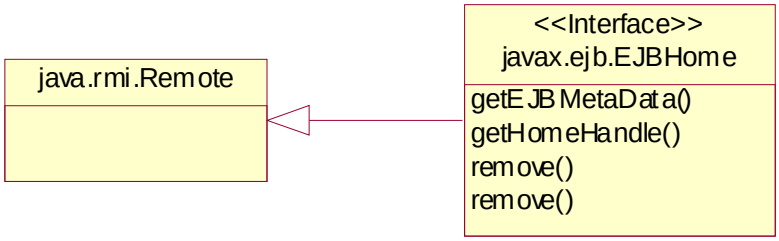


***Figura 11.3 – Diagrama de Classes UML da Interface Home.***

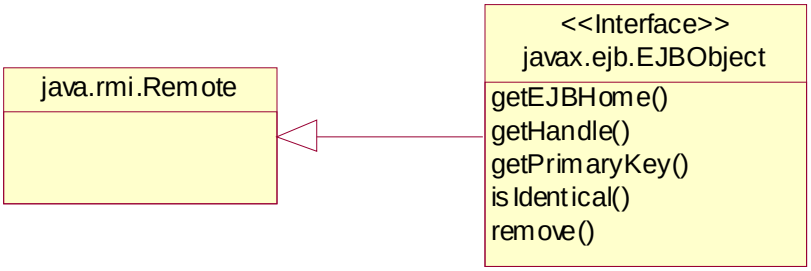


***Figura 11.4 – Diagrama de Classes UML da Interface Remote.***

No acesso local, o cliente e o EJB devem estar na mesma JVM. O acesso ao EJB não é transparente, dessa forma devemos especificar que o acesso é local. O acesso local

pode ser usado em vez do remoto para melhora no desempenho do negócio, mas deve-se fazer isto com cautela, pois um EJB definido como acesso local não pode ser executado em container em forma de cluster, isto é, não pode ser acessado remotamente de forma alguma.

Para criarmos um Enterprise JavaBean com acesso local, devemos implementar a interface Local e a interface LocalHome. A interface Local define os métodos de negócio específicos do EJB (assim como a interface Remote) e a interface LocalHome define os métodos do ciclo de vida do EJB (assim como a interface Home). Para os Entity Beans a interface LocalHome também define os métodos de busca (finders).

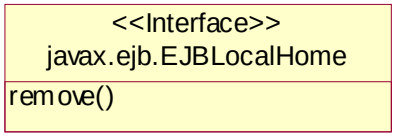


***Figura 11.5 - Diagrama de Classes UML da Interface LocalHome***

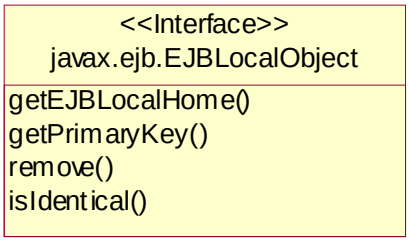


***Figura 11.6 - Diagrama de Classes UML da Interface Local***

Para um EJB do tipo Message-Driven Bean, não precisamos implementar nenhuma dessas interfaces porque, como veremos no *Capítulo 5 – Message-Driven Beans*, este tipo de Enterprise JavaBean apresenta um comportamento diferente de um Session Bean e um Entity Bean, proporcionando processamento assíncrono. O que precisamos fazer é implementar uma interface de Listener que será associado ao MOM.

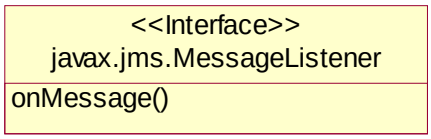


***Figura 11.7 – Diagrama de Classes UML da Interface Listener.***

## 11.9 EJBObject e EJBHome

Como podemos observar ao longo dos tópicos explicados até este ponto, vimos que os componentes EJB não são acessados diretamente, isto é, não acessamos a instância do Bean diretamente, mas fazemos o acesso aos serviços disponíveis por eles através de interfaces que são disponibilizadas para acesso remoto ou local.

Outro detalhe que pode ser observado, que apesar de utilizarmos as interfaces de um componente EJB para acessar seu métodos, vimos que a implementação dos seus serviços oferecidos, isto é, o Bean não implementa as interfaces oferecidas por ele. Bem, conhecendo a definição de interfaces e herança devemos nos perguntar:

Por que o Bean não implementa as interfaces locais e remotas? E não implementando estas interfaces, como é possível acessar os métodos contidos no Bean?

Esta pergunta pode ser respondida simplesmente pela explicação de como o container se comporta com os componentes EJB. Cada fabricante de servidores de aplicação provê a implementação para as interfaces remote e home definidas para o componente e que são respectivamente as classes EJBObject e EJBHome.

A classe EJBObject implementa a interface remote para os acessos remotos e locais e encapsula (wraps) a instância do EJBque foi solicitada pelo cliente. O EJBObject é criado baseado nas informações contidas nos arquivos de deploy e na classe de Bean.

No caso da classe EJB Home, esta se comporta da mesma forma que a classe EJB Object. Ela implementa todos os métodos da interface home para os acessos remotos e locais e ajuda o container a gerenciar o ciclo de vida do Bean, tais como sua criação, remoção etc.

Quando um cliente solicita uma instância de um EJB através da interface home pelo método create(), a classe EJB Home cria uma instância da classe EJB Object que faz referência à instância do EJB solicitado. A instância do EJB é associada com a classe EJB Object e o método ejbCreate() implementado no Bean é chamado. Depois que a instância é criada, a classe EJB Home retorna uma referência para a interface remote (o stub) da classe EJB Object para o cliente.

Com a referência da interface remota, o cliente pode executar os métodos de negócio do Bean. Estas chamadas são enviadas do stub para a classe EJB Object que repassa as chamadas para os métodos corretos na instância do Bean. No caso de retorno de valores nos métodos, o mesmo faz o caminho de volta pelo mesmo caminho utilizado na chamada do método, retornando os valores para o cliente.

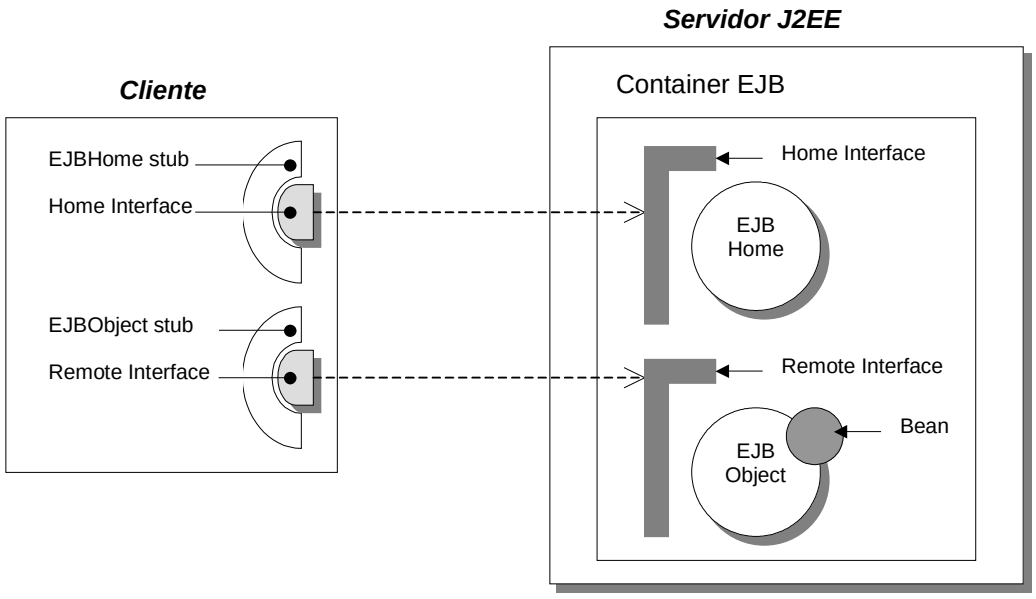


***Figura 11.8 – Cliente acessando o servidor EJB, com as classe EJBHome e EJBObject (encapsulando o EJB).***

# 11.10 Como construir, executar e acessar os componentes

Construir um EJB pode parecer difícil, mas apesar de ser uma tarefa demorada, não apresenta uma complexidade muito elevada. Esta demora pode ser diminuída utilizando de ferramentas que propiciam a sua criação de uma forma automatizada.

Primeiro se faz necessário identificar qual tipo de EJB, ou quais tipos de EJB serão necessários para uma determinada aplicação. Definido os EJBs que farão parte da aplicação, deve-se definir os métodos de negócio de cada um deles, ou seja, definir o comportamento de cada um. Após isso, começamos o desenvolvimento do EJB.

Com os EJBs definidos, e assim, com seus métodos de negócio definidos, devemos criar as interfaces necessárias que serão usadas pelos clientes para o acessarem. No caso

de Session Beans ou Entity Beans, devemos definir a interface Remote (ou Local caso o acesso seja somente local) com os métodos de negócio do EJB. Logo após, definimos a interface Home (ou LocaHome para acesso local) com os métodos do ciclo de vida do EJB, isto é, normalmente com os métodos de criação do EJBe métodos de busca (utilizados em Entity Beans e conhecidos como finders).

Como já foi dito, para o EJB Message-Driven Bean não precisamos definir as interfaces Home (ou LocalHome) e Remote (ou Local), pois o mesmo se comporta diferentemente dos outros EJBs.

Por fim, devemos criar o EJB propriamente dito, implementando a interface específica de cada tipo de EJBe codificando cada método de negócio.

Depois de ter construído os EJBs necessários para uma aplicação, devemos empacotar os mesmo em um arquivo, ou em arquivos separados.

Para empacotar um Enterprise JavaBeans, devemos utilizar do utilitário "jar", que é fornecido juntamente com o JSDK e que facilita a criação do Java Archive (JAR). Com uma linha de comando e alguns argumentos, empacotamos os EJBs em arquivos JAR.

Observe que um arquivo JAR pode ser instalado tranqüilamente em um servidor de aplicação, mas há um outro tipo de arquivo, que empacota todos os recursos da aplicação além dos EJBs, em um mesmo arquivo. Este arquivo é conhecido como EAR (Enterprise Archive) e contempla arquivos JAR, arquivos WAR (Web Archive), arquivos RAR (Resource Adapters Archive), arquivos de configuração, figuras entre outros recursos.

É extremamente aconselhável o uso deste tipo de arquivo, para a instalação de aplicações Enterprise, conforme especificado pela Sun MicroSystems.

Estas operações serão descritas com mais detalhes na Parte III – Instalando e Executando EJB.

Após ter feito isso, podemos instalá-los em um servidor de aplicação. Note que o processo de instalação de um EJB nos servidores de aplicação variam de acordo com cada fabricante, então é sugerido uma leitura da documentação do servidor de aplicação específico.

Para acessar um EJB, precisamos criar um cliente que consiga localizar o EJB onde ele está residindo (servidor de aplicação - container), obtêr uma referência ao EJB remoto ou local e obtêr uma instância para acessarmos os métodos de negócio do EJB. Claro que para o EJB do tipo Message-Driven Bean, como apresenta um comportamento diferente e se propõe a solucionar um problema diferente, utilizamos seus serviços de forma diferente também.

No caso mais comum devemos realizar um “lookup”, isto é, procurar pelo EJB no servidor de aplicação que ele está instalado. Quando criamos um EJB, definimos um nome para ele, e este nome será utilizado pelo cliente para localizar o EJB. Com a referência do EJB em mãos, podemos acessá-lo pela sua interface Home (ou LocalHome), obtêr uma referência para a interface Remote (ou Local) e executar os métodos de negócio desejados.

Lembre-se que o acesso ao EJB pelo cliente é realizado através das interfaces Home e Remote, sendo que o acesso diretamente à instância do EJB propriamente dito é de responsabilidade do container, que opera sobre o Enterprise JavaBean e executa seus métodos e retorna valores, através das solicitações dos clientes, por meio das interfaces (stubs e skeletons).

# Capítulo 12

# Session Beans

## 12.1 O que são Session Beans?

Session Beans são componentes que apresentam serviços para seus clientes. Estes serviços são fornecidos para o cliente pelas interfaces do EJB Session Bean e implementadas pelos métodos de negócio no próprio Bean. O estado do objeto Session Bean consiste no valor da instância de seus atributos, sendo que estes não são persistidos.

Imagine que uma aplicação necessite realizar alguns cálculos e retornar este valor para o cliente, sendo que deseja-se que este serviço seja remoto para ser acessado por vários clientes pelo país. Isto pode ser implementado por um EJB do tipo Session Bean e disponiblizado em um servidor de aplicações para todos os clientes. Este é um exemplo de utilização de um Session Bean.

Mais adiante veremos os códigos fonte de Session Bean exemplo.

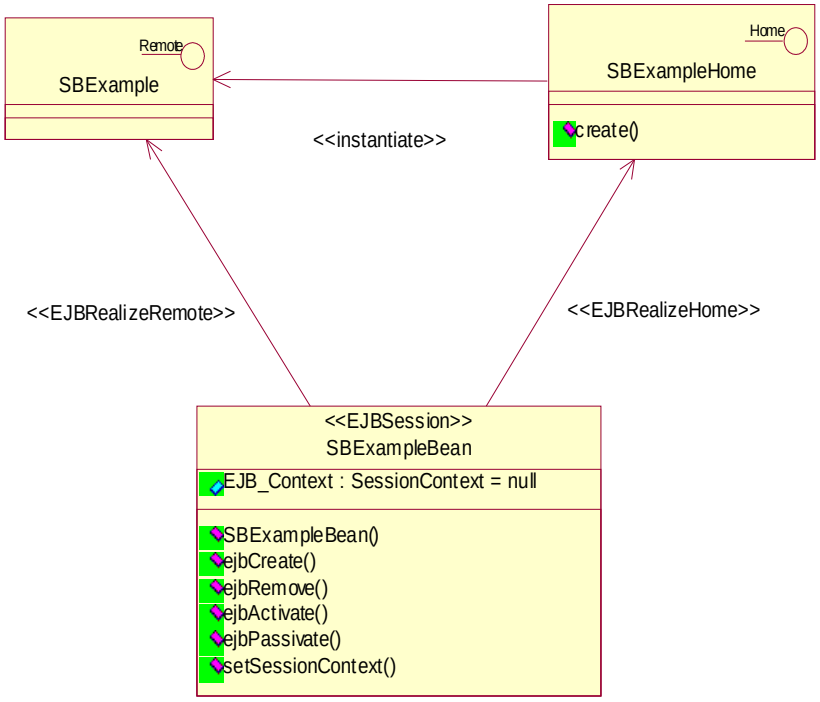


***Figura 12.1 – Diagrama de Classes UML do Session Bean.***

Analisando o diagrama de classes acima temos uma classe e duas interfaces. A classe do EJB Session Bean SBExampleBean (EJBSession) e as interfaces SBExample (Remote) e SBExampleHome (Home). Para cada classe do Bean devemos definir as interfaces (Remote e/ou Local e Home e/ou LocalHome). No exemplo acima, foram definidas as interfaces Remote e Home, mas poderiam ter sido definidas as interfaces Local e LocalHome ou todas elas (com isto teríamos acesso local e remoto ao mesmo EJB).

O Bean deve conter os métodos definidos para um EJB Session Bean conforme a API J2EE que são: ejbCreate(), ejbRemove(), ejbActivate(), ejbPassivate() e setSessionContext(). Também devem apresentar os métodos de negócio com suas devidas implementações que serão os serviços disponibilizados pelo EJB.

Na interface Home do EJBdevemos definir o método create (), que será utilizado pelo cliente para solicitar ao container que crie uma instância do Bean e forneça uma referência para acessar os seus serviços ou os métodos de negócio através da interface Remote.

### Interface Home : SBExampleHome

```
package com.book.example.ejb.session;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

public interface SBExampleHome extends javax.ejb.EJBHome {
    public SBExample create() throws CreateException,
RemoteException;
}
```

Na interface Remote do EJBdevemos definir os métodos de negócio que fornecerão ao cliente os serviços disponibilizados pelo Bean. Estes métodos tem a mesma assinatura tanto na interface Remote quanto na própria implementação do Bean.

A seguir veremos como poderia ser definido esta interface, com um exemplo de serviço fornecido pelo Bean, para calcular o valor de um desconto informando alguns parâmetros.

### Interface Remote: SBExample.

```
package com.book.example.ejb.session;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

public interface SBExample extends javax.ejb.EJBObject {
    public Integer calcDiscount(Integer value, Integer range) throws
RemoteException;
}
```

A seguir apresentamos a implementação do Bean deste exemplo.

**Bean: SBExampleBean.**

```
package com.book.example.ejb.session;

import javax.ejb.*;

public class SBExampleBean implements SessionBean {

    private SessionContext sessionContext;

    public void ejbCreate() throws CreateException {
    }

    public void ejbRemove() {
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }

    public Integer calcDiscount(Integer value, Integer range) {
    // implementação deste método de negócio.
    }
}
```

Para obter este serviço, o cliente deve realizar a localização do EJBno servidor de aplicação utilizando a API JNDI, solicitar uma referência para a interface Home do EJB e com ela executar o método de ciclo de vida: create(). Assim, o cliente terá acesso à interface Remote que apresenta os métodos de negócio, isto é, os serviços

disponíveis para o EJB e dessa forma, poderá executá-los para as operações desejadas.

Maiores detalhes de localização e obtenção das referências para as interfaces, serão vistas no *Apêndice A* que apresentará exemplos mais detalhados de EJB e a utilização de cada serviço, além de especificar os arquivos de instalação (deployment descriptors).

Até aqui introduzimos o EJB Session Bean, mas ainda não detalhamos os dois tipos de Session Bean que serão vistos nos próximos tópicos e que são:

Session Bean Stateless

Session Bean Stateful

## 12.2 Quando usar um Session Bean?

Deve-se utilizar um Session Bean quando deseja-se prover serviços a seus clientes, sendo que estes serviços sejam transacionais e seguros, rápidos e eficientes. Session Beans apresentam uma forma de executar a lógica de negócio do lado do servidor, com todos os ganhos que um servidor de aplicação apresenta e vistos nos capítulos anteriores dessa segunda parte do livro.

## 12.3 Session Bean Stateless

Um Session Bean Stateless não mantêm o estado para um cliente em particular. Quando invocamos um método, o estado de suas variáveis se mantêm apenas durante a invocação deste método. Quando o método é finalizado o estado não é retido. São componentes que não estão associados a um cliente específico e, portanto, implementam comportamentos que atendem a necessidade de muitos clientes.

**Session Bean Stateless: SBStatelessExampleBean.**

```
package com.book.example.ejb.session;
```

```
import javax.ejb.*;

// Exemplo de Session Bean que apresenta os serviços aos seus clientes.
// Estes serviços são implementados através de métodos de negócio.
// EJB do tipo Session Bean apresentam os métodos ejbCreate() e ejbRemove(), além
// dos métodos ejbActivate() e ejbPassivate() para Session Beans do tipo Stateful.
public class SBStatelessExampleBean implements SessionBean {
    //Contexto do Session Bean.
    private SessionContext sessionContext;

    // Executado pelo container após a criação da instância do EJB.
    // @throws CreateException exceção na criação de uma entidade do EJB.
    public void ejbCreate() throws CreateException {
    }

    // Utilizado pelo container para destruir a instância do EJB.
    // Para o EJB Session Bean Stateless, deve-se liberar os recursos alocados
    // para este EJB neste método.
    public void ejbRemove() {
    }

    // Utilizado pelo container para ativar o objeto do pool de EJBs.
    // Neste momento devem ser recuperados todos os recursos utilizados pelo EJB.
    // Isto é feito somente para Session Bean do tipo Stateful.
    // Este método deve ser definido para o EJB Session Bean Stateless também,
    // apesar de não ser utilizado, pois este tipo de EJB não fica passivo.
    public void ejbActivate() {
    }

    // Utilizado pelo container para devolver o objeto ao pool de EJBs.
    // Neste método devem ser liberados todos os recursos alocados pelo EJB.
```

```
    // Isto é feito somente para Session Bean do tipo Stateful.
    // Este método deve ser definido para o EJB Session Bean
Stateless também,
    // apesar de não ser utilizado, pois este tipo de EJB não fica
passivo.
    public void ejbPassivate() {
    }

    // Configura o contexto do Session Bean.
    // @param sessionContext contexto do SB.
    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }

  // ... abaixo defina os métodos de negócio, isto é, os serviços
  // que o Session Bean irá fornecer aos seus clientes.
```

## 12.4 Ciclo de vida - Session Bean Stateless

A figura 12.2 representa o ciclo de vida de um Session Bean Stateless.

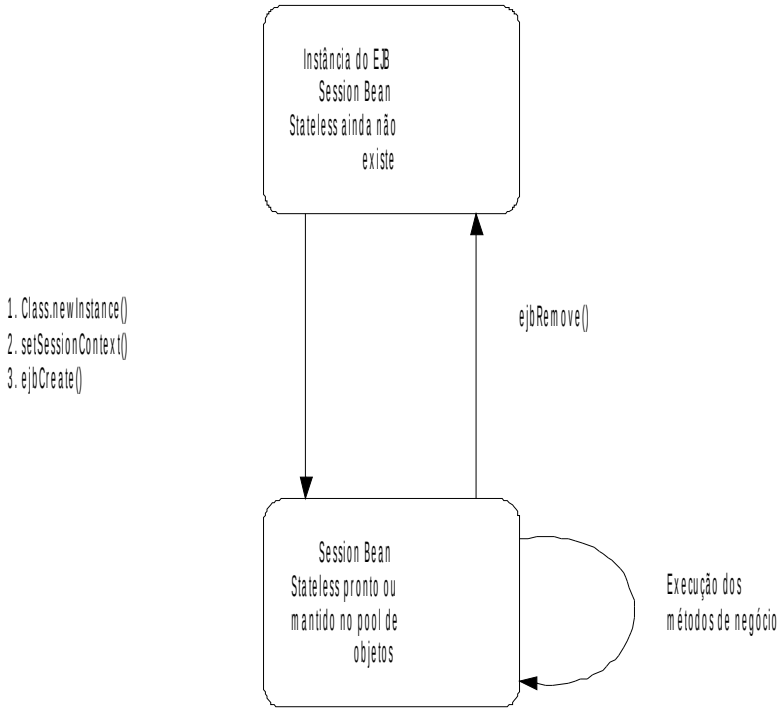


***Figura 12.2 – Ciclo de vida de um Session Bean Stateless.***

Assim que o servidor de aplicações é inicializado, não existem beans instanciados, então, dependendo das políticas de pooling adotadas, o container instancia o número definido de beans. Caso o container decida que precisa de mais instâncias no pool, instancia outros beans. Os beans no pool devem ser equivalentes (porque são Stateless), pois eles podem ser reutilizados por diferentes clientes. Quando o container decide que não precisa mais de instancias no pool, ele as remove.

# Session Bean Stateful

EJB Session Bean Stateful são componentes que mantêm o estado dos seus atributos e um relacionamento forte com o cliente que o utiliza. Se a execução termina ou se o cliente solicita a remoção da instância deste EJB para o container, a sessão é finalizada e o estado é perdido, isto

é, o valor dos atributos configurados pelo cliente para este EJB são perdidos e em uma próxima utilização estarão com seus valores nulos.

Este tipo de Session Bean, diferente do Session Bean Stateless, mantém os valores dos atributos entre várias chamadas aos seus métodos de negócio (ou serviços), sendo assim, o cliente pode configurar os valores dos atributos do Bean através dos métodos setters e assim o EJB pode utilizar estes valores para os métodos de negócio.

**Session Bean Stateful : SBStatefulExampleBean.**

```
package com.book.example.ejb.session;

import javax.ejb.*;

// Exemplo de Session Bean que apresenta os serviços aos seus clientes.
// Estes serviços são implementados através de métodos de negócio.
// EJB do tipo Session Bean apresentam os métodos ejbCreate() e ejbRemove(),
// além dos métodos ejbActivate() e ejbPassivate() para Session Beans
// do tipo Stateful.
public class SBStatefulExampleBean implements SessionBean {
    // Contexto do Session Bean.
    private SessionContext sessionContext;

    // ... defina os atributos que serão mantidos enquanto
    // a sessão de um cliente específico estiver aberta
    private String name;

    // Executado pelo container após a criação da instância do EJB.
    // @throws CreateException exceção na criação de uma entidade do EJB.
    public void ejbCreate() throws CreateException {
    }

    // Utilizado pelo container para destruir a instância do EJB.
    public void ejbRemove() {
```

```
    }

    // Utilizado pelo container para ativar o objeto do pool de EJBs.
    // Neste momento devem ser recuperados todos os recursos
utilizados pelo EJB.
    // Isto é feito somente para Session Bean do tipo Stateful.
    public void ejbActivate() {
    }

    // Utilizado pelo container para devolver o objeto ao pool de
EJBs.
    // Neste método devem ser liberados todos os recursos alocados
pelo EJB.
    // Isto é feito somente para Session Bean do tipo Stateful.
    public void ejbPassivate() {
    }

    // Configura o contexto do Session Bean.
    // @param sessionContext contexto do SB.
    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }

    // Configura o valor atributo name.
    // @param name valor do atributo a ser configurado.
    public void setName(String name){
        this.name = name;
    }

    // Obtém o valor do atributo name.
    // @return valor do atributo name.
    public String getName(){
        return name;
    }

    // ... abaixo defina os métodos de negócio, isto é, os serviços
    // que o Session Bean irá fornecer aos seus clientes.
}
```

# Ciclo de vida - Session Bean Stateful

O diagrama a seguir, figura 12.3, representa o ciclo de vida de um Session Bean Stateful. Temos as transições entre estado ativo e passivo, nos quais o bean deixa de ser utilizado por um tempo e fica aguardando novas chamadas do cliente quando esta passivo, e ativo quando volta a ser utilizado pelo cliente. Nestes dois momentos, pode-se liberar os recursos alocados para determinado bean que se tornará passivo, e obter estes recursos quando o bean se tornar ativo.

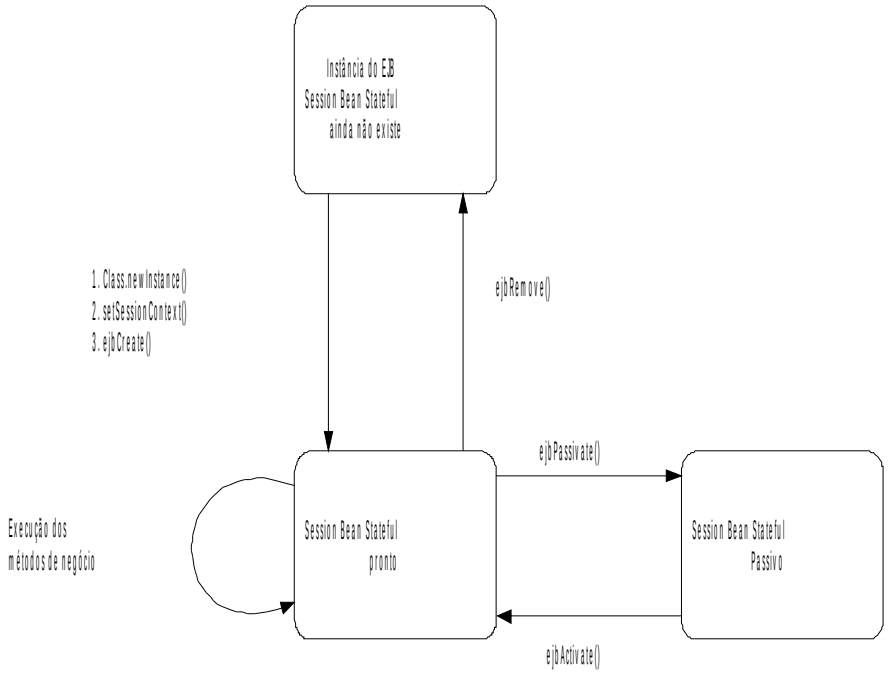


***Figura 12.3 – Ciclo de vida de um Session Bean Stateful.***

# Capítulo 13
# Entity Beans

## 13.1 O que são Entity Beans?

Entity Bean são Beans de Entidade, isto é, representam entidades persistentes. Em outras palavras, são componentes de negócio com mecanismo de persistência de dados.

O estado do Entity Bean pode ser persistido em um banco de dados relacional, arquivo XML além de outros tipos de repositórios de dados. Isto quer dizer que o estado do Entity Bean é mantido além do tempo de vida da aplicação ou do servidor J2EE. Esta é uma característica muito útil em situações onde deseja-se utilizar os dados desta entidade em momentos que seria inviável mante-los em memória.

Existem dois tipos de Entity Beans:

Bean-Managed Persistence.

Container-Managed Persistence.

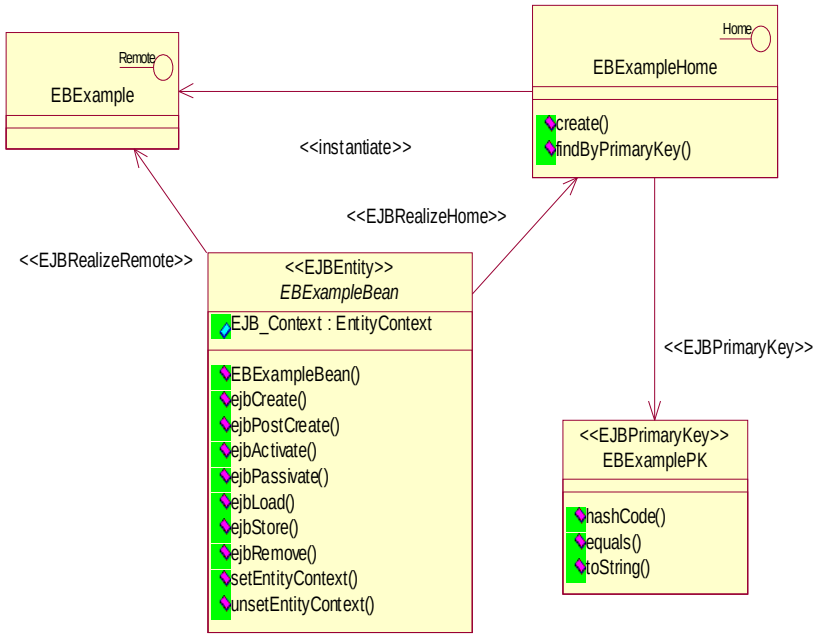


***Figura 13.1 – Diagrama de Classes UML do Entity Bean.***

# 13.2 Quando usar um Entity Bean?

Deve-se utilizar o EJB do tipo Entity Bean quando seu estado precisa ser persistido. Se a instância do EJB não estiver ativa ou se o servidor de aplicações for derrubado, pode-se recuperar o estado do mesmo, pois estará persistido na base de dados. O que torna os Entity Beans diferentes dos Sessions Beans é que primeiramente o estado do Entity Bean é salvo através de um mecanismo de persistência, além disso possui uma chave primária que o identifica.

# 13.3 Entity Bean Bean-Managed-Persistence

Utilizando esta estratégia, a codificação das chamadas de acesso à base de dados estão na classe de negócios do EJB e são responsabilidade do desenvolvedor. Então, para os métodos de busca (métodos utilizados para encontrar Entity Beans na base de dados) e para os métodos de criação, remoção e atualização dos Entity Beans, deve-se codificar os comandos responsáveis por realizar estas operações.

O Entity Bean BMP - Bean de Entidade com Persistência Gerenciada pelo Bean - oferece ao desenvolvedor a flexibilidade de desenvolver as operações de persistência de dados que em alguns casos, podem ser complexas de serem implementadas pelo container.

Esta estratégia demanda mais tempo de desenvolvimento e alguns autores sugerem que se utilize a estratégia de persistência CMP (Container-Managed-Persistence), que será vista a seguir.

Detalhes de implementação de um EJB Entity Bean BMP, assim como os deployment descriptors serão vistos no *Apêndice A*.

### Entity Bean BMP: EBBMPExampleBean.

```
package com.book.example.ejb.entity;

import javax.ejb.*;

// Exemplo de Entity Bean BMP utilizado para mantêr os dados em um meio
// persistente e apresentar as operações sobre os atributos deste EJB.
// Observe que no Entity Bean BMP, todo o código de criação (persistência do
// objeto), remoção, atualização etc. deve ser implementado nestes métodos,
// ficando a cargo do programador definir a melhor forma de fazer isso.
public class EBBMPExampleBean implements EntityBean {

    // Contexto do Entity Bean.
    EntityContext entityContext;
```

```
    // Atributo 1 a ser persistido.
    java.lang.String field1;

    // Atributo 2 a ser persistido.
    java.lang.String field2;

    // Cria uma instância do objeto em memória e persiste seus
dados.
    // @param field1 campo a ser persistido.
    // @param field1 campo a ser persistido.
    // @return chave única que identifica o objeto persistido. pode
ser null.
    // @throws CreateException exceção na criação do objeto.
    public java.lang.String ejbCreate(java.lang.String field1,
            java.lang.String field2) throws CreateException {

        // deve conter o código de persistência dos atributos do EJB.
        setField1(field1);
        setField2(field2);
        return null;
    }

    // Executado pelo container após a criação do EJB.
    // @param untitledField1 campo persistido.
    // @throws CreateException exceção na criação do objeto.
    public void ejbPostCreate(java.lang.String field1, java.lang.String
field2)
            throws CreateException {
    }

    // Executado pelo container para remover o objeto persistido.
    // @throws RemoveException
    public void ejbRemove() throws RemoveException {
        // Deve conter o código de remoção do EJB no meio de
persistência.
    }

    // Configura o valor do atributo field1.
    // @param field1 valor do atributo a ser configurado.
    public void setField1(java.lang.String field1) {
```

```
        this.field1 = field1;
    }

    // Configura o valor do atributo field2.
    //  @param field2 valor do atributo a ser configurado.
    public void setField2(java.lang.String field2) {
        this.field2 = field2;
    }

    // Obtém o valor do atributo field1.
    // @return valor do atributo field1.
    public java.lang.String getField1() {
        return field1;
    }

    // Obtém o valor do atributo field2.
    //  @return valor do atributo field2.
    public java.lang.String getField2() {
        return field2;
    }

    // Implementação do método de seleção do objeto pela sua
chave- primária.
    // @param field1 chave- primária do EJB.
    // @return chave primária do EJB.
    // @throws FinderException erro ao localizar o EJB.
    public java.lang.String ejbFindByPrimaryKey(java.lang.String field1)
            throws FinderException {

        // Deve conter a implementação da operação de localização
        // do objeto no meio persistente.
        return null;
    }

    // Utilizado para carregar o objeto persistente do meio de
persistência
    //  e atualizar os dados de sua instância.
    public void ejbLoad() {
    }
```

```
    // Utilizado pelo container para atualizar os dados do objeto
    //  no meio de persistência.
    public void ejbStore() {
    }

    // Utilizado pelo container quando ativa o objeto do pool.
    public void ejbActivate() {
    }

    // Utilizado pelo container quando devolve o objeto ao pool.
    public void ejbPassivate() {
    }

    // Desconfigura o contexto do Entity Bean.
    public void unsetEntityContext() {
        this.entityContext = null;
    }

    // Configura o contexto do Entity Bean.
    // @param entityContext contexto do Entity Bean.
    public void setEntityContext(EntityContext entityContext) {
        this.entityContext = entityContext;
    }
}
```

## 13.4 Ciclo de vida – Entity Bean BMP

A figura 13.2, representa o ciclo de vida dos Entity Beans BMP. Todos os métodos são chamados pelo container para o bean. Para criar um novo Entity Bean é utilizado o método create e para removê-lo é necessário executar o método remove. Pode-se carregar um Entity Bean usando os métodos de busca do entity bean (finders, ejbFind methods). Quando o bean é ativado, ele carrega o dado do meio de persistência e quando é desativado, persiste o dado no meio de persistência.

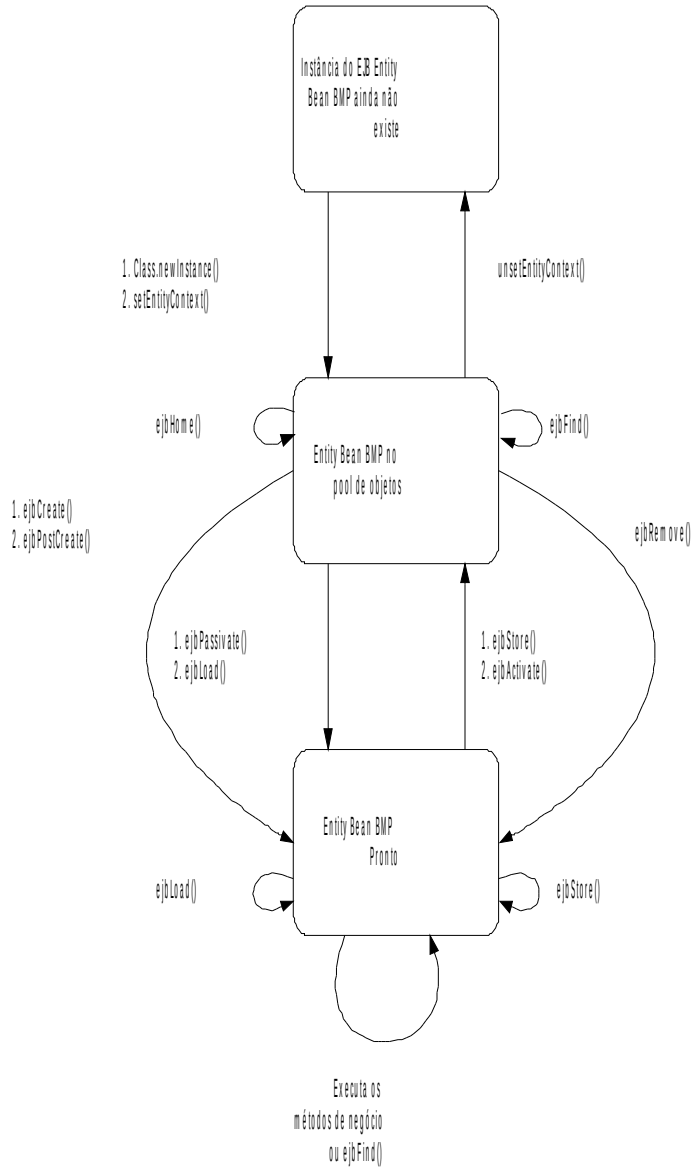


***Figura 13.2 – Ciclo de vida BMP Entity Bean.***

# 13.5 Entity Bean Container-Managed- Persistence

Entity Beans CMP (Bean de Entidade com Persistência Gerenciada pelo Container) oferecem mais rapidez e facilidade no desenvolvimento de objetos persistentes pois o desenvolvedor não precisa escrever os comandos para persistir e manipular os dados. O container se encarrega de realizar estas operações e retornar os valores desejados. Ele faz isso para as operações triviais tais como inserção do objeto, remoção, atualização e seleção pela chave- primária (create, remove, store e findByPrimaryKey respectivamente).

Para as operações específicas de consulta a objetos persistidos, se faz necessário o uso de uma linguagem de consulta conhecida como EQL (EJB Query Language). Os detalhes desta linguagem serão vistos a seguir.

Estas operações devem ser definidas no arquivo de deploy do componente (ejb-jar.xml) utilizando-se de EQL para cada operação desejada. Neste arquivo define-se os campos que serão persistidos e os relacionamentos entre Entity Beans.

Detalhes de implementação de um EJB Entity Bean CMP, assim como os deployment descriptors serão vistos no *Apêndice A*.

### Entity Bean CMP: EBCMPExampleBean.

```
package com.book.example.ejb.entity;

import javax.ejb.*;

// Exemplo de Entity Bean utilizado para mantêr os dados em um
meio persistente
// e apresentar as operações sobre os atributos deste EJB.
// No caso do uso de Entity Bean CMP, as operações de persistência
são
// implementadas pelo próprio container, sendo que o programador
não precisa
// se precocupar com estas operações triviais.
// Para as operações específicas, como uma consulta com outros
parâmetros por
```

```
// exemplo, esta pode ser definida através de EQL no deployment descriptor do
// Entity Bean CMP.
abstract public class EBCMPExampleBean implements EntityBean {

    // Contexto do Entity Bean.
    private EntityContext entityContext;

    // Cria uma instância do objeto em memória e persiste seus dados.
    // @param field1 campo a ser persistido.
    // @param field1 campo a ser persistido.
    // @return chave única que identifica o objeto persistido. pode ser null.
    // @throws CreateException exceção na criação do objeto.
    public java.lang.String ejbCreate(java.lang.String field1,
            java.lang.String field2) throws CreateException {
        setField1(field1);
        setField2(field2);
        return null;
    }

    // Executado pelo container após a criação do EJB.
    // @param untitledField1 campo persistido.
    // @throws CreateException exceção na criação do objeto.
    public void ejbPostCreate(java.lang.String field1, java.lang.String field2)
            throws CreateException {
    }

    // Executado pelo container para remover o objeto persistido.
    // @throws RemoveException
    public void ejbRemove() throws RemoveException {
    }

    // Configura o valor do atributo field1.
    // Observe que na especificação EJB2.0, estes métodos são implementados
    // pelo próprio container.
    // @param field1 valor do atributo a ser configurado.
    public abstract void setField1(java.lang.String field1);
```

```
    // Configura o valor do atributo field2.
    // Observe que na especificação EJB2.0, estes métodos são
implementados
    // pelo próprio container.
    // @param field2 valor do atributo a ser configurado.
    public abstract void setField2(java.lang.String field2);

    // Obtém o valor do atributo field1.
    // @return valor do atributo field1.
    public abstract java.lang.String getField1();

    // Obtém o valor do atributo field2.
    // @return valor do atributo field2.
    public abstract java.lang.String getField2();

    // Utilizado para carregar o objeto persistente do meio de
persistência
    // e atualizar os dados de sua instância.
    public void ejbLoad() {
    }

    // Utilizado pelo container para atualizar os dados do objeto
    // no meio de persistência.
    public void ejbStore() {
    }

    // Utilizado pelo container quando ativa o objeto do pool.
    public void ejbActivate() {
    }

    // Utilizado pelo container quando devolve o objeto ao pool.
    public void ejbPassivate() {
    }

    // Desconfigura o contexto do Entity Bean.
    public void unsetEntityContext() {
        this.entityContext = null;
    }
```

```
    // Configura o contexto do Entity Bean.
    // @param entityContext contexto do Entity Bean.
    public void setEntityContext(EntityContext entityContext) {
        this.entityContext = entityContext;
    }

}
```

## 13.6 Ciclo de vida – Entity Bean CMP

A Figura 13.3, representa o ciclo de vida do Entity Bean CMP. Ele é praticamente o mesmo de um Entity Bean BMP, sendo que a única diferença é que pode-se chamar o método ejbSelect(), tanto nos beans que estão no pool quanto nos que estão ativos.

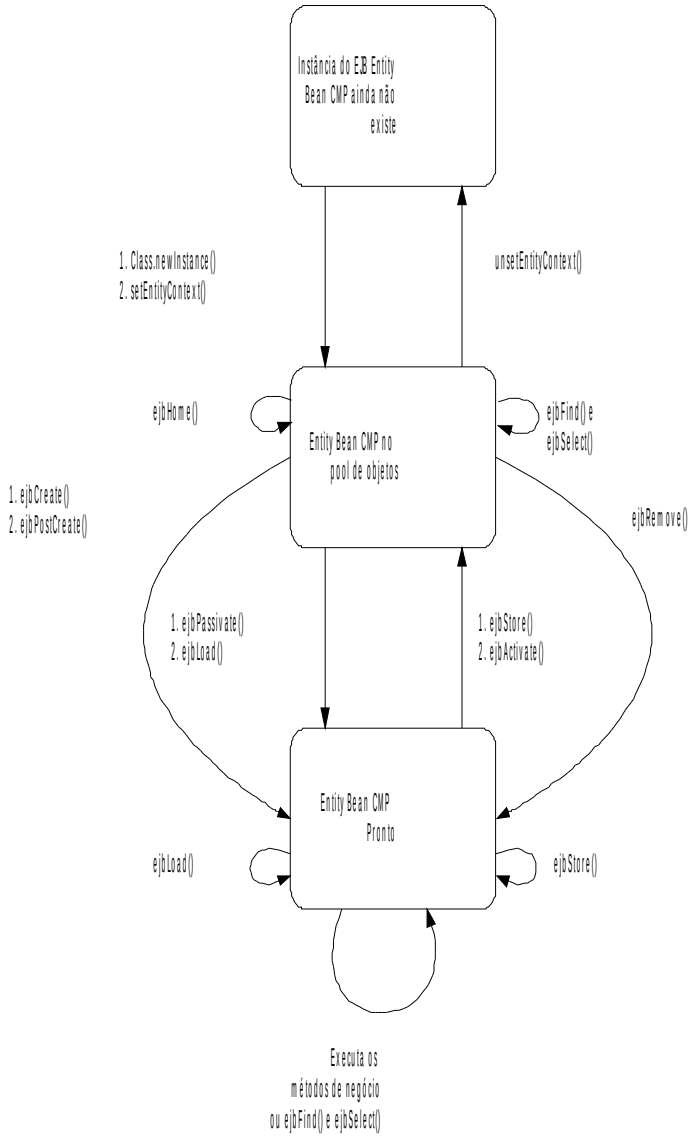


***Figura 13.3 – Ciclo de vida CMP Entity Bean.***

# 13.7 Relacionamento EJB Entity Bean CMP

## Relacionamentos entre entity beans

Um entity bean pode se relacionar com outro, como um relacionamento entre duas tabelas de um banco de dados relacional. A implementação de um relacionamento para entity beans BMP é feita por meio da codificação na classe de negócio do EJB, enquanto para os entity beans CMP o container se encarrega de oferecer os mecanismos de relacionamento através dos elementos de relacionamento definidos no deployment descriptor.

## Relationship fields – Campos de relacionamento de um CMP

Um relationship field é equivalente a uma chave estrangeira em uma tabela de um banco de dados relacional e identifica um entity bean através do relacionamento.

## Multiplicidade em relacionamentos entre entity beans CMP

Existem quatro tipos de multiplicidades:

**1 – 1:** Cada instância de um entity bean pode se relacionar com uma única instância de outro entity bean. Por exemplo, um entity bean marido tem um relacionamento **1 - 1** com o entity bean esposa, supondo que esta relação ocorra numa sociedade onde não é permitido a poligamia.

**1 – N:** Uma instância de um entity bean pode se relacionar com múltiplas instâncias de outro entity bean. O entity bean gerente, por exemplo, tem um relacionamento **1 - N** com o entity bean empregado, ou seja, um gerente pode ter vários empregados sob seu comando.

**N – 1:** Múltiplas instâncias de um entity bean podem se relacionar com apenas uma instância de outro entity bean. Este caso é o contrário do relacionamento **1 – N**,

ou seja, o entity bean empregado tem um relacionamento **N – 1** com o entity bean gerente.

**N – M:** Instâncias de entity beans podem se relacionar com múltiplas instâncias de outro entity bean. Por exemplo, em um colégio cada disciplina tem vários estudantes e todo estudante pode estar matriculado em várias disciplinas, então o relacionamento entre os entity beans disciplina e estudante é **N – M.**

## Direcionamento em relacionamentos entre entity beans CMP

Um relacionamento entre entity beans pode ser bidirecional ou unidirecional. Num relacionamento bidirecional cada entity bean tem um campo (relationship field) que referencia outro entity bean. Através deste campo um entity bean pode acessar o objeto relacionado.Em um relacionamento unidirecional, apenas um entity bean tem o campo referenciando outro entity bean.Consultas em EJB-QL podem navegar através destes relacionamentos. A direção dos relacionamentos determina quando a consulta pode navegar de um entity bean para outro.

Sendo um relacionamento bidirecional as consultas EJB-QL podem navegar nas duas direções acessando outros entity beans através dos campos de relacionamento. A tag cmr-field-name no arquivo ejb-jar-xml define o nome do campo do relacionamento e na classe abstrata do entity CMP teremos os respectivos métodos abstratos get e set relacionados a este campo. Os tipos que um método get como esse retorna é o tipo da interface local do objeto relecionado sendo um relacionamento **X - 1**, caso o relacionamento seja **X – N** o tipo retornado pode ser ou java.util.Collection ou java.util.Set.

Sempre que tivermos campos de relacionamento no entity bean não devemos invocar o seu método set no ejbCreate, ele deve ser invocado no método ejbPostCreate ou através dos métodos set e get que estiverem disponíveis na interface local. Exemplos:

Relacionamento bidirecinal:

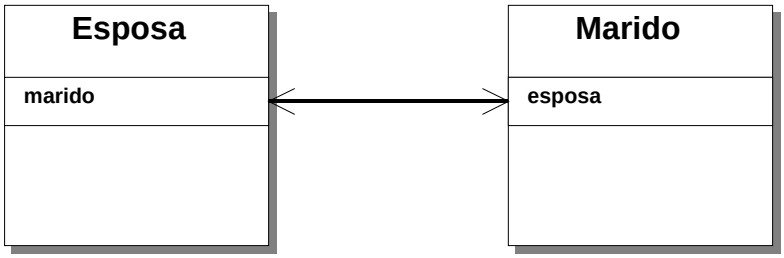


***Figura 13.4 – Relacionamento Bidirecional Entity Bean CMP.***

Neste caso podemos ver que os dois entity beans podem navegar e acessar as informações sobre o entity bean relacionado, só para visualizar iremos agora mostrar como ficariam as configurações no deployment descriptor.

```
<ejb-relation>
    <ejb-relation-name>esposa-marido</ejb-relation-name>
    <ejb-relationship-role>
        <description>esposa</description>
        <ejb-relationship-role-name>EsposaRelationshipRole</ejb-
relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
            <description>esposa</description>
            <ejb-name>Esposa</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <description>marido</description>
            <cmr-field-name>marido</cmr-field-name>
        </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
        <description>marido</description>
```

```
        <ejb- relationship- role- name>MaridoRelationshipRole</ejb-
relationship- role- name>
        <multiplicity>One</multiplicity>
        <relationship- role- source>
            <description>marido</description>
            <ejb- name>Marido</ejb- name>
        </relationship- role- source>
        <cmr- field>
            <description>esposa</description>
            <cmr- field- name>esposa</cmr- field- name>
        </cmr- field>
    </ejb- relationship- role>
</ejb- relation>
```

Na verdade não será precisa digitar todo esse código xml para criarmos os relacionamentos pois existem ferramentas que fazem isso por você, mas você deve ter idéia de como funciona e o que é configurado nesse arquivo.

A tag cmr- field- name configura o nome do campo do ejb nos quais serão criados os métodos get e set que estarão disponíveis na interface local, caso configuramos a multiplicidade 1 – 1, se fóssemos configurar 1 – N por exemplo deveríamos trocar a tag multiplicity para many e acrescentaríamos a tag cmr- field- type como o exemlo mostra a seguir:

```
<cmr- field>
    <description>esposa</description>
    <cmr- field- name>esposa</cmr- field- name>
    <cmr- field- type>java.util.Collection</cmr- field- type>
</cmr- field>
```

Neste caso você pode escolher se o tipo retornado será java.util.Collection ou java.util.Set.

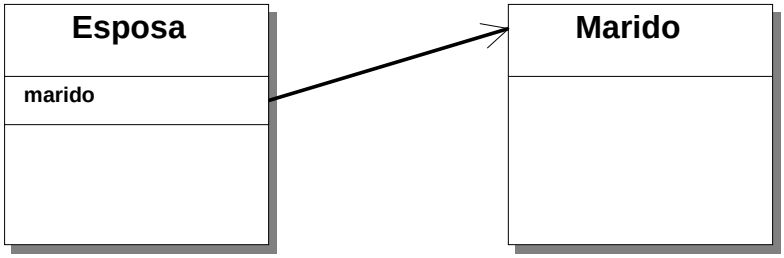


***Figura 13.5 – Relacionamento Unidirecional Entity Bean CMP.***

Aqui temos um relacionamento unidirecional onde apenas a esposa pode encontrar o marido, o marido não é capaz de recuperar informações das mulheres por meio do relacionamento. Neste caso a tag cmr-field no deployment descriptor ejb-jar.xml não estará configurada no lado em que definimos o relacionamento do lado do EJB marido.

```
<ejb-relation-name>esposa-marido</ejb-relation-name>
<ejb-relationship-role>
    <description>esposa</description>
    <ejb-relationship-role-name>EsposaRelationshipRole</ejb-
relationship-role-name>
    <multiplicity>One</multiplicity>
    <relationship-role-source>
        <description>esposa</description>
        <ejb-name>Esposa</ejb-name>
    </relationship-role-source>
    <cmr-field>
        <description>marido</description>
        <cmr-field-name>marido</cmr-field-name>
    </cmr-field>
</ejb-relationship-role>
<ejb-relationship-role>
    <description>marido</description>
    <ejb-relationship-role-name>MaridoRelationshipRole</ejb-
relationship-role-name>
    <multiplicity>One</multiplicity>
    <relationship-role-source>
        <description>marido</description>
        <ejb-name>Marido</ejb-name>
```

```
    </relationship-role-source>
  </ejb-relationship-role>
  </ejb-relation>
```

## 13.8 EJB-QL

EQL ou EJB-QL é a linguagem portável utilizada para construir os métodos de consulta dos EJB's CMP de acordo com a especificação 2.0. Esta linguagem não é utilizada para os EJB's BMP, pois os mesmos utilizam a API JDBC, ou outro mecanismo de persistência para acesso à base de dados.

### Como e quando utilizar EQL?

Devemos utilizar EQL para implementar os métodos de busca de seus Entity Beans CMP que não seja a partir de sua chave-primária (findByPrimaryKey), pois este já é oferecido automaticamente pelo container.

### EQL Queries

Um comando EJB-QL contém três partes:

Uma cláusula SELECT

Uma cláusula FROM

Uma cláusula opcional WHERE

### A cláusula FROM

A cláusula FROM ela define o domínio de uma consulta, indica qual parte de uma base de dados você irá consultar. No caso de uma base de dados relacional, a cláusula FROM tipicamente restringe quais tabelas serão consultadas. Por exemplo:

```
SELECT OBJECT(o)
FROM X AS o
```

Aqui estamos declarando uma variável na cláusula FROM. Esta variável o pode ser usada posteriormente em outras cláusulas desta mesma consulta, neste caso estamos reusando a variável na cláusula SELECT.

Algumas vezes precisamos declarar variáveis na cláusula FROM que representa um conjunto de valores:

```
SELECT OBJECT(a)
FROM X AS o, IN (o.items) a
```

A frase X AS o declara a varíavel o que representa os entity beans X, e a frase IN(o.items) a declara a variável a que representa uma coleção de items do entity Bean X. Então AS é usado quando a varável declarada representa um único valor e IN é usado quando a variável representa uma coleção de valores.

## A cláusula WHERE

A Cláusula WHERE restringe o resultado da consulta, você escolhe os valores desejados a partir das variáveis declaradas na cláusula FROM:

```
SELECT OBJECT(p)
FROM Pessoa p
WHERE p.nome = ?1
```

Esta consulta recupera todas as pessoas que possuem o atributo nome igual ao parâmetro que será passado no método. Por exemplo, o método poderia ser construído da seguinte forma:

```
findByNome(String nome)
```

Para ultizar collections na cláusula WHERE você precisa declará- la primeiramente na cláulua FROM:

```
SELECT OBJECT(a)
FROM Pessoa AS p, IN(p.parentes a)
WHERE a.filho.name = 'Pedro'
```

Algumas vezes você pode declarar mais de uma varíavel que representa o mesmo entity bean. Quando fazemos comparações este artifícil é muito útil:

```
SELECT OBJECT(o1)
FROM Pessoa o1, Pessoa o2
WHERE o1.idade > o2.idade AND o2.nome = 'João'
```

## A cláusula SELECT

A cláusula SELECT especifica o resultado retornado pela consulta. Por exemplo:

```
SELECT OBJECT(o)
FROM Pessoa p, IN(o.parentes) a
```

Na consulta são definidas duas variáveis p e a, a cláusula select determina qual deve ser selecionada.

Neste exemplo a consulta retorna todos os produtos em todas as regras que contem items:

```
SELECT l.produtos FROM REGRA as r, IN(r.items) l
```

Como você pode ver podemos utlizar o ponto para acessar relacionamentos na cláusula SELECT. Este código é interpretado para um SQL-padrão onde um JOIN é feito para recuperar os dados desejados.

Perceba que nest exemplo não utlizamos a notação Object (), ele é apenass utilizado quando se trata de uma variável simples que não faz uso do ponto para acessar informações por meio dos relacionamentos entre entity beans.

Outra informação importante é que a cláusula where não trabalha com collections, apenas aceita variáveis simples, ou seja:

```
SELECT r.items
FROM Regra AS r
```

Esta query não está correta, a forma correta é:

```
SELECT Object(a)
FROM Regra AS r, IN(r.items) a
```

Para filtrar as informações, ou seja, não trecuperar informações repetidas é utlizado o filtro DISTINCT:

```
SELECT DISTINCT l.produtos FROM REGRA AS r, IN(r.items)
```

Você pode também fazer que seu método de busca retorne um java.util.Set que não permite que valores repetidos sejam inseridos.

Agora a seguir temos um exemplo de como um método é definido no arquivo ejb-jar-xml pode ser acessado por meio das interfaces Home e/ou localHome do EJB. Na interface o método estaria assim:

```
public java.util.Collection findByNome(String nome) throws
FinderException,
        RemoteException ;
```

Este método vai retornar todas as pessoas que tem o nome passado como parâmetro. No arquivo ejb-jar-xml temos:

```
. . .
<query>
    <query-method>
        <method-name>findByNome</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        <![CDATA[SELECT OBJECT(a) FROM Pessoa AS a WHERE nome
= ?1]]>
    </ejb-ql>
<query>
. . .
```

# Capítulo 14

# Message- Driven Beans

## 14.1 O que são Message- Driven Beans?

São um tipo de componentes EJB que recebem mensagens por meio de um JMS (Java Message Service).

Enquanto tradicionalmente os outros Beans são acessados através de interfaces (RMI- IIOP) com acesso síncrono, os Message- Driven Beans, também conhecidos pela sigla MDB, são utilizados por meio de mensagens, com acesso assíncrono.

Isso se dá por meio de um middleware orientado a mensagens (MOM), localizado entre o cliente, Message Producer, e o bean, Message Consumer. Este middleware recebe mensagens de um ou mais clientes e envia as mensagens para os Beans destino.

JMS é a API utilizada para receber as mensagens dos clientes e enviar as mensagens aos Message- Driven Beans.

O estado do objeto é muito simples e semelhante a um EJB Session Bean Stateless que não mantêm o estado para um cliente em particular. Quando enviamos uma mensagem para invocamos um Message- Driven Bean, o estado de suas variáveis se mantêm apenas durante a invocação. Quando o processamento requerido é finalizado o estado não é retido. São componentes que não estão associados a

um cliente específico e, portanto, implementam comportamentos que atendem a necessidade de muitos cliente.

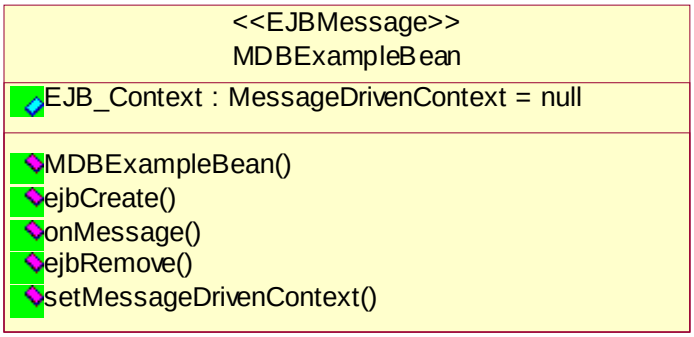


***Figura 14.1 – Diagrama de Classes UML do Message-Driven Bean***

## Message- Driven Bean : MDBExampleBean.

```
package com.book.example.ejb.mdb;

import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;

// Este EJB exemplo do tipo Message- Driven Bean, apresenta os
métodos- padrão
// de ciclo de vida ejbCreate() e ejbRemove() e o método onMessage()

// executado no recebimento de uma mensagem.
public class MDBExampleBean implements MessageDrivenBean,
MessageListener {

    // Contexto do MDB.
  private MessageDrivenContext messageDrivenContext;

    // Utilizado pelo container para instanciar o EJB.
    // @throws CreateException exceção na instanciação do objeto.
    public void ejbCreate() throws CreateException {
    }
```

```
    // Utilizado pelo container para remover o EJB.
    public void ejbRemove() {
    }

    // Executado no recebimento de uma mensagem para este MDB.
    // @param msg mensagem recebida através do MOM.
    public void onMessage(Message msg) {
        try {
            // imprime a mensagem recebida
            System.out.println("Mensagem recebida: " + ((TextMessage)
msg).getText());
        }catch (JMSException ex) {
            ex.printStackTrace();
            System.err.println("Erro ao obter o texto da mensagem: " +
ex);
        }
    }

    // Configura o contexto do MessageDriven Bean.
    // @param messageDrivenContext contexto do MDB.
    public void setMessageDrivenContext(MessageDrivenContext
messageDrivenContext) {
            this.messageDrivenContext = messageDrivenContext;
    }
}
```

## 14.2 Quando usar um Message-Driven Bean?

Deve-se utilizar EJB Message-Driven Bean quando necessita-se de operações assíncronas sendo executadas no servidor (container).

Observe que para realizarmos um processamento em paralelo no servidor sem o uso de Message-Driven Bean, era necessário o uso de Threads em EJB Session Bean Stateless.

Esta prática é desaconselhada pela própria Sun MicroSystems Inc., pois recai em vários problemas, um deles a falta de controle dessas execuções por parte do container, perdendo assim as facilidades e ganhos que o container oferece.

Para sanar este problema, foi criado o EJB Message- Drive Bean que pode ter várias instâncias de um mesmo componente sendo executados em paralelo, para realizar a mesma operação.

## 14.3 Ciclo de vida - Message- Driven Bean

A Figura 14.2, representa o ciclo de vida do Message- Driven Bean. Ele é bem simples comparado aos outros beans, pois o container instancia um determinado número de beans e os mantém no pool de acordo com as necessidades e removidas quando o container decidir, da mesma forma que o Session Bean Stateless. Assim que uma mensagem é recebida pelo JMS, ela é redirecionada para um bean específico, para que seja tratada.

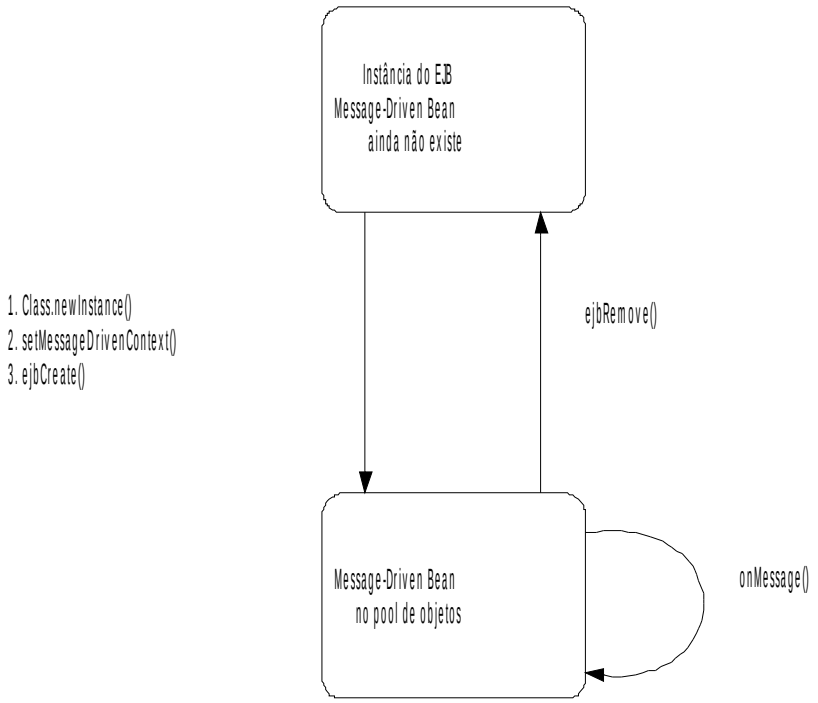


***Figura 14.2 – Ciclo de vida Message-Driven Bean.***

# 14.4 O que é e para que serve o JMS?

Antes de falarmos a respeito de JMS, devemos enteder o conceito de Messaginig (envio de mensagens). Messaging pode ser entendido pela troca de mensagens entre duas aplicações, programas ou componentes. Um cliente pode enviar mensagens para um ou mais receptores, que também pode retornar um outra mensagem, ou executar algum método de negócio. Utilizando este recurso, o receptor de mensagens não precisa estar disponível no mesmo momento que o cliente enviou a mensagem, podendo consumi-la, ou recebê-la posteriormente, isto é, no momento em que seu serviço estiver ativo. Assim, o cliente que envia a mensagem e o receptor da mensagem devem somente conhecer bem o formato da mensagem, com os dados que ela carrega.

Para fazer uso destes recursos na linguagem Java, foi criada uma API conhecida como JMS – Java Messaging Service. Esta API fornece serviços que permitem a criação, envio, recebimento e leitura de mensagens. O importante a saber é que um cliente cria uma mensagem e a envia utilizando esta API. Um receptor de mensagens, receberá esta mensagem através de um Message-Oriented-Middleware ou MOM. Alguns conceitos muitos usados nas referências a JMS são produtores e consumidores de mensagem (producers / consumers), para designar um cliente como um produtor de mensagens e um componente EJB Message-Driven Bean por exemplo, como um consumidor de mensagens.

O caminho percorrido pela mensagem é bem simples. Vamos ver isso na figura abaixo, na qual o produtor envia a mensagem, que é recepcionada pelo MOM e a seguir redirecionada para o consumidor correto. Observe na figura, que o produtor também pode ser um consumidor de mensagens e que o consumidor de mensagens também pode ser um produtor.

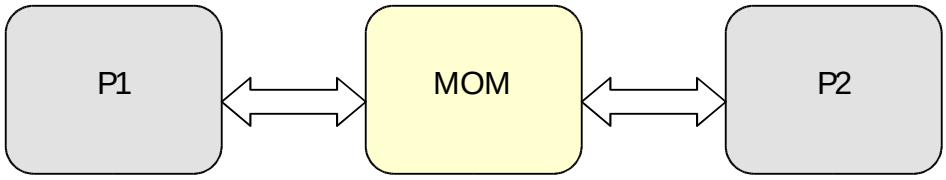


***Figura 14.3 – MOM.***

A arquitetura JMS é composta por cinco partes que são: Provedor JMS, que implementa as interfaces definidas na API JMS e provê recursos para administrar esse serviço; os clientes JMS que podem ser programas ou componentes que agem como produtores e consumidores de mensagens; as mensagens propriamente ditas que são objetos que transportam os dados do cliente para o receptor; os objetos de administração do serviço JMS que são utilizados pelos clientes para enviar as mensagens; e os clientes nativos que são clientes que usam produtos de Messaging nativos e não da API JMS.

A seguir vamos detalhar os dois tipos de postagem de mensagens e enteder as diferenças entre eles.

Point- to- Point

Publish/Subscribe.

## Mensagens Point- To- Point (Queue)

O conceito de mensagens Point- To- Point – PTP – é de enfileirar as mensagens para serem consumidas. Os produtores enviam as mensagens para uma determinada fila (Queue), que são consumidas por um destinatário. Assim que receber a mensagem, o destinatário avisa ao MOM que a mensagem foi recebida e processada corretamente (sinal de acknowledge). A fila armazena todas as mensagens que são enviadas, até o momento que são consumidas pelos receptores, ou até o momento que forem expiradas.

Observe ainda, que vários consumidores de mensagens (muitas instâncias do mesmo consumidor) podem consumir mensagens do MOM. Este recurso normalmente é disponibilizado e gerenciado pelo servidor de aplicações para processamento de mensagens em paralelo.

O cliente pode enviar mensagens para o MOM, mesmo sem existir nenhum consumidor de mensagens ativo naquele momento.

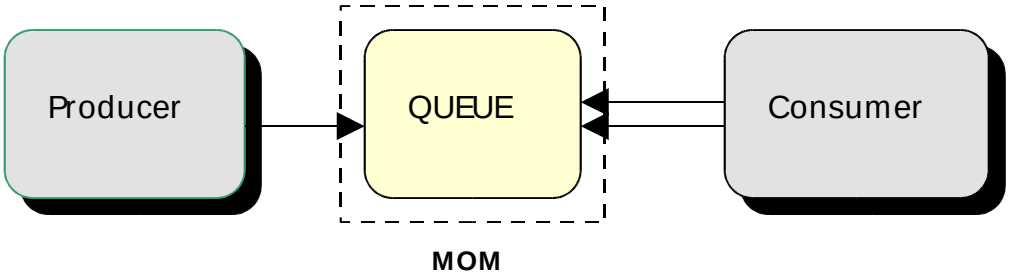


***Figura 14.4 – Point- To- Point (Queue).***

## Mensagens Publish/Subscribe (Topic)

No uso de publish/subscribe – Publica/Inscreve – os clientes enviam a mensagem para um tópico (topic). Os consumidores registram-se nos tópicos que lhes são convenientes e são notificados da chegada de uma nova mensagem.

Os consumidores podem somente consumir mensagens que forão postadas depois de terem se registrado (inscrito) no MOM, isto é, as mensagens enviadas pelos clientes antes disto, não podem ser consumidas por eles. Observe que neste caso, cada mensagem pode ter mais de um tipo de consumidor.

Se algum cliente emviar uma mensagem para um tópico que não possue nenhum consumidor registrado, esta mensagem não será entregue. Após o consumidor se registrar, as mensagens que chegam ao MOM são notificadas aos consumidores, que fornece estas mensagens a eles.

Por haver uma dependência muito grande do tempo em que o consumidor deve estar ativo enquanto o cliente envia a mensagem, os consumidores podem fazer registros duráveis (DURABLE) e receber mensagens enviadas enquanto eles não estam ativos. Isto permite a facilidade de uma fila, com a diferença de consumo de mensagens por diversos tipos de consumidores.

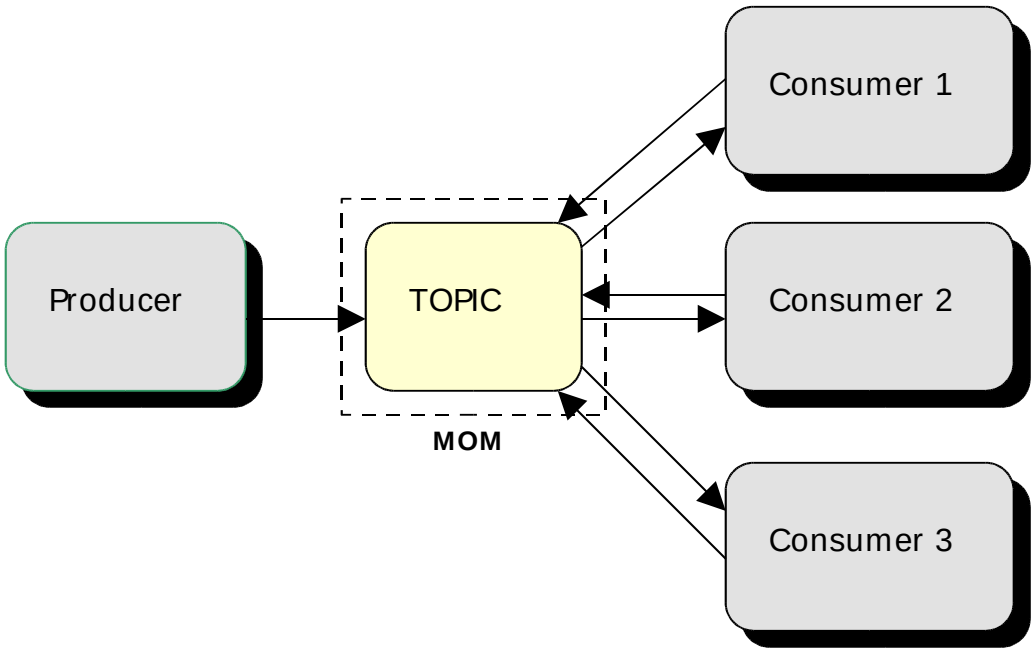


***Figura 14.5 – Publish/Subscribe (Topic).***

## Exemplo

A seguir apresentamos um exemplo simples do envio de mensagens por um cliente TextClient e o recebimento das mensagens por um Message- Driven Bean TestMDB.

### Teste de envio de mensagens: TestClient

```
import javax.jms.*;
import javax.naming.*;

// Esta classe exemplo, envia mensagens para um fila (queue)
// para ser consumida por um MDB.
public class TestClient {

    public static void main(String[] args) {
        QueueConnection queueConnection = null;
        try {
```

```
            // cria o contexto
            Context ctx = new InitialContext();

            // localiza a connection factory
            QueueConnectionFactory queueConnectionFactory =
                (QueueConnectionFactory) ctx.lookup
("java:comp/env/jms/
                MyQueueConnectionFactory");

            // localiza a queue (fila)
            Queue queue = (Queue) ctx.lookup
("java:comp/env/jms/QueueName");

            // cria a conexão
            queueConnection =
queueConnectionFactory.createQueueConnection();

            // cria a sessão da conexão
            QueueSession queueSession =
queueConnection.createQueueSession(false,
                Session.AUTO_ACKNOWLEDGE);

            // cria o sender
            QueueSender queueSender = queueSession.createSender
(queue);

            // cria a mensagem e as envia
            TextMessage message = queueSession.createTextMessage();
            for (int i = 0; i < 3; i++) {
                message.setText("Mensagem nr: " + (i+1));
                System.out.println(message.getText());
                queueSender.send(message);
            }
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        } finally {
            // fecha a conexão
            if (queueConnection != null) {
                try {
                    queueConnection.close();
```

```
                } catch (JMSException e) {}
            }
            System.exit(0);
        }
    }
}
```

## Message- Driven Bean : TestMDB

```
import javax.ejb.*;
import javax.naming.*;
import javax.jms.*;

// Teste de recebimento de mensagens pelo MDB.
// Recebe uma mensagem do tipo <code>TextMessage</code> e a
imprime no console.
public class TestMDB implements MessageDrivenBean,
MessageListener {

    // Contexto do Message- Drive Bean.
    private transient MessageDrivenContext ctx = null;

    // Instância o objeto no servidor pelo container.
    public void ejbCreate() {
    }

    // Remove a instância do objeto no servidor.
    public void ejbRemove() {
    }

    // Configura o contexto do EJB MDB.
    // @param ctx contexto do MDB.
    public void setMessageDrivenContext(MessageDrivenContext ctx)
{
        this.ctx = ctx;
    }

    // Valida a mensagem recebida e a imprime no console.
    // @param message mensagem a ser notificada.
    public void onMessage(Message message) {
        try {
```

```
            if (message instanceof TextMessage) {
                System.out.println("Mensagem recebida pelo MDB = "
                    + ((TextMessage) message).getText());
            }
            else {
                System.err.println("Mensagem com tipo errado : "
                    + message.getClass().getName());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

# Capítulo 15

# Transações e Segurança

A proposta inicial da API desenvolvida para a plataforma Java é a de prover aos desenvolvedores os recursos básicos de infra- estrutura e permitir que eles pudessem se concentrar na implementação das regras de negócio de suas aplicações. Assim, foram disponibilizadas bibliotecas para trabalhar com coleções, com entrada e saída de dados, com recursos de internacionalização, entre outras.

Esta idéia foi utilizada novamente na especificação da plataforma J2EE, com o objetivo de prover implementações para problemas recorrentes de aplicações distribuídas, agilizando seus tempos de desenvolvimento e permitindo que o foco do desenvolvimento fique nas regras de negócio a serem atendidas, deixando os recursos de infra-estrutura complexos a cargo dos fabricantes dos containers. Desta forma, há um aumento significativo nas chances das necessidades dos clientes serem atendidas e, por consequência, os sistemas construídos serem bem sucedidos.

Entre a gama de serviços previstos na especificação da plataforma J2EE ligados aos Enterprise JavaBeans, serão abordados dois dos mais usados: transações e segurança. O primeiro serviço serve para garantir a integridade dos dados, permitindo que falhas e execuções concorrentes não tornem as informações gerenciadas pela aplicação

inconsistentes. O segundo serviço trata da proteção da aplicação com relação ao acesso não autorizado às funcionalidades disponibilizadas por ela.

## 15.1 Transações

Tipicamente os sistemas provêem funcionalidades para seus clientes baseados em uma massa de dados armazenados em um meio peristente. A integridade das informações armazenadas neste repositório é fundamental para o correto funcinamento do sistema e muitas vezes a perda de informações pode causar prejuísos para seus usuários.

Os problemas relacionados a manutenção da integridade das informações armazenadas em um meio persistente podem ser causados por fatores físicos ou lógicos. Dentre os fatores físicos podemos destacar a falta de energia elétrica durante o processamento de uma funcionalidade que envolva várias operações sobre os dados. Já fatores lógicos estão relacionados com erros na programação do acesso concorrente aos dados, onde alterações realizadas por um processo podem ser sobrepostas por outros processos executando de forma concorrente.

Estes problemas são significativamente agravados quando estamos desenvolvendo em um ambiente distribuído. Uma vez que partes de nossas aplicações podem ser executadas em máquinas diferentes, há a chance de uma delas quebrar sem que o restante da aplicação tenha conhecimento disto, fazendo com que uma operação realizada pela aplicação seja executada parcialmente, o que pode corromper os dados sendo manipulados. Também há a chance de uma infomação ser alterada por uma parte da aplicação sem que a outra seja notificada, o que também ocasiona resultados errôneos.

Como solução a estes problemas, foi desenvolvido o conceito de transações. Para tanto é feito um agrupamento das operações realizadas em uma determinada porção do software em uma unidade denominada transação, que

apresenta quatro propriedades fundamentais, chamdas de ACID:

Atomicidade - garante a completude da execução das operações de uma transação, ou seja, ou todas as operações de uma transação são executadas ou nenhuma operação é realizada. Caso não seja possível completar uma operação após outras terem sido executadas, deve ser possível anular o efeito destas últimas para atender a esta propriedade.

Consistência - garante que o conjunto de operações que compõem uma transação nunca deixem o sistema em um estado inconsistente.

Isolamento – garante que a execução de uma transação não seja afetada pela execução de outra transação. Esta propriedade é especialmente importante quando há aplicações concorrentes que acessam os mesmos dados, uma vez que durante a execução de uma transação os dados podem ficar temporariamente inconsistentes, levando outras transações que utilizam estes dados a produzirem resultados incorretos e possivelmente violarem a propriedade de consistência.

Durabilidade - garante que os resultados obtidos em uma transação sejam armazenados em um meio persistente.

Para ilustrar estas propriedades, vamos utilizar como exemplo uma transação bancária de transferência de fundos de uma conta corrente para outra. Suponha a existência da seguinte classe responsável por realizar a transferência:

**Classe usada para demonstar propriedades transacionais.**

```
public class Banco  {
    // ...
    public void transferenciaEntreCC(ContaCorrente origem,
ContaCorrente destino,
            double montante) {
        // verifica se há saldo suficiente na conta a ser debitada
```

```
        if (origem.getSaldo() < montante)
            throw new SaldoInsuficienteException("A conta " + origem
+ " deve ter
                    saldo igual ou superior a " + montante);
        origem.debitar(montante);
        destino.creditar(montante);
    }
    // ...
}
```

A necessidade da primeira propriedade pode ser logo vista nas duas últimas operações do método de transferência entre contas correntes. Caso logo após a primeira operação ser processada ocorra uma queda do sistema, por falta de energia elétrica, por exemplo, o montante será reduzido da primeira conta mas não será creditado na segunda. Isto significa que o montante iria simplesmente desaparecer, tornando a base de dados do sistema bancário inconsistente.

A segunda propriedade diz respeito à demarcação dos limites da transação. Ela garante que uma transação conterá um método de negócio completo. Assim, no caso do método de transferência entre contas correntes, não seria possível inclir a operação de débito em uma transação e a de crédito em outra, pois neste caso ao final da primeira transação o sistema estaria inconsistente.

O isolamento diz respeito a tornar a execução em paralelo de transações independentes. Como a consistência só é garantida no término de uma transação, é possível que durante sua execução o sistema esteja num estado inconsistente (no exemplo esta situação seria atingida no momento após a operação de débito e antes da operação de crétido). Assim, caso uma outra transação seja iniciada enquanto uma primeira ainda não terminou, é possível que seus resultados não sejam corretos, uma vez que a premissa da consistência é que o sistema estaria inicialmente consistente.

A última propriedade trata da confirmação dos resultados efetuados pelas transações. Neste caso, após a confirmação das operações, seus resultados são

efetivamente armazenados em meios persistentes. Assim, é garantido que em caso de falhas no sistema após o encerramento da transação, seus resultados ainda poderão ser vistos após o restabelecimento do sistema.

## Delimitação das Transações

O uso de transações traz muitas vantagens e garante a integridade dos dados, mas há um custo alto em utilizá-las. Os controles que são utilizados para garantir as propriedades ACID podem tornar uma aplicação lenta de tal forma que se torne sua utilização inviável. Assim, este recurso deve ser usado de forma consciente, ficando a cargo dos desenvolvedores determinar as operações de negócio que necessitam de transações e as que não. Ele também deve determinar a melhor forma de agrupar as operações em transações.

Na especificação J2EE há duas formas do desenvolvedor informar aos containers EJB quais são as operações transacionais e como elas devem se comportar com relação às outras: uma forma programática e outra declarativa. A primeira forma consiste na inclusão no código fonte dos EJBs instruções para que as transações sejam iniciadas e terminadas. Uma vez que o controle todo de quando as transações iniciam e são concluídas faz parte da especificação do bean, este tipo de delimitação é chamada de BMT (Bean Managed Transaction).

A forma declarativa de determinar os limites das transações é feita através de instruções nos arquivos descritores dos EJBs. Desta forma, o próprio container fica a par da existência das transações no momento do deploy e é capaz de controlar sua execução. Este tipo de transação é chamado de CMT (Container Managed Transaction) e permite um elevado grau de flexibilidade para mudanças na política de controle de transações da aplicação. Os dois tipos de transações serão discutidos com detalhes nas próximas seções.

## Transações BMT

Transações BMT (Bean Managed Transactions) ou programadas permitem maior controle por parte do desenvolvedor, pois ele é que insere em seu código instruções que indicam o momento em que a transação foi iniciada, o momento em que ela foi concluída com sucesso ou que ocorreu um erro e a transação deve ser abortada.

As instruções para o controle das transações são disponibilizadas numa API chamada JTA (Java Transaction API). Ela permite o acesso transparente aos mais variados gerenciadores de transações disponíveis nos containers. A implementação da Sun para tal API é o JTS (Java Transaction Service).

A comunicação com um gerenciador de transações se inicia com a obtenção de uma referência para a interface javax.transaction.UserTransaction, que é feita através do contexto do EJB. Esta interface permite enviar comandos para delimitar o início de uma transação (begin), conclusão com sucesso (commit) ou solicitar que as operações realizadas sejam descartadas pela ocorrência de falhas (rollback). Abaixo um exemplo de uso de uma transação gerenciada pelo bean em um método que realiza a transferência de um montante de uma conta corrente para outra.

### Exemplo do uso de JTA

```
public void transferenciaEntreContasCorrentes(ContaCorrente
origem,
        ContaCorrente destino, double montante) {
    javax.transaction.UserTransaction ut =
sessionContext.getUserTransaction();
    try {
        ut.begin();
        origem.debitar(montante);
        destino.creditar(montante);
        ut.commit();
    }catch (Exception ex) {
        try {
            ut.rollback();
```

```
        } catch (Exception rbex) {
            rbex.printStackTrace();
        }
        throw new RuntimeException(ex);
    }
}
```

Este tipo de transações pode ser utilizado apenas Session Beans e Message Driven Beans. Os Entity Beans exigem um maior controle do container sobre suas ações e portanto não tem disponível o gerenciamento de transações feito por ele próprio.

## Transações CMT

Transações CMT (Container Managed Transactions) ou declarativas podem ser utilizadas com qualquer tipo de EJB. Neste tipo de transação não há a necessidade de programação explícita das delimitações das transações, esta tarefa é efetuada automaticamente pelo próprio container. Para tanto, é necessário informar nos descritores dos EJBs a necessidade de suporte transacional às operações e como ele deve gerenciá-lo.

Uma vez que não há a necessidade de inclusão de cógido específico para gerência de transações, a flexibilidade de mudança da estratégia de emprego deste recurso é muito maior, pois basta alterar os descritores dos EJBs para atingir este objetivo.

Abaixo está um exemplo de descritor que define a inclusão de suporte transacional num EJB pra gerenciamento de contas bancárias:

**Exemplo de descritor - ejb-jar.xml**:

```
.....
<assembly-descriptor>
    <container-transaction>
        <method>
            <ejb-name>Banco</ejb-name>
            <method-name>*</method-name>
```

```
        </method>
        <trans- attribute>Required</trans- attribute>
    </container- transaction>

    <container- transaction>
        <method>
            <ejb- name>Banco</ejb- name>
            <method- name>transferenciaEntreCC</method- name>
        </method>
        <trans- attribute>Mandatory</trans- attribute>
    </container- transaction>
</assembly- descriptor>
.....
```

O suporte a transações é definido dentro do assembly-descriptor. Nele são definidos, para cada método do EJB, qual o seu atributo transacional. Observando o exemplo, verifica-se que para o EJB chamado Banco são definidas dois blocos de transações. No primeiro é definido o atributo Required para todos os seus métodos através do uso do wildcard *. Logo em seguida este atributo é sobreposto para o método transferenciaEntreCC, modificando o atributo para Mandatory.

O atributo transacional, citado acima, é usado para dizer ao container como as transações devem ser efetuadas quando os métodos especificados são chamados. Deve-se levar em conta que o cliente do EJB, que pode uma aplicação Swing, um Servlet ou mesmo outro EJB, também pode estar usando transações para efetuar suas tarefas e a chamada a um método do EJB deve levar isto em conta. Tendo isto em mente, a segir serão mostrados os possíveis atributos transacionais definidos na especificação EJB2.0 e o comportamento do container para cada um deles.

**Required** (Requerido): Configura o bean (ou método) para sempre executar em uma transação. Se a chamada for feita dentro de uma transação no cliente, a chamada de método passará a fazer parte desta transação. Caso contrário, uma nova transação é criada.

**RequiresNew** (Requer novo): Utilizada para que o método do EJB sempre execute dentro de uma transação nova. Assim, o método executará dentro de uma transação própria que será encerrada quando o método termina sua execução. Caso o método seja chamado dentro de uma transação do cliente, esta é suspensa, é criada uma nova transação para o EJB, executada e finalizada, e só então a transação do Cliente é retomada.

**Mandatory** (Mandatório): Indica que o método somente pode ser chamado dentro de uma transação do cliente. Diferente do Required, que caso o cliente não esteja numa transação cria uma nova, este atributo gera uma exceção se sua chamada não for dentro de uma transação.

**NotSupported** (Não Suportado): Usado para indicar que o método não irá executar dentro de uma transação. É o complementar do Required, ou seja, caso o cliente esteja ou não dentro de uma transação o método será executado sempre fora de uma transação.

**Supports** (Suportado): Nesta modalidade, o método do EJB será executado em uma transação caso o cliente esteja em uma transação. Caso contrário, o EJB será executado sem nenhuma transação. Este é o caso menos recomendável para uso de transações, uma vez que os resultados da execução são inesperados, devido à dependência do suporte transacional do cliente.

**Never** (Nunca): Utiliza-se este atributo quando o EJB (ou método) não pode ser chamado por um cliente que esteja dentro de uma transação. Caso isto ocorra é lançada uma exceção. Normalmente usado quando o método acessa algum recurso que não é transacional e portanto não é capaz de prover as garantias definidas para as transações.

A seguir apresentamos uma tabela que demonstra os efeitos do uso de cada atributo de transação. No exemplo, temos um cliente e um EJB, os quais apresentam transação ou não. Temos duas transações, a transação 1 - T1 - e a

transação 2 - T2. Observe os efeitos de cada atributo de transação, quando o cliente define e não define uma transação.

Tabela 5.1 – Resumo do comportamento dos atributos transacionais.

| Atributo Transacional | Transação do Cliente | Transação do Bean |
|---|---|---|
| Required | Nenhuma | T2 |
| | T1 | T1 |
| Requires New | Nenhuma | T2 |
| | T1 | T2 |
| Mandatory | Nenhuma | Exceção |
| | T1 | T1 |
| NotSupported | Nenhuma | Nenhuma |
| | T1 | Nenhuma |
| Supports | Nenhuma | Nenhuma |
| | T1 | T1 |
| Never | Nenhuma | Nenhuma |
| | T1 | Exceção |

# Segurança

Em uma aplicação J2EE, há duas formas que os clientes devem ser avaliados no acesso ao sistema e aos componentes que ele utiliza. Para um cliente acessar um sistema, inicialmente ele deverá estar autenticado no mesmo. Autenticar um cliente significa que o sistema deve verificar se o cliente é quem ele diz que é. Para isso o mesmo deverá fornecer algumas informações como usuário e senha, ou algum código de acesso ou algo parecido. O sistema autenticará o usuário e sendo assim, associará o mesmo a uma identidade de segurança pré

estabelecida, tal como um perfil de administrador, coordenador ou atendente por exemplo.

Assim que o usuário é autenticado e acessa o sistema, este último deverá apresentar formas de autorizar o usuário a acessar operações do sistema válidas para o seu perfil de usuário. Por exemplo, se o perfil do usuário é de atendente, o mesmo não poderia acessar a operação de relatórios gerenciais.

A autenticação é feita antes de realizar as operações nos EJBs e veremos mais adiante as formas que podem ser utilizadas para tal. Já a autorização é realizada durante a chamada de métodos dos EJBs, que permitem ou negam o acesso de determinado perfil de usuário.

Veremos a seguir uma breve explicação da API JAAS e como utilizá-la

## JAAS

JAAS (Java Authenticated and Autorizated Service) apresenta interfaces que possibilitam que usuário sejam autenticados e autorizados em aplicações J2EE. Com isso, permite que o usuário acesse sistema e operações dele, não importando como é implementado pelo fabricante do servidor J2EE. E assim, o servidor J2EE se encarrega de localizar os dados dos usuários que estão aptos a acessar o sistema, o perfil de cada um, possibilitando os mesmos de acessarem operações específicas oferecidas pela aplicação em questão.

Um usuário poderá acessar um sistema e estar autenticado para o mesmo, sendo ele uma aplicação Web ou uma aplicação standalone. Isso deverá ser transparente para o usuário, sendo que a aplicação J2EE poderia prover as duas interfaces para o usuário com as mesmas funcionalidades.

## Autenticação

Em versões mais antigas da especificação EJB não havia uma API que definia os serviços necessários para

operações de segurança. Com a criação da API JAAS isto foi possível e autenticar um usuário ficou mais simples e portável.

Com foi dito anteriormente, um usuário poderá acessar um sistema por uma interface Web ou standalone, sendo que a aplicação deverá prover os mesmos recursos para o usuário. Em interfaces standalone, a autenticação parece ser mais simplista, tendo a aplicação que utilizar a API JAAS para autenticar o usuário, a partir das informações fornecidas pelo mesmo.

Em interfaces Web isto também é necessário, sendo que o usuário também deverá fornecer informações como usuário e senha para o servidor Web que irá verificar a autenticidade da informação. Para fazer isso, o navegador poderá apresentar quatro formas de realizar uma autenticação que são: Basic Authentication (Autenticação Básica), Form-Based Authentication (Autenticação baseada em Formulários), Digest Authentication (Autenticação com Mensagem Alterada) e Certificate Authentication (Autenticação com Certificado Digital).

Iremos a seguir detalhar cada uma delas.

**Basic Authentication** - o navegador apresenta uma tela de login e fornece ao servidor o usuário e senha para a autenticação. Esta tela depende do navegador que está sendo utilizado.

**Form-Based Authentication** - a aplicação fornece uma página HTML (que poderá ser gerada por um JSP, por exemplo) com um formulário no qual o cliente informaria o usuário e senha. Para isso, há um padrão utilizado pela API JAAS.

```
...
<form method=”post” action=”j_secutiry_check”>
  Usuário: <input type=”text” name=”j_username”>
  Senha:   <input type=”password” name=”j_password”>
</form>
...
```

**Digest Authentication** - para este tipo de autenticação é usado um algoritmo para converter o usuário e senha em um texto ilegível à leitura, dificultando que usuários mau intencionados descubram estas informações. Esta informação é passada ao servidor que utilizando do mesmo algoritmo, autentica o cliente em questão.

**Certificate Authentication** - o servidor recebe do cliente um certificado digital pelo qual será autenticado.

Assim que o cliente é autenticado pela aplicação, o mesmo estará associado a uma identidade ou perfil, que será propagado por toda a aplicação e utilizado pelo servidor para a execução dos métodos dos EJBs, isto é, o seu perfil será utilizado na sua autorização.

## Autorização

Estando cliente autenticado, ele deverá ser autorizado a realizar certas operações fornecidas pelo sistema, de acordo com o seu perfil de usuário. Para isso, a aplicação deve estar configurada com security policies ou regras de segurança para cada serviço fornecido por seus componentes, isto é, para cada método de cada EJB.

A autorização pode ser apresentada de duas formas: Autorização Programática ou Declarativa. Na Autorização Programática o programador deve implementar a verificação de segurança no EJB, isto é, deve verificar qual usuário está acessando o serviço e validar o mesmo. Na Autorização Declarativa o container realiza toda a validação de segurança, não sendo preciso implementá-la. Para isso, deve-se configurar no deployment descriptor as propriedades de segurança para cada EJB e para cada método do mesmo.

Em um mundo perfeito, o melhor forma a utilizar é a Autorização Declarativa. Haverá casos que será necessário mesclar as duas formas de autorização, sendo que somente a declarativa não será suficiente. Por exemplo, se em uma rede de lojas um usuário com perfil de gerente pode acessar o serviços de obtenção de relatórios

gerenciais, sendo somente permitido acessar esses dados das redes de uma determinada praça (região de São Paulo, por exemplo), se faz necessário incluir uma validação programática da região da rede de lojas que o gerente em questão atua.

## Security Roles

O conceito de security roles é simples, mas necessário para o entendimento do uso de autorização. Uma security role é um conjunto de identidades de usuários (identity). Para um usuário ser autorizado a realizar um operação por exemplo, sua identidade deverá estar na correta security role (perfil) para a operação em questão. O uso de security roles é interessante, pois o desenvolvedor não precisa especificar o perfil do usuário no código do EJB.

## Autorização Programática

Para realizar a autorização de forma programática, se faz necessário obter as informações do usuário autenticado na implementação do EJB. Isto deve ser feito utilizando a interface javax.ejb.EJBContext que fornece os métodos getCallerPrincipal() e isCallerInRole(). Vejamos a seguir os métodos desta interface.

```
public interface javax.ejb.EJBContext {
    ...
    public java.security.Principal getCallerPrincipal();
    public Boolean isCallerInRole(String roleName);
    ...
}
```

O método getCallerPrincipal() fornece informações do usuário atual, autenticado no sistema. Este método retorna o objeto java.security.Principal no qual pode-se obter informações importantes do usuário, utilizadas para a sua autorização.

O método isUserInRole(String roleName) verifica se o usuário atual está dentro de uma security role específica.

Dessa forma, pode-se realizar a autorização para um determinado serviço do EJBde forma programática.

Vejamos a seguir um exemplo de deployment descriptor, configurando secutiry roles e links para as security roles reais. Esta última propriedade pode ser configurada, pois em tempo de desenvolvimento o programador pode definir uma security role dentro do seu código e depois no momento da instalação do sistema, a security role real tem um nome diferente, então cria-se um link para o nome correto.

```
...
<enterprise-beans>
   <session>

      ...
      <security-role-ref>
         <description>Perfil do usuário=gerente</description>
         <role-name>gerente</role-name>
         <role-link>manager</role-link>
      </security-role-ref>
      ...

   </session>
   <assembly-descriptor>

      ...
      <security-role>
         <description>Perfil de usuário=gerente</description>
         <role-name>manager</role-name>
      </security-role>
      ...

   </assembly-descriptor>
</enterprise-beans>
...
```

## Autorização Declarativa

A diferença de utilizar autorização declarativa ao invés da declaração programática é que não há a necessidade de programar a autorização, necessitando somente de configurar o deployment descriptor, definindo qual para cada EJB a security role a ser utilizada. Pode-se definir uma security role para todos os métodos do EJB, ou definir uma específica para cada método. Há a possibilidade de excluir métodos que não deseja-se que seja acessado por nenhum perfil de usuário, e isto também deve ser feito no deployment descriptor.

Observe que se alguma operação realizar uma chamada a algum método com um perfil inadequado, isto é, com um perfil que não foi configurado ou definido para o método, o container lançará um exceção do tipo java.lang.SecurityException.

Vejamos a seguir um exemplo de deployment descriptor, configurando security roles e as permissões para os métodos dos EJBs.

```
...
<enterprise-beans>
   <session>

      ...
      <security-role-ref>
         <description>Perfil do usuário=gerente</description>
         <role-name>gerente</role-name>
      </security-role-ref>
      ...

   </session>
   <assembly-descriptor>

      ...
      <method-permission>
         <role-name>gerente</role-name>
         <method>
            <ejb-name>EJBTestX</ejb-name>
```

```
            <method- name>*</method- name>
        </method>
    </method- permission>
    ...

    ...
    <method- permission>
        <role- name>gerente</role- name>
        <method>
            <ejb- name>EJBTestY</ejb- name>
            <method- name>calc</method- name>
            </method>
        <method>
            <ejb- name>EJBTestY</ejb- name>
            <method- name>go</method- name>
        </method>
        <method>
            <ejb- name>EJBTestY</ejb- name>
            <method- name>getAccount</method- name>
        </method>
        <method>
            <ejb- name>EJBTestY</ejb- name>
            <method- name>doTransfer</method- name>
            <method- params>Integer</method- params>
            <method- params>Integer</method- params>
            <method- params>Integer</method- params>
        </method>
        ...
    </method- permission>
    <description>Este método não sera executado</description>
    <method>
        <ejb- name>EJBTestY</ejb- name>
        <method- name>cleanFunds</method- name>
    </method>
    <exclude- list>

    </exclude- list>
    ...
</assembly- descriptor>
```

```
</enterprise- beans>
...
```

## Propagação de Segurança

Em uma aplicação J2EE, teremos com certeza casos em que serviços de alguns EJBs utilizam serviços de outros. Dependendo da aplicação, poderíamos querer que a identidade do usuário (perfil) seja propagado para os métodos que estão sendo chamados pelos próprios EJB para outros EJBs, ou em vez disso, definir um perfil para executar um determinado método em um EJB específico.

Imagine uma situação em que um cliente acessa um serviço de cálculo de um certo imposto para a venda de um produto. Chamaremos este como serviço doXXX do EJB A. Este método chamaria outro método do EJB B, o método doYYY. Podemos definir na aplicação, que este último método execute com a identidade do chamador, isto é, utilize a mesma identidade do cliente que chamou o método doXXX do EJB A. Ou ainda poderíamos definir que o método doYYY do EJB B rodasse com uma nova identidade, definida pela aplicação. Essas duas opções podem ser configuradas no deployment descriptor da aplicação. Vejamos um exemplo a seguir:

```
<enterprise- beans>
   <session>
      <ejb- name>A</ejb- name>

      ...
      <security- identity>
         <use- caller- identity/>
      </security- identity >
      ...

   </session>
   <session>
      <ejb- name>B</ejb- name>

      ...
```

```
            <security-identity>
               <run-as>
                  <role-name>gerente</role-name>
               </run-as>
            </security-identity >
            ...

         </session>

         <assembly-descriptor>

            ...
            <security-role>
               <description>Perfil de usuário=gerente</description>
               <role-name>gerente</role-name>
            </security-role>
            ...

         </assembly-descriptor>

      </enterprise-beans>
```

## Exemplo

Veremos a seguir, um exemplo de módulo de segurança utilizando a API JAAS. Observe que não provemos uma implementação específica para o módulo, sendo o intuito mostrar somente um esqueleto com os métodos principais que devem ser implementados conforme o contrato estabelecido pelas interfaces. Fica a cargo do leitor implementar os métodos se desejado.

### ExampleLoginModule.java

```
package com.book.example.security;

import java.util.*;
import javax.security.auth.*;
import javax.security.auth.spi.*;
import javax.security.auth.login.*;
```

```
import javax.security.auth.callback.*;

// Exemplo do módulo de login para realizar a autenticação do usuário.
// Esta classe não apresenta implementação, somente sendo um ponto de partida
// para a criação de um módulo de login.
public class ExampleLoginModule implements LoginModule {

    // Representa um gurpo de informações relacionados a uma entidade (cliente).
    private Subject subject = null;

    // Inicializa o módulo de login.
    // Este método é executado pelo contexto de login, após a sua instanciação,
    // com o propósito de inicializar este módulo com informações relevantes ao
    // seu uso.
    // @param subject subject a ser autenticado.
    // @param callbackhandler handler utilizado para se comunicar com o cliente.
    // @param sharedState estado compartilhado com outros módulos.
    // @param options opções especificadas na configuração do login.
    public void initialize(Subject subject, CallbackHandler callbackhandler,
            Map sharedState, Map options) {
        this.subject = subject;
    }

    // Este método é utilizado para autenticar o usuário.
    // @return verdadeiro se a autenticação se realizar com sucesso,
    // falso caso contrário.
    // @throws LoginException exceção ao tentar autenticar o usuário.
    public boolean login() throws LoginException {
        // Deve autenticar o usuário e senha do cliente que está tentando acesso
        // Isso pode ser feito por arquivo de properties, LDAP, banco etc.
```

```
    }

    // Utilizado para realizar o commit da autenticação.
    // @return verdadeiro se obtiver sucesso no commit, falso caso
contrário.
    // @throws LoginException caso falhe o commit.
    public boolean commit() throws LoginException {
        return true;
    }

    // Utilizado par abortar o processo de autenticação.
    // Chamado pelo contexto de login se a autenticação falhar.
    // @return verdadeiro caso haja sucesso no abort, falso caso
contrário.
    // @throws LoginException caso falhe no abort.
    public boolean abort() throws LoginException {
        return true;
    }

    // Utilizado para efetuar o logout do usuário autenticado.
    // A sua implementação deverá destruir o Principal e
    // Credenciais do subject (cliente).
    // @return veradeiro caso o logout efetue com sucesso, falso caso
contrário.
    // @throws LoginException exceção caso o logout falhe.
    public boolean logout() throws LoginException {
        return true;
    }
}
```

## ExamplePrincipal.java

```
package com.book.example.security;

import java.security.*;

// Representa o usuário a ser autenticado.
public class ExamplePrincipal implements Principal {

    // Nome do usuário.
    private String name = null;
```

```
    // Construtor customizado.
    // @param name nome do usuário.
    public ExamplePrincipal(String name) {
        this.name = name;
    }

    // Obtém o nome do usuário.
    // @return nome do usuário.
    public String getName() {
        return name;
    }

    // Implementação do método equals.
    // Verifica se um objeto é igual a este.
    // @param obj objeto a ser comparado.
    // @return verdadeiro caso o objeto seja igual a este, falso caso
contrário.
    public boolean equals(Object obj) {
        if (obj == this)
            return true;
        if (!(obj instanceof ExamplePrincipal))
            return false;
        return name.equals(((ExamplePrincipal) obj).getName());
    }

    // Implementação do método hashCode.
    // Informa uma chave de hash utilizada para este objeto.
    // @return chave de hash.
    public int hashCode() {
        return name.hashCode();
    }

    // Retorna uma string com informações do usuário.
    // @return string com informações do usuário.
    public String toString() {
        return name;
    }
}
```

## ExampleRoleGroup.java

```
package com.book.example.security;

import java.util.*;
import java.security.*;
import java.security.acl.*;

// Apresenta o esqueleto para a implementação de um grupo de
usuários.
public class ExampleRoleGroup extends ExamplePrincipal
implements Group {

    // Construtor customizado.
    // @param name nome do usuário.
    public ExampleRoleGroup(String name) {
        // implementação do construtor...
    }

    // Adiciona o usuário em questão ao grupo.
    // @param user usuário a ser adicionado ao grupo.
    // @return veradeiro se o usuário foi adicionado com sucesso,
    // falso caso contrário.
    public boolean addMember(Principal user) {
        return true;
    }

    // Remove o usuário do grupo.
    // @param user usuário a ser adicionado ao grupo.
    // @return veradeiro se o usuário foi removido com sucesso,
    // falso caso contrário.
    public boolean removeMember(Principal user) {
        Object prev = members.remove(user);
        return prev != null;
    }

    // Verifica se o usuário é membro do grupo.
    // @param member usuário que deverá ser verificado se é
membro.
    // @return verdadeiro se o usuário é membro, falso caso
contrário.
```

```
    public boolean isMember(Principal member) {
        return true;
    }

    // Retorna uma lista com os membros do grupo.
    // @return lista com os membros do grupo.
    public Enumeration members() {
        return Collections.enumeration(members.values());
    }
}
```

## ExampleTest.java

```
package com.book.example.security;

import java.util.*;
import javax.rmi.*;
import javax.naming.*;
import java.security.*;

// Implementação de uma ação privilegiada.
// Executada com segurança.
public class ExampleTest implements PrivilegedAction {

    // Executa a ação com segurança.
    // @return informação retornada pelo método do EJB.
    public Object run() {
        try {
            Context ctx = new InitialContext();
            Object obj = ctx.lookup("SecurityExampleTest");
            SecurityExampleTestHome home =
(SecurityExampleTestHome)
                PortableRemoteObject.narrow(obj,
SecurityExampleTestHome.class);
            SecurityExampleTest test = home.create();

            return test.doIt();

        }catch (Exception ex) {
            ex.printStackTrace();
        }
```

```
    }
}
```

**ExampleMain.java**

```
package com.book.example.security;

import javax.naming.*;
import javax.security.auth.*;
import javax.security.auth.login.*;

public class ExampleMain {

    public static void main(String[] args) throws Exception {

        LoginContext loginContext = new LoginContext
("ExampleTest");
        loginContext.login();

        Subject subject = loginContext.getSubject();

        ExampleTest exampleTest = new ExampleTest();
        Subject.doAs(subject, action);
    }
}
```

Observamos que o uso de transações é inevitável para que a aplicação apresente eficiência e robustez. Agora sabemos que a arquitetura J2EE provê transações do tipo Flat e que podemos utilizá-las, tanto como Container-Managed como Bean-Managed, para gerenciar as transações de nossa aplicação.

# Capítulo 16

# Descobrindo Enterprise JavaBeans

## 16.1 Qual servidor J2EE utilizar?

A quantidade de servidores de aplicação J2EE encontrados no mercado hoje é grande. Existem servidores de aplicação de uso livre e outros proprietários, alguns dispondo de alguns serviços como diferenciais, outros dispondo de um desempenho melhor.

Quando decidimos em usar a tecnologia J2EE e precisamos escolher qua servidor de aplicações utilizar, devemos levar em consideração aspectos como desempenho, tamanho da aplicação, recursos que serão utilizados, custos entre outros.

Para executar e exemplificar a aplicação J2EE que iremos apresentar neste capítulo, iremos utilizar o servidor de aplicações livre Jboss na versão 3.0.1. Já existem novas versões deste servidor de aplicações, mas não entraremos em detalhes. A escolha deste servidor de aplicações, se deve ao seu grande uso em aplicações J2EE que não necessitam de um servidor proprietário. Apresenta uma solução ótima e de alto desempenho para os custos envolvidos.

Não é o enfoque deste tópico avaliar outros servidores de aplicação e decidir qual deles é o melhor, mas nas

referências no final do livro, pode-se ter acesso aos links dos fabricantes de diversos servidores de aplicações comerciais.

## 16.2 Instalando, configurando e executando um Servidor J2EE

Não existem muitos segredos para instalar, configurar e executar o servidor de aplicações Jboss.

O que devemos fazer é copiar o pacote do servidor de aplicações Jboss na versão 3.0.1, do site do próprio Jboss (http://www.jboss.org). O arquivo que foi copiado deve ter sido um arquivo compactado. Este arquivo deve ser descompactado em um diretório ou sub-diretório no qual deseja-se ter a instalação do servidor de aplicações Jboss. Após esta fase a instalação está completa.

O interessante é que não precisaremos realizar nenhuma configuração para executar os exemplos contidos neste livro. A configuração padrão da instalação do Jboss já nos proporciona todos os serviços necessários para a nossa aplicação exemplo. Veremos que necessitaremos de utilizar um banco de dados, mas isto não será problema e nem precisaremos nos precocupar em instalar um. O Jboss oferece um banco de dados para realizar testes e executar exemplos. Assim, a fase de configuração também está completa sem muita dificuldade.

Para executar o servidor de aplicações Jboss também é uma operação muito simples. Execute o prompt de comando do seu sistema operacional e acesse o diretório de instalação do Jboss. Entre no diretório bin, isto é, <Instalação_Jboss>/bin, e execute o comando run.bat para iniciar o servidor. Então deve-se ter o seguinte: <Instalação_Jboss>/bin/run. Com isto, o servidor de aplicações inicializará, gravando várias mensagens no console, até aparecer a última mensagem informando que o servidor foi inicializado com sucesso. Assim, poderemos realizar a instalação da aplicação exemplo que veremos

logo a seguir, que também não apresentará muita dificuldade.

Observe que também não é o intuito deste capítulo, explorar os recursos e detalhes de configuração do servidor de apliações Jboss. Para maiores detalhes consulte a documentação oferecida e disponível no site do Jboss. Se não for o suficiente, o Jboss vende uma documentação mais completa para pessoas que desejam utilizar o servidor Jboss em aplicações comerciais, e que necessitam de informações mais complexas sobre o servidor.

Veremos a seguir, tópicos que exemplificam na forma de uma aplicação, os tipos de EJB apresentados nos capítulos anteriores deste livro. A aplicação completa se encontra no final do livro no apêndice Aplicação J2EE - Exemplo. Então, cada tópico seguinte trará explicações detalhadas do que foi implementado em cada EJB. No fim, poderemos juntar todos os EJBs e outras classes auxiliares em uma única aplicação e fazermos a instalação e acesso à aplicação exemplo.

A aplicação exemplo apresentada nos próximos tópicos se trata de uma aplicação de venda de produtos, no qual um usuário poderá escolher alguns produtos, calcular algumas informações pertinentes a estes produtos e incluindo estes produtos em uma cesta de compras. Poderá ainda excluir, alterar ou remover todos os produtos de sua cesta. Na finalização da compra, os dados dos produtos escolhidos serão persistidos e o usuário será avisado do término da compra, com os dados dos produtos vendidos.

## 16.3 Criando um Session Bean Stateless

Neste tópico serão analisadas as partes da codificação do EJB Session Bean Stateless, que faz parte da aplicação exemplo. Não detalharemos as interfaces, pois elas só apresentam o contrato dos métodos de negócio e ciclo de

vida. Os métodos de negócio serão brevemente explicados.

O EJB Session Bean utilizado como exemplo, apresenta serviços de cálculo do desconto concedido ao usuário e cálculo de parcelas para um produto em venda. O nome deste EJB é SalesSupportBean.

O código a seguir, pertencente a interface Home do EJB SalesSupportBean, apresenta o método que solicita a criação do EJB e retorna a referência para a execução dos métodos de negócio.

```
...
public SalesSupport create() throws CreateException,
RemoteException;
...
```

Observe que o método create() ao ser executado, solicita ao container que crie uma instância do EJB SalesSuuportBean. Dessa forma o container cria a instância do objeto, do tipo EJBObject que encapsula o EJB SalesSupportBean e retorna ao cliente chamador uma referência da interface Remote, na qual se tem acesso aos métodos de negócios ou serviços disponíveis para o EJB em questão.

A seguir, observe os métodos da interface Remote do EJB SalesSupportBean, métodos de negócio utilizados pelo cliente para calcular o desconto concedido ao produto vendido e cálculo das parcelas a serem pagas.

```
...
public java.lang.Integer calcDiscount(Integer value, Integer range)
        throws RemoteException;
public java.lang.Integer calcPiece(Integer value, Integer times)
        throws RemoteException;
...
```

A seguir vamos analisar a implementação do EJB SalesSupportBean. Iremos somente comentar os métodos de negócio deste EJB que são muito simples. Os métodos de ciclo de vida do EJB serão somente citados.

No método de ciclo de vida ejbCreate() que apresenta a mesma assinatura da interface Home SalesSupportHome, não foi necessário nenhuma implementação, sendo assim este método não contém lógica. Isto também ocorreu para os métodos ejbRemove(), ejbActivate() e ejbPassivate(), por não necessitar de executar operações nos momentos de criação e remoção da instância do EJB, e na ativação e passivação quando ocorre a liberação de recursos alocados, mas que não é o caso.

Os métodos de negócio calcDiscount() e calcPiece() apresentam operações simples para o cálculo do desconto concedido na venda do produto e cálculo do valor de parcelas de acordo com o valor da venda.

```
...
public java.lang.Integer calcDiscount(Integer value, Integer perc) {
    return new Integer(value.intValue() - (value.intValue() *
perc.intValue()));
}
public java.lang.Integer calcPiece(Integer value, Integer times) {
    return new Integer(value.intValue() / times.intValue());
}
...
```

Observe que o método calcDiscount() retorna o resultado da operação do valor de venda diminuido do valor de desconto concedido de acordo com o percentual informado. O método calcPiece() somente divide o valor total da venda pela quantidade de vezes também informada para o método.

Veja que as operações implmentadas são bem simples e também não é o escopo deste livro, detalhar demasiadamente as operações, sendo que estas servem somente de exemplo e apresentam uma idéia de como deve-se implementar os serviços oferecidos pelo EJB Session Bean Stateless.

Abaixo veremos o deployment descriptor do EJB SalesSupportBean, arquivo que deve ser criado juntamente com o EJB e utilizado pelo servidor de aplicações para obter informações adicionais do EJB e realizar o deploy do

mesmo. Este arquivo tende a ser simples para EJBs Session Bean. Iremos somente comentar algumas linhas.

As linhas iniciais apresentam o nome de exibição e o nome propriamente dito do EJB, utilizados pelo servidor de aplicações. A seguir seguem os nomes das interfaces Home e Remote e o nome da classe que implementa o EJB em questão. Depois é informado o tipo de EJB Session Bean, que pode ser Stateless ou Stateful e por fim o tipo de transação, isto é, se será gerenciada pelo Container ou pelo Bean. Por fim é definido o tipo de transação utilizada para cada método específico do EJB. Deve-se fazer referência ao EJB que se está definindo as propriedades de transação, pelo seu nome anteriormente definido e a seguir, definir para cada método o tipo de transação que este se enquadrará. No nosso caso utilizamos um asterístico, com isso todos os métodos do EJB informado utilizarão o mesmo tipo de transação que foi definida como Required.

```
...
<session>
   <display-name>SalesSupport</display-name>
   <ejb-name>SalesSupport</ejb-name>

<home>com.book.project.ejb.session.SalesSupportHome</home>
   <remote>com.book.project.ejb.session.SalesSupport</remote>
   <ejb-class>com.book.project.ejb.session.SalesSupportBean</ejb-
class>
   <session-type>Stateless</session-type>
   <transaction-type>Container</transaction-type>
</session>
</enterprise-beans>
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>SalesSupport</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Required</trans-attribute>
      </container-transaction>
```

...

## 16.4 Criando um Session Bean Stateful

Iremos agora detalhar um exemplo de EJB Session Bean Stateful, utilizado na aplicação exemplo. Não serão detalhados os métodos de ciclo de vida da interface Home SalesBasketHome neste tópico, pois o mesmo já foi abordado no tópico anterior e o seu comportamento é o mesmo. A única exceção se faz para o método ejbCreate(), por um pequeno detalhe que será mostrado.

Este EJB, chamado de SalesBasketBean, será encarregado de prover todos os serviços de venda dos produtos escolhidos. Para isso, fornece métodos de negócio tais como adicionar produtos na cesta de compras, removê-los, limpar a cesta, calcular o preço total dos produtos etc. Assim, vamos analisar cada método de negócio deste EJB. A seguir segue a assinatura dos métodos de negócio constantes na interface Remote SalesBasket.

```
...
public void initSales(com.book.project.vo.UserVO user)
        throws RemoteException;
public java.lang.Boolean finalizeSale() throws RemoteException;
public void addProduct(com.book.project.vo.ProductVO product)
        throws RemoteException;
public java.lang.Boolean removeProduct
(com.book.project.vo.ProductVO
        product) throws RemoteException;
public java.lang.Integer calcBasketPrice() throws RemoteException;
public void freeBasket() throws RemoteException;
...
```

Na classe de implementação do EJB SalesBasketBean, criamos um atributo chamado basket, que é uma Map que mantém os produtos adicionados na cesta de compras. Com isto, entendemos a diferença do EJB Session Bean Stateless do EJB Session Bean Stateful, no qual este último mantém os dados de um cliente específico, isto é, mantém

os produtos na cesta de um determinado cliente e não compartilha estes dados com outros usuários.

```
...
private Map basket;
...
```

Também mantemos no EJB SalesBasketBean os dados do usuário que iniciou a compra, para no final desta, validarmos seus dados e relacionarmos os produtos comprados para um determinado usuário. Observe que existe um objeto do tipo UserVO. Este objeto é utilizado para carregar informações do usuário em questão.

```
...
private UserVO user;
...
```

O método ejbCreate() do EJB SalesBasketBean somente inicializa a Map que contém os produtos comprados por um determinado usuário.

```
...
public void ejbCreate() throws CreateException {
    this.basket = new HashMap();
}
...
```

O EJB SalesBasketBean dispõe de um método utilizado par ainicilizar a venda. Para isso, configura o usuário que estará realizando a compra. Observe que esta operação poderia ser executada no método ejbCreate(), passando o objeto UserVO como parâmetro no momento da criação da instância do EJB.

```
...
public void initSales(UserVO user) {
    this.user = user;
}
...
```

O trecho a seguir implementa o método de finalização da compra. Esta operação é um pouco mais complexa que as apresentadas até o momento, então atente para os detalhes. Inicialmente é solicitado à uma classe chamada

de SalesServiceLocator, uma referência para a instância do EJB Entity Bean ProductBean, utilizado para persistir os dados dos produtos vendidos para determinado usuário. A operação de obter a referência para o EJB ProductBean está implementada na classe SalesServiceLocator como pode-se notar.

Esta classe implementa um Design Pattern bastante conhecido, no qual oculta a implementação da localização do EJB e facilita esta operação para as classes que a utilizarão. Não é o intuito deste livro detalhar Design Patterns utilizados nos códigos. Para maiores detalhes consulte as referências bibliográficas.

Após os detalhes deste método, apresentamos a implementação do método do SalesServiceLocator que localiza o EJB e retorna a referência para os seus clientes.

Com a referência da interface Home ProcuctLocalHome, podemos realizar a execução do método create() para cada produto contido na cesta de produtos. Persistimos cada produto da cesta do usuário, fazendo isso por meio de um iterador. Este iterador irá obter cada produto contido na Map e criar uma instância do EJB ProductBean.

```
...
public java.lang.Boolean finalizeSale() throws Exception {
    ProductLocalHome localHome = (ProductLocalHome)
        SalesServiceLocator.getInstance().
        getLocalHome(SalesServiceLocator.PRODUCT_BEAN);
    try {
        for (Iterator i = basket.entrySet().iterator(); i.hasNext(); ) {
        Map.Entry product = (Map.Entry) i.next();
        ProductVO vo = (ProductVO) product.getValue();
        localHome.create(vo.getName(), vo.getId(), vo.getDescription(),
            new Integer(vo.getPrice()));
    }
        return new Boolean(true);
    }catch (CreateException ex) {
        throw new Exception("Error while trying to finalize Sale. " +
ex);
    }
}
```

```
...
```

A seguir vemos a implementação do método getLocalHome, da classe SalesServiceLocator, na qual é obtido referência para as interfaces dos EJBs contidos no servidor de aplicação. Veja que a operação é simples, somente realizando um lookup (localização) do EJB solicitado pelo seu nome JNDI. Para efeito de melhora de desempenho, a referência para as interfaces Home solicitadas são armazenadas em um cachê e podendo ser utilizadas futuramente em outras chamadas a este método.

```
...
public synchronized EJBLocalHome getLocalHome(String jndiName)
        throws RuntimeException {
    try {
        if (! localHomeCache.containsKey(jndiName)) {
            EJBLocalHome localHome = (EJBLocalHome) ctx.lookup
(jndiName);
            localHomeCache.put(jndiName, localHome);
        }
        return (EJBLocalHome) localHomeCache.get(jndiName);
    } catch (Exception ex) {
        throw new RuntimeException("Error while trying to get a local
home object
            reference of " + jndiName + ex);
    }
}
...
```

A seguir continuamos com os métodos do EJB SalesBasketBean.

A implementação dos métodos addProduct() e removeProduct() que adicionam e removem produtos da cesta é simples. Utilizamos os métodos de adição e remoção de objetos do tipo ProductVO na Map. Também há a implemetação de um serviço para limpar todos os produtos da cesta. Este método também aproveita o método clear da Map e executa-o.

```
...
public void addProduct(com.book.project.vo.ProductVO product) {
```

```
    this.basket.put(product, product);
}
public java.lang.Boolean removeProduct
(com.book.project.vo.ProductVO
        product) {
    if (this.basket.remove(product) != null)
    return new Boolean(true);
    return new Boolean(false);
    }
    public void freeBasket() {
        this.basket.clear();
}
...
```

O EJB também apresenta o método de cálculo do preço total dos produtos contidos na cesta. Esta também é uma operação simples, que consiste em iterar todos os produtos contidos na cesta, acumulando os valores dos produtos e retornando este valor.

```
...
public Integer calcBasketPrice() {
    int value = 0;
    for (Iterator i = basket.entrySet().iterator(); i.hasNext();) {
        Map.Entry entry = (Map.Entry) i.next();
        value =+ ((ProductVO) entry.getValue()).getPrice();
    }
    return new Integer(value);
}
...
```

Observe que este EJBtambém não apresenta complexidade de implementação de código, mas como já foi dito, este não é o intuito do livro. O enfoque desta aplicação exemplo é de somente validar os conhecimentos de EJBs..

A seguir segue o deployment descriptor do EJB SalesBasketBean. Também não apresenta muitos segredos e a única diferença para o deployment descriptor do EJB Session Bean Stateless visto no tópico anterior, é na tag tipo de EJB Session Bean. Nesta é configurado o nome Stateful, para informar que o EJB a qual está se referenciando, se trata de um EJB Session Bean Stateful.

Também não houve alterações na configuração do tipo de transação para cada método, utilizando o mesmo tipo de transação para todos os métodos do EJB SalesBasketBean, conforme explicado no tópico anterior.

```
...
<ejb-jar>
   <enterprise-beans>
      <session>
         <display-name>SalesBasket</display-name>
         <ejb-name>SalesBasket</ejb-name>

<home>com.book.project.ejb.session.SalesBasketHome</home>

<remote>com.book.project.ejb.session.SalesBasket</remote>
         <ejb-
class>com.book.project.ejb.session.SalesBasketBean</ejb-class>
         <session-type>Stateful</session-type>
         <transaction-type>Container</transaction-type>
      </session>
...
```

# 16. 5 Criando um Entity Bean BMP

O próximo EJB a ser comentado será o EJB Entity Bean BMP UserBean. Neste EJB foi implementado toda a lógica de persistência e manipulação do objeto. Para realizar a persistência em um banco de dados, não precisaremos instalar e configurar um. O próprio servidor de aplicações Jboss fornece um banco de dados como padrão, que pode ser utilizado para executar exemplo de aplicações J2EE. Então utilizaremos esse recurso para manter os EJB Entity Beans.

Veremos as interfaces Home, LocalHome, Remote e Local que são respectivamente UserHome, UserLocalHome, User e UserLocal. Não iremos detalhar todas, somente a Home e Remote, pois as interfaces utilizadas para acesso local apresentam os mesmos métodos, somente suprimindo a exceção.

A seguir temos o contrato da interface Home - UserHome = que apresenta os métodos de ciclo de vida do EJB. O método create() é utilizado para solicitar- se uma instância do EJB para o container. Neste método são informados os valores do objeto que serão persistidos. Este método declara as exceções CreateException utilizada em caso de erro na persistência do objeto, sendo uma exceção de Runtime e a RemoteException utilizada para outros erros que poderiam ser tratados. Veremos mais detalhes deste método, isto é, da sua implementação no próprio EJB UserBean.

O outro método de ciclo de vida definido nesta interface é o método findByPrimaryKey(). Este método é utilizado para localizar um objeto persistido através de sua chave primária, que no nosso caso será o CPF do usuário. Na execução deste método, não é necessário utilizar o método create(), pois não iremos criar um novo objeto e sim localizar um objeto anteriormente persistido. Com isso, o método findByPrimaryKey() também retorna uma referência para a instância do EJB. Declara as exceções FinderException utilizada caso não consiga localizar o objeto através da chave- primária informada, e a exceção RemoteException para outros erros na execução do método.

```
...
public interface UserHome extends javax.ejb.EJBHome {
    public User create(String name, Integer cpf, String address, String
email)
        throws CreateException, RemoteException;
    public User findByPrimaryKey(Integer cpf) throws
FinderException,
        RemoteException;
}
...
```

O código a seguir apresenta os métodos de negócio do EJB. Estes métodos são definidos nas interfaces Remote e Local. Iremos detalhar somente a interface Remote - User - pois a interface Local - UserLocal - apresenta os mesmo métodos e a única diferença é que os métodos não declaram exceção necessariamente.

Vemos os métodos de negócio do EJB UserBean. São definidos nesta interface os métodos getters e setters para o objeto em questão. A única particularidade da interface Remote em relação a Local como dito anteriormente é a exceção RemoteException, sendo que no acesso remoto é utilizado em caso de erro na execução dos métodos de negócio que seguem.

```
...
public interface User extends javax.ejb.EJBObject {
    public String getName() throws RemoteException;
    public Integer getCpf() throws RemoteException;
    public String getAddress() throws RemoteException;
    public String getEmail() throws RemoteException;
    public void setName(String name) throws RemoteException;
    public void setAddress(String address) throws RemoteException;
    public void setEmail(String email) throws RemoteException;
}
...
```

Veremos em seguida o detalhamento da implementação do EJBEntity Bean BMP - UserBean.

Definimos um atributo chamado connection, utilizado para manter a conexão com o banco de dados responsável por manter os dados do objeto. Essa conexão é obtida por meio de um nome JNDI definido pelo servidor de aplicações Jboss como DataSource padrão configura neste servidor. Usaremos este para persistirmos os objetos. Veremos que definimos um método para obter a conexão, o método getConnection() utilizado em alguns métodos de persistência do objeto.

Seguem também alguns atributos definidos como constantes que apresentam as operações de seleção, remoção, atualização e localização dos objetos em SQL. Estes atributos são utilizados em determinados métodos que serão vistos posteriormente, para realizar as operções respectivas. Observe que essas operações no banco de dados poderiam não ser definidas como constantes do EJB, mas serem criadas somente em cada método. Esta é uma decisão do programador e aqui segue somente como exemplo.

```
...
private transient Connection connection = null;

// Query de inserção do objeto no BD.
private final String INSERT_QUERY =
        "INSERT INTO USER(Name, Cpf, Address, Email) VALUES
(?, ?, ?, ?)";

// Query de atualização do objeto no BD.
private final String UPDATE_QUERY =
        "UPDATE USER SET Name = ?, Address = ?, Email = ? WHERE
Cpf = ?";

// Query de remoção do objeto no BD.
private final String DELETE_QUERY = "DELETE FROM USER WHERE
Cpf = ?";

// Query de obtenção do objeto do BD.
private final String SELECT_QUERY = "SELECT Name, Cpf, Address,
        Email FROM USER WHERE Cpf = ?";

// Query utilizada para obter os objetos dado sua chave primária.
private final String FIND_BY_PK = "SELECT Name, Cpf, Address,
        Email FROM USER WHERE Cpf = ?";

// Cria uma conexão com o banco de dados padrão.
// @return conexão do banco de dados.
private Connection getConnection() {
    Connection conn = null;
    try {
        InitialContext ctx = new InitialContext();
```

```
            conn = (Connection) ctx.lookup("java:/DefaultDS");
        }catch (Exception e){
            throw new RuntimeException(e.getMessage());
        }
        return conn;
    }
    ...
```

Vemos a seguir a implementação do método ejbCreate() definido na interface Home e LocalHome. Neste método são informados os atributos do objeto a serem persistidos e nele é implementado a operação de gravação dos dados em um banco de dados. Observe que obtemos uma conexão com o banco de dados, preparamos a operação a ser executada, realizamos o bind dos atributos e em seguida executamos o comando. Por fim, são configurados todos os atributos do objeto e retornado o seu atributo chave-primária para o container. Este tratará essa informação e retornará ao objeto chamador uma referência para a interface Remote ou Local, disponibilizando assim a execução dos métodos de negócio.

Não mostramos detalhes do método ejbPostCreate(), pois não há implementação. Somente é importante salientar que este método deve ter a mesma assinatura do método create(). Normalmente ele é utilizado para configurar relacionamente entre EJB Entity Beans, mas isto não é detalhado neste livro. Maiores detalhes sobre relacionamentos de EJB Entity Bean consulte a bibliografia.

```
    ...
    public Integer ejbCreate(String name, Integer cpf, String address,
    String email)
            throws CreateException {
        try {
            connection = getConnection();
            try {
                PreparedStatement stmt = connection.prepareStatement
    (INSERT_QUERY);
                try {
                    stmt.setString(1, name);
                    stmt.setInt(2, cpf.intValue());
```

```
                stmt.setString(3, address);
                stmt.setString(4, email);
                stmt.execute();
            }
            finally {stmt.close();}
            }
            finally {connection.close();}
    }catch (Exception e){
        throw new CreateException();
    }
    setName(name);
    setCpf(cpf);
    setAddress(address);
    setEmail(email);
    return getCpf();
}
...
```

O método a seguir trata da remoção do objeto do meio persistente. Como no método ejbCreate(), o método ejbRemove() também obtém uma conexão com o meio de persistência, prepara a execução do comando SQL, realiza o bind da variável pelo chave-primária e executa o comando. Assim, o registro definido com a chave-primária informada é removido do banco de dados. Esse método não retorna nenhum valor, mas em caso de erro na execução do método, a exceção RemoveException é lançada e poderá ser tratada pelo objeto chamador.

```
...
public void ejbRemove() throws RemoveException {
    try {
        connection = getConnection();
        try {
            PreparedStatement stmt = connection.prepareStatement
(DELETE_QUERY);
            try {
                stmt.setInt(1, this.cpf.intValue());
                stmt.execute();
            }
            finally { stmt.close(); }
```

```
        }
        finally { connection.close(); }
    } catch (Exception e){
        throw new RemoveException("Error removing element.");
    }
}
...
```

O próximo método ejbFindByPrimaryKey() apresenta a localização do objeto por meio de sua chave-primária. A chave deve ser informada para o método e o objeto é localizado no meio persistente e uma instância sua é criada no container do servidor de aplicação. É então retornado sua chave primária que será tratada pelo container que disponibilizará para o objeto chamador a referência para interface Remote ou Local, para a execução dos métodos de negócio. A implementação deste método também é muito parecida com os anteriores, obtendo uma conexão, preparando o comando SQL, realizando o bind da variável tida como chave-primária, executando o comando, configurando cada atributo com os valores localizados no meio persistente e por fim, retornando para o container a chave-primária do objeto. Em caso de erros na localização do objeto, uma exceção FinderException é lançada.

```
...
public Integer ejbFindByPrimaryKey(Integer cpf) throws
FinderException {
    try {
        connection = getConnection();
        try {
            PreparedStatement stmt = connection.prepareStatement
(FIND_BY_PK);
            try {
                stmt.setInt(1, cpf.intValue());
                ResultSet rs = stmt.executeQuery();
                try {
                    if (rs.next()) {
                        this.setName(rs.getString(1));
                        this.setAddress(rs.getString(2));
```

```
                    this.setEmail(rs.getString(3));
                    this.setCpf(new Integer(rs.getInt(4)));
                }
            }
            finally {rs.close();}
        }
        finally {stmt.close();}
    }
    finally {connection.close();}
}catch (Exception e){
    throw new FinderException("Error finding object by id: " +
cpf);
}
return this.cpf;
}
...
```

O método ejbLoad() é executado pelo container quando este deseja carregar os dados do objeto do meio persistente e mantê- lo em memória. Também não retorna nenhum valor, somente atualiza os dados do objeto atual. A implementação também é trivial realizando as mesmas operações dos métodos de persistência anteriores.

```
...
public void ejbLoad() {
    try {
        connection = getConnection();
        try {
            PreparedStatement stmt = connection.prepareStatement
(SELECT_QUERY);
            try {
                stmt.setInt(1, this.cpf.intValue());
                ResultSet rs = stmt.executeQuery();
                try {
                    if (rs.next()) {
                        this.setName(rs.getString(1));
                        this.setAddress(rs.getString(2));
                        this.setEmail(rs.getString(3));
                    }
                }
```

```
                finally {rs.close();}
            }
            finally {stmt.close();}
        }
        finally {connection.close();}
    }catch (Exception e){
        throw new EJBException("Error loading objects.");
    }
}
...
```

O método ejbStore() também não apresenta nenhuma complexidade e é muito parecido com o método anterior. Este método é responsável por atualizar os dados do objeto em memória no meio persistente, isto é, para cada atributo configurado no objeto em memória, este objeto é executado para manter o sincronismo com os dados persistentes. Observe que este método executa a operação contrária do método ejbLoad().

```
...
public void ejbStore() {
    try {
        connection = getConnection();
        try {
            PreparedStatement stmt = connection.prepareStatement
(UPDATE_QUERY);
            try {
                stmt.setString(1, this.name);
                stmt.setString(2, this.address);
                stmt.setString(3, this.email);
                stmt.setInt(4, this.cpf.intValue());
                stmt.execute();
            }
            finally {stmt.close();}
        }
        finally { connection.close(); }
    }catch (Exception e){
        throw new EJBException("Error persisting objects.");
    }
}
```

...

Não iremos detalhar os métodos ejbActivate() e ejbPassivate(), pois os mesmo não apresentam implementação. Os métodos setters e getters também não serão explorados por não apresentarem nenhuma dificuldade. Também suprimimos os métodos setEntityContext() e unsetEntityContext() pelos mesmos motivos.

Por fim observamos o deployment descriptor do EJB Entity Bean BMP = User Bean - o qual apresenta algumas particularidades e diferenças em relação aos anteriores EJB Session Bean e por isso serão detalhados.

Observe que a tag posterior a enterprise-beans é a tag entity, informando que os dados que seguem irão informar detalhes de um EJB Entity Bean. As linhas iniciais são as mesmas, definindo o nome do EJB a ser mostrado e o nome utilizado pelo container para referenciar o EJB.

Após são informados os nomes das interfaces Home, Remote, LocalHome e Local e o nome da classe que implementa o EJB. A seguir o tipo de persistência utilizado par aum EJB Entity Bean que podem ser duas: Bean ou Container. No nosso caso, como implementamos os métodos de persistência do EJB, esta deve ser configurada como Bean. Depois devemos informar qual a classe da chave-primária do EJB. Vemos que neste caso a classe é uma classe da API Java, mas poderia ser uma classe definida para a aplicação. Por fim, a tag reentrant define se o EJB pode chamar outro EJB.

Os atributos de transação não mudaram, sendo iguais aos dos EJB Session Beans.

```
...
<?xml version=”1.0" encoding=”UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC “-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN” “http://java.sun.com/dtd/ejb-
jar_2_0.dtd”>
<ejb-jar>
   <enterprise-beans>
      <entity>
```

```
            <display-name>User</display-name>
            <ejb-name>User</ejb-name>
            <home>com.book.project.ejb.entity.UserHome</home>
            <remote>com.book.project.ejb.entity.User</remote>
            <local-
home>com.book.project.ejb.entity.UserLocalHome</local-home>
            <local>com.book.project.ejb.entity.UserLocal</local>
            <ejb-class>com.book.project.ejb.entity.UserBean</ejb-
class>
            <persistence-type>Bean</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
        </entity>
    </enterprise-beans>
...
```

# 16.6 Criando um Entity Bean CMP

Vamos agora analisar os detalhes da implementação do EJB Entity Bean CMP - ProductBean. Este EJB apresenta as interfaces Home, Remote, LocalHome e Local que são ProductHome, Product, ProductLocalHome e ProductLocal, além da classe de implementação do EJB ProductBean e seu deployment descriptor. Há uma novidade para este EJB. Implementamos a classe da chave- primária que se chamada ProductPK. Como vimos no exemplo do EJB Entity Bean BMP - UserBean - este não implementava uma classe para a chave primária, utilizando uma classe da API Java. Faremos diferente neste EJB para exemplificar o seu uso.

A seguir vemos um trecho do código da interface Home - ProductHome. Ela define os métodos de ciclo de vida do EJB tais como o create(), que recebe como parâmetros os valores a serem persistidos no bando de dados e também declara as exceções CreateException, quando ocorre algum erro na persistência do objeto e a RemoteException para outros erros. Define também os métodos conhecidos como finders, utilizados para localizar objetos já persistidos. Neste caso, definimos somente o método-padrão findByPrimaryKey() informando qual o valor da

chave- primária do objeto que deseja- se localizar. Este método também declara as exceções FinderException, para erros na localização do objeto no meio persistente e a RemoteException para outros erros.

A interface LocalHome também define os mesmos métodos, tendo a única diferenção de não declarar as exceções remotas - RemoteException.

```
...
public Product create(String name, String description, Integer price,
        ProductPK productPK) throws CreateException,
RemoteException;
public Product findByPrimaryKey(ProductPK pk) throws
FinderException,
        RemoteException;
...
```

A interface Remote - Product - apresenta os métodos de negócio do EJB. Neste somente declaramos os métodos getters e setters para o EJB Entity Bean CMP - ProductBean. Não iremos detalhá- los, pois são muito simples. A interface Local = ProductLocal - também define esses mesmos métodos, com a única diferença de não declarar as exceções remotas - RemoteException.

Logo a seguir apresentamos a implementação da classe de chave- primária, utilizada para o EJB ProductBean. Esta classe também não apresenta nenhuma complexidade, somente apresentando os atributos de chave- primária que no caso é somente um id. Importante salientar que como esta classe será utilizada pelos clientes remotos, ela deve implementar a interface Serializable.

```
...
public class ProductPK implements Serializable {
...
```

O próximo código apresenta a implementação do EJB Entity Bean CMP - Product Bean - propriamente dito. Veremos que também não apresenta muita complexidade, pois os métodos de persistência e manutenção do objeto são implementadas pelo container do servidor de aplicações, tirando essa responsabilidade do implementador, assim não havendo códigos para estes métodos.

A classe deve ser declarada como abstrata, pois o container irá implementá-la quando esta for instalada no servidor de aplicações na aplicação exemplo.

```
...
abstract public class ProductBean implements EntityBean {
...
```

Observe que o método ejbCreate() somente configura os atributos do objeto e retorna o valor da chave-primária. A persistência dos seus dados fica a cargo do container.

```
...
public ProductPK ejbCreate(java.lang.String name, java.lang.String
description,
        java.lang.Integer price, com.book.project.ejb.entity.ProductPK
        productPK) throws CreateException {
    setName(name);
    setDescription(description);
    setPrice(price);
    setProductPK(productPK);
    return getProductPK();
}
...
```

Veja também que os métodos de manipulação do objeto tais como ejbRemove(), ejbLoad() e ejbStore() são desprovidos de implementação, ficando também a cargo do container esta tarefa. Os métodos de negócio também não apresentam implementação, mas a diferença para os métodos de ciclo de vida é que devem ser declarados como abstratos, pois serão implementados pelo próprio container novamente.

```
...
public void ejbRemove() throws RemoveException {
```

```
}

public void ejbLoad() {
}

public void ejbStore() {
}

public abstract void setName(java.lang.String name);
public abstract void setDescription(java.lang.String description);
public abstract void setPrice(java.lang.Integer price);
public abstract void setProductPK
(com.book.project.ejb.entity.ProductPK
      productPK);
public abstract java.lang.String getName();
public abstract java.lang.String getDescription();
public abstract java.lang.Integer getPrice();
public abstract com.book.project.ejb.entity.ProductPK getProductPK();
...
```

O deployment descriptor do EJB Entity Bean CMP - Product Bean - não traz muitas novidades em relação ao EJB Entity Bean BMP - UserBean. Uma das poucas diferenças deste é que o tipo de persistência do EJB deve ser configurado como Container, pois ficará a cargo deste realizar a persistência. Informamos também a classe da chave primária, que neste caso será a classe que implementamos exclusivamente para esse propósito. A seguir vemos a definição da versão de CMP que no nosso caso utilizamos a versão 2.x e a seguir o nome abstrato do esquema, utilizado para as consultas EQL. Por fim definimos cada campo CMP. Estes campos devem ter os mesmos nomes dos contidos na implementação do EJB. A última linha define qual é o nome do atributo que será utilizado como chave para a persistência do objeto.

A configuração dos tipos de transação para cada método, segue a mesma configuração utilizada para os outros EJBs anteriores.

```
...
    <persistence- type>Container</persistence- type>
```

```
    <prim-key-class>com.book.project.ejb.entity.ProductPK</prim-
key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>Product</abstract-schema-name>
    <cmp-field>
       <field-name>name</field-name>
    </cmp-field>
    <cmp-field>
       <field-name>description</field-name>
    </cmp-field>
    <cmp-field>
       <field-name>price</field-name>
    </cmp-field>
    <cmp-field>
       <field-name>productPK</field-name>
    </cmp-field>
    <primkey-field>productPK</primkey-field>
</entity>
...
```

# 16.7 Criando um Message-Driven Bean

O próximo EJB a ser comentado será o EJB Message-Driven Bean - UserNotifierBean. Como vimos nos tópicos teóricos deste tipo de EJB, o mesmo não necessita da definição de interfaces como nos EJB Session Bean e EJB Entity Bean. O que devemos fazer é implementar a classe do EJB que servirá como consumidor de mensagens e a mensagem propriamente dita, sendo que esta última pode ser utilizada como uma das implementações definidas pela API Java.

Vamos analisar o EJB UserNotifierBean, que nos apresentará algumas diferenças dos outros exemplos apresentados anteriormente.

Vemos inicialmente que a classe deve implementar a interface MessageDrivenBean e a interface

MessageListener. A primeira interface é utilizada pelo container para tratar do ciclo de vida do EJB, tais como as interface SessionBean e EntityBean. A segunda interface é utilizada para definir a classe como um consumidor de mensagens. Nesta interface está declarado o método onMessage(), o qual veremos sua implementação logo a seguir.

```
...
public class UserNotifierBean implements MessageDrivenBean,
MessageListener {
...
```

Os EJBs do tipo Message-Driven Bean somente definem os métodos de ciclo de vida ejbCreate() e ejbRemove(). Os mesmos no nosso caso estão desprovidos de implementação, por isso não serão detalhados.

O método de negócio importante para a execução deste tipo de EJB é o método onMessage(). Este método é executado assim que mensagens são enviadas pelos produtores de mensagens, isto é, os clientes que realizam as chamadas assíncronas ao EJB. Estas mensagens são postadas em um MOM e logo em seguida consumidas pelo EJB Message-Driven Bean. Para isto, a mensagem é enviada como parâmetro para o método onMessage(), que valida a mensagem, recupera os dados contidos nela e realiza a lógica de negócio necessária.

No nosso exemplo, a mensagem é validada verificando se a mesma é uma instância do tipo de mensagem que estamos esperando receber. Caso isto ocorra, obtemos a mensagem através do método getMessate() da interface Message e realizamos a lógica de negócio, que neste caso será enviar um email para o usuário que efetuou a compra dos produtos. Observe que todas as mensagens customizadas, isto é, definidas para uma determinada aplicação, devem implementar a interface Message.

```
...
public void onMessage(javax.jms.Message msg) {
    try {
        if (msg instanceof UserMessage) {
```

```
            String message = ((UserMessage) msg).getMessage();
            UserVO user = ((UserMessage) msg).getUser();
            this.sendEmail(user.getHost(), user.getPartialEmail(),
                "Sales Notification", message);
        }
    }catch (Exception ex) {
        System.err.println("Error sending email for user. " + ex);
    }
}
...
```

Este EJB também apresenta um método privado chamado sendEmail(), no qual é implementado a lógica de envio de emails. Ele não será detalhado.

A classe que implementa a mensagem UserMessage também não será detalhada, pois não apresenta uma complexida extra. Somente define atributos que serão utilizados para carregar os dados da mensagem do produtor para o consumidor, isto é, do cliente que está efetuando a chamada assíncrona, para o EJB Message-Driven Bean que executará a operação.

A seguir vemos o deployment descriptor do EJB Message-Driven Bean - UserNotifierBean. Nele podemos observar algumas particularidades para este tipo de EJB. Como define somente a classe de implementação do EJB, nele definimos somente este atributo, além do nome de visualização e do nome utilizado pelo container para referenciar o EJB. O tipo de transação também é definida como nos outros tipos de EJB. A particularidade do EJB Message-Driven Bean está em definir o destino da mensagem. Devemos informar qual o tipo de destino utilizado para receber a mensagem, se é Queue ou Topic.

```
...
<message-driven>
    <display-name>UserNotifier</display-name>
    <ejb-name>UserNotifier</ejb-name>
    <ejb-class>com.book.project.ejb.mdb.UserNotifierBean</ejb-
class>
    <transaction-type>Container</transaction-type>
    <message-driven-destination>
```

```
        <destination- type>javax.jms.Topic</destination- type>
        <subscription- durability>NonDurable</subscription-
durability>
    </message- driven- destination>
</message- driven>
...
```

## 16.8 Empacotando a aplicação

Para empacotar a aplicação, podemos fazê-la de várias formas. Podemos utilizar os empacotadores do próprio J2SDK e J2SDKEE que são o jar e o packager respectivamente. Poderíamos utilizar do ANT, uma ferramenta muito utilizada no meio Java para criar makefiles.

Utilizaremos um arquivo de 'build' do ANT para realizar tal operação. Observe que o intuito deste tópico não é de explicar as particularidades dessa ferramenta. Para maiores detalhes veja no tópico Bibliografia as referências para o ANT.

A seguir apresentamos o arquivo build.xml, utilizado como makefile para compilar a aplicação, gerar documentação, criar os pacotes (jar) e a aplicação Enterprise (ear).

**Build File : build.xml**

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

<project name="exemplos" default="help" basedir=".">

    <property name="name"           value="${ant.project.name}"/>
    <property name="fullname"       value="Example"/>
    <property name="version"        value="1.0"/>
    <property name="build.num"          value="001"/>
    <property name="build.tag"      value=""/>
    <property name="compile.debug"      value="on"/>
    <property name="build.dir"      value="../tmp"/>
    <property name="build.classes"      value="${build.dir}/
classes"/>
    <property name="build.lib"      value="${build.dir}/lib"/>
```

```
    <property name="build.resources" value="${build.dir}/etc"/>
    <property name="build.packages" value="${build.dir}/
packages"/>
    <property name="build.doc"          value="${build.dir}/doc"/>
    <property name="build.javadoc"      value="${build.dir}/api"/>

    <property name="proj.dir"         value="../"/>
    <property name="proj.src"         value="${proj.dir}/src"/>
    <property name="proj.resources"  value="${proj.dir}/
etc/resources"/>
    <property name="proj.descriptors"   value="${proj.dir}/
etc/descriptors"/>
    <property name="proj.lib"         value="${proj.dir}/lib"/>
    <property name="proj.web"         value="${proj.dir}/src/web"/>
    <property name="doc.packages"       value="*"/>

    <!— configura o classpath baseado nas libs do projeto —>
    <path id="classpath">
       <pathelement path="${build.classes}"/>
       <fileset dir="${proj.lib}" includes="**/*.jar"/>
    </path>

    <!— mostra o help do build file —>
    <target name="help">
       <echo                        message="${fullname} Build
Script"/>
       <echo                        message=""/>
       <echo                        message="Use um dos
seguintes targets (alvos):"/>
       <echo                        message=""/>
       <echo                        message="help _____
    Mostra esta mensagem!"/>
       <echo                        message="clean_____
    Remove estrutura antiga de deploy"/>
       <echo                        message="build _____
    Empacota o sistema a partir dos arquivos locais"/ >
       <echo                        message="docs _____
    Gera o javadoc do projeto (API)"/>
       <echo                        message="compile _____
    Compila as classes do projeto"/>
       <echo                        message=""/>
```

```
</target>

<!— limpa a estrutura antiga de deploy —>
<target name=”clean”>
   <delete dir=”${build.dir}”/>
</target>

<!— inicializações básicas —>
<target name=”init”>
   <mkdir dir=”${build.dir}”/>
</target>

<!— inicializações necessárias para a compilação —>
<target name=”init- compile” depends=”init”>
   <mkdir dir=”${build.classes}”/>
   <mkdir dir=”${build.packages}”/>
   <mkdir dir=”${build.lib}”/>
</target>

<!— inicialização para o build da aplicação —>
<target name=”init- build” depends=”init”>
<!— resources da aplicação —>
<copy todir=”${build.resources}”>
   <fileset dir=”${proj.resources}”/>
</copy>
<!— libs do projeto —>
<copy todir=”${build.lib}”>
   <fileset dir=”${proj.lib}”>
   </fileset>
</copy>
</target>

<!— compila as classes da aplicação —>
<target name=”compile” depends=”init- compile”>
   <javac    srcdir=”${proj.src}” destdir=”${build.classes}”
         classpathref=”classpath” debug=”${compile.debug}”/>

</target>

<!— cria arquivos JARs —>
```

```
   <target name="jar">
      <!— empacota todos os arquivos de configurações do sistema
—>
      <jar jarfile="${build.packages}/resources.jar">
         <fileset dir="${build.resources}"/>
      </jar>
      <!— empacota as classes menos as do EJB —>
      <jar   jarfile="${build.packages}/example- client.jar"
             manifest="${proj.descriptors}/jar/MANIFEST.MF">
         <fileset dir="${build.classes}">
            <include name="com/book/project/client/**"/>
            <include name="com/book/project/ejb/session/
                  ProjectSupport.class"/>
            <include name="com/book/project/ejb/session/
                  ProjectSupportHome.class"/>
            <include name="com/book/project/ejb/session/
                  SalesBasket.class"/>
            <include name="com/book/project/ejb/session/
                  SalesBasketHome.class"/>
            <include name="com/book/project/ejb/session/
                  SalesSupport.class"/>
            <include name="com/book/project/ejb/session/
                  SalesSupportHome.class"/>
            <include
name="com/book/project/ejb/mdb/UserMessage.class"/>
         </fileset>
      </jar>
   </target>

   <!— cria o war da aplicação —>
   <target name="war">
      <war  warfile="${build.packages}/example- web.war"
            webxml="${proj.web}/WEB- INF/web.xml">
         <fileset dir="${proj.web}" excludes="WEB- INF/web.xml"/>
      </war>
   </target>

   <!— empacota os ejbs da aplicação exemplo —>
   <target name="ejb- jar">
      <jar   jarfile="${build.packages}/example- ejb.jar"
             manifest="${proj.descriptors}/ejb/MANIFEST.MF">
```

```
            <fileset dir="${build.classes}" >
                <include name="com/book/project/vo/**"/>
                <include name="com/book/project/ejb/**"/>
            </fileset>
        </jar>
    </target>

    <!— cria o arquivo ear —>
    <target name="ear" depends="jar, war, ejb- jar">
        <ear earfile="${build.packages}/example- ear.ear"
          appxml="${proj.descriptors}/ear/application.xml" >
            <fileset dir="${build.packages}">
                <include name="example- ejb.jar"/>
                <include name="example- web.war"/>
                <include name="example- client.jar"/>
                <include name="resources.jar"/>
            </fileset>
        </ear>
    </target>

    <!— faz o build da aplicação —>
    <target name="build" depends="clean, init- build, compile, ear">
    </target>

    <!— gera o javadoc da aplicação —>
    <target name="docs" depends="compile">
        <mkdir dir="${build.doc}"/>
        <mkdir dir="${build.javadoc}"/>
        <javadoc sourcepath="${proj.src}" destdir="${build.javadoc}"
              classpathref="classpath" access="private"
version="yes" use="yes"
              Splitindex="yes" Windowtitle="${fullname} API
Specification"
              Doctitle="${fullname} API Specification"
Header="${fullname} -
              Build ${build.num}" bottom="${fullname} - Javadoc">
            <package name="com.book.*"/>
            <link href="http://java.sun.com/j2se/1.3/docs/api"/>
            <link
href="http://java.sun.com/j2ee/sdk_1.3/techdocs/api"/>
        </javadoc>
```

```
        <zip zipfile="${build.dir}/${fullname}-javadoc.zip">
            <fileset dir="${build.javadoc}"/>
        </zip>
    </target>

</project>
```

## 16.9 Instalando a aplicação no servidor J2EE

A instalação de uma aplicação J2EE no servidor de aplicação Jboss não é uma tarefa muito difícil nem complexa. Configurações adicionais no próprio servidor de aplicação não serão necessárias, pois não utilizamos recursos adicionais, mas se isso fosse preciso também não seria uma operação muito complexa.

Tendo em mãos o arquivo EAR, a única tarefa que devemos realizar é a cópia do Enterprise Archive no diretório de deploy do Jboss. Devemos nos certificar de executar o Jboss com as configurações padrões, isto é, executar o comando <Dir_Instalação_Jboss>/bin/run ou <Dir_Instalação_Jboss>/bin/run =c default, que inicializará o servidor em seu modo de configuração padrão. Sendo assim, podemos realizar o deploy da aplicação, somente copiando o arquivo para o diretório <Dir_Instalação_Jboss>/server/default/deploy.

Algumas informações sobre o deploy serão impressas no console do Jboss, detalhando a instalação do pacote. Por fim, se nenhum problema no arquivo EAR for detectado pelo servidor de aplicação, a mensagem de deploy realizado com sucesso será impressa no console.

Observe que o servidor de aplicações Jboss pode ser configurado para inicializar com serviços e recursos a mais ou a menos. A instação padrão sem configurações adicionais, nos apresenta três modos de configuração fornecidos pelo servidor de aplicação. A configuração recomendável a ser utilizada para a execução da aplicação exemplo deste livro é a configuração default. Para maiores

detalhes sobre as outras configurações ou configurações customizadas, consulte a documentação do servidor de aplicação.

# Apêndice A

# Deployment Descriptor

## A.1 O que é um deployment descriptor?

Deployment descriptors ou arquivos descritores são utilizados pelos componentes da arquitetura J2EE para definir seus detalhes, informações que são usadas pelo container para gerenciar esses objetos.

Existem vários tipos de descritores, um diferente do outro e específicos para cada tipo de componente.

São alguns exemplos de descritores:

web.xml para componentes Web;

ejb-jar.xml para componentes EJB;

application.xml para o empacotamente de aplicações enterprise etc.

Para componentes EJB por exemplo, se faz necessário o uso de um arquivo denominado ejb-jar.xml, especificado pela Sun MicroSystems Inc. para aplicações enterprise e que apresenta informações dos componentes EJB tais como Session Bean, Entity Bean e Message-Driven Bean. Além disso, apresenta formas de definir o escopo da transação na qual irá sobreviver o EJB, segurança e autenticação, relacionamentos entre Entity Beans, tipo de sessão, persistência dos Entity Beans, consultas em EQL

para Entity Beans, tipo de Session Bean, recursos remotos ou locais localizados através de JNDI entre outros.

A seguir veremos mais detalhes do deployment descriptor ejb-jar.xml.

## A.2 Elementos do deployment descriptor ejb-jar.xml

O deployment descriptor de um pacote de componentes EJB, deve ter o nome de ejb-jar.xml e estar no diretório META-INF da aplicação. Este arquivo será empacotado juntamente com as classes que os componentes utilizarão.

Este arquivo tem sua estrutura definida por um DTD (Document Type Definition). Iremos apresentar uma breve explicação de cada elemento e atributo do deployment descriptor ejb-jar.xml. Para maiores delhaes, consulte o DTD ejb-jar_2_0.dtd.

O arquivo em questão, deve ter como linha inicial com a seguinte declaração:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0
    //EN" 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
```

### A.2.1 <ejb-jar>

O elemento-raiz do deployment descriptor ejb-jar.xml é o elemento <ejb-jar>. A seguir iremos detalhar seus atributos e elementos.

```
<ejb-jar>
    <description></description>
    <display-name></display-name>
    <small-icon></small-icon>
    <large-icon></large-icon>
    <enterprise-beans></enterprise-beans>
    <relationships></relationships>
    <assembly-descriptor></assembly-descriptor>
    <ejb-client-jar></ejb-client-jar>
```

```
</ejb-jar>
```

**<description>** : atributo que apresenta a descrição do pacote EJB.

**<display-name>** : atributo com nome do módulo de EJB a ser utilizado por outras ferramentas para referenciar a este pacote.

**<small-icon>** : atributo com o caminho completo de uma figura que será utilizada como um ícone pelas ferramentas, que manipularão este módulo. 16x16 pixels.

**<large-icon>** : atributo com o o mesmo que a small-icon, sendo que esta ícone poderá ser maior. 32x32 pixels.

**<enterprise-beans>** : elemento que define as informações dos EJBs contidos neste pacote. Neste elemento estarão as informações de cada EJB Session Bean, Entity Bean e Message-Driven Bean.

**<relationships>** : elemento que apresenta os relacionamentos entre os EJB Entity Bean CMP.

**<assembly-descriptor>** : elemento que define as informações de segurança e transações.

**<ejb-client-Jar>** : elemento usado para conter o nome do pacote com as classe para acesso remoto aos EJBs.

### A.2.2 <enterprise-beans>

O elemento <enterprise-beans> define informações de cada tipo de EJB, tais como, Session Bean, Entity Bean e Message-Driven Bean, e não possui atributos. A seguir iremos detalhar seus elementos.

```
<enterprise-beans>
   <session></session>
   <entity></entity>
   <message-driven></message-driven>
</enterprise-beans>
```

O elemento <session> define informações para os EJBs Session Bean contidos no arquivo de deploy, sendo assim, podemos ter várias ocorrências deste elemento. A sua declaração apresenta alguns atributos e elementos.

```
<session>
   <description></description>
   <display-name></display-name>
   <small-icon></small-icon>
   <large-icon></large-icon>
   <ejb-name></ejb-name>
   <home></home>
   <remote></remote>
   <local-home></local-home>
   <local></local>
   <ejb-class></ejb-class>
   <session-type></session-type>
   <transaction-type></transaction-type>
   <env-entry> </env-entry>
   <ejb-ref></ejb-ref>
   <ejb-local-ref></ejb-local-ref>
   <security-role-ref></security-role-ref>
   <security-identity></security-identity>
   <resource-ref></resource-ref>
   <resource-env-entry></resource-env-entry>
</session>
```

**<description>**: descrição do EJB Session Bean.

**<display-name>**: nome utilizado pelo container para se referir ao EJB.

**<small-icon>**: caminho completo de uma figura que será utilizada como um ícone pelas ferramentas, para este EJB.

**<large-icon>**: caminho completo de uma figura que será utilizada como um ícone pelas ferramentas, para este EJB.

**<ejb-name>**: nome do EJB, utilizado posteriomente dentro do próprio deployment descriptor para referenciar este EJB.

**<home>**: nome completo da interface Home do EJB.

**<remote>**: nome completo da interface Remote do EJB.

**<local-home>**: nome completo da interface LocalHome do EJB.

**<local>**: nome completo da interface Local do EJB.

**<ejb-class>**: nome completo da classe que implementa o EJB.

**<session-type>**: tipo de EJB Session Bean. Pode ser Stateless ou Stateful.

**<transaction-type>**: define quem gerenciará a transação. Pode ser Bean ou Container.

**<env-entry>**: define propriedades de ambiente para o EJB.

**<ejb-ref>**: declara referências para outros EJBs.

**<ejb-local-ref>**: declara referências locais para outros EJBs.

**<security-role-ref>**: declara referência para regras de segurança para este EJB.

**<security-identity>**: informa como propagar o contexto de segurança.

**<resource-ref>**: declara referência para recursos que podem ser utilizados pelos EJBs.

**<resource-env-entry>**: associa os recursos com nome JNDI.

O elemento <entity> apresenta informações do EJB Entity Bean e também pode ocorrer mais de uma vez no arquivo de deploy. Além de apresentar informações do EJB, pode definir as querys utilizadas pelo container para um EJB CMP.

```
<entity>
    <description></description>
    <display-name></display-name>
    <small-icon></small-icon>
    <large-icon></large-icon>
```

```
    <ejb-name></ejb-name>
    <home></home>
    <remote></remote>
    <local-home></local-home>
    <local></local>
    <ejb-class></ejb-class>
    <persistence-type></persistence-type>
    <prim-key-class></prim-key-class>
    <reentrant></reentrant>
    <cmp-version></cmp-version>
    <abstract-schema-name></abstract-schema-name>
    <cmp-field></cmp-field>
    <primkey-field></primkey-field>
    <env-entry> </env-entry>
    <ejb-ref></ejb-ref>
    <ejb-local-ref></ejb-local-ref>
    <security-role-ref></security-role-ref>
    <security-identity></security-identity>
    <resource-ref></resource-ref>
    <resource-env-entry></resource-env-entry>
    <query></query>
</entity>
```

**<description>**: descrição do EJB Entity Bean.

**<display-name>**: nome utilizado pelo container para se referir ao EJB em questão.

**<small-icon>**: caminho completo de uma figura que será utilizada como um ícone pelas ferramentas, para este EJB.

**<large-icon>**: caminho completo de uma figura que será utilizada como um ícone pelas ferramentas, para este EJB.

**<ejb-name>**: nome do EJB, utilizado posteriomente dentro do próprio deployment descriptor para referenciar este EJB.

**<home>**: nome completo da interface Home do EJB.

**<remote>**: nome completo da interface Remote do EJB.

**<local-home>**: nome completo da interface LocalHome do EJB.

**<local>**: nome completo da interface Local do EJB.

**<ejb-class>**: nome completo da classe que implementa o EJB.

**<persistence-type>**: tipo de persistência do objeto. Pode ser BMP ou CMP.

**<prim-key-class>**: nome completo da classe que define a chave primária da entidade.

**<reentrant>** : pode ser True ou False.

**<cmp-version>**: versão de CMP. Depende da implementação do servidor de aplicação. Pode ser 1.x ou 2.x.

**<abstract-schema-name>**: nome do objeto utilizado adiante para criar em EQL, formas de consulta e manipulação do objeto em questão.

**<cmp-field>**: apresenta todos os campos de um EJB Entity Bean CMP que serão persistidos.

**<primkey-field>**: campo de um EJB Entity Bean CMP que será considerado como chave primária.

**<env-entry>**: define propriedades de ambiente para o EJB.

**<ejb-ref>**: declara referências para outros EJBs.

**<ejb-local-ref>**: declara referências locais paa outros EJBs.

**<security-role-ref>**: declara referência para regras de segurança para este EJB.

**<security-identity>**: informa como propagar o contexto de segurança.

**<resource-ref>**: declara referência para recursos que podem ser utilizados pelos EJBs.

**<resource-env-entry>**: associa os recursos com nome JNDI.

**<query>**: apresenta uma lista de códigos de manipulação de objetos, utilizado para EJB Entity Bean CMP, utilizando a linguagem EQL (EJB Query Language).

O elemento <message-driven> define as informações dos EJB Message-Driven Bean. Como nos outros casos, o arquivo de deploy pode conter mais de uma ocorrência deste elemento, isto é, no caso de deploy de mais de um EJB. A seguir os seus atributos e elementos.

```
<message-driven>
   <description></description>
   <display-name></display-name>
   <small-icon></small-icon>
   <large-icon></large-icon>
   <ejb-name></ejb-name>
   <ejb-class></ejb-class>
   <transaction-type ></transaction-type >
   <message-selector></message-selector>
   <acknowledge-mode></acknowledge-mode>
   <message-driven-destination></message-driven-destination>
   <env-entry> </env-entry>
   <ejb-ref></ejb-ref>
   <ejb-local-ref></ejb-local-ref>
  <security-identity></security-identity>
   <resource-ref></resource-ref>
   <resource-env-entry></resource-env-entry>
</message-driven>
```

**<description>**: descrição do EJB Message-Driven Bean.

**<display-name>**: nome utilizado pelo container para se referir ao EJB.

**<small-icon>**: caminho completo de uma figura que será utilizada como um ícone pelas ferramentas, para este EJB.

**<large-icon>**: caminho completo de uma figura que será utilizada como um ícone pelas ferramentas, para este EJB.

**<ejb-name>**: nome do EJB, utilizado posteriomente dentro do prórprio deployment descriptor para referenciar este EJB.

**<ejb-class>**: nome completo da classe que implmenta o EJBem questão.

**<transaction-type>**: define quem gerenciará a transação. Pode ser Bean ou Container.

**<message-selector>**: filtra mensagems baseadas no seletor de strings do JMS.

**<acknowledge-mode>**: especifica como serão as mensagens de confirmação (acknowledge). Pode ser Auto-acknowledge ou Dups-ok-acknowledge.

**<message-driven-destination>**: define o tipo de destino da mensagem que pode ser javax.jms.Queue ou javax.jms.Topic. Além de definir a durabilidade do envio da mensagem que pode ser Durable ou NonDurable.

**<env-entry>**: define propriedades de ambiente para o EJBem questão.

**<ejb-ref>**: declara referências para outros EJBs.

**<ejb-local-ref>**: declara referências locais paa outros EJBs.

**<security-identity>** : informa como propagar o contexto de segurança.

**<resource-ref>** : declara referência para recursos que podem ser utilizados pelos EJBs.

**<resource-env-entry>** : associa os recursos com nome JNDI.

### A.2.3 <relationships>

O elemento <relationships> apresenta o relacionamento entre EJBs Entity Bean CMP, sendo que podem ocorrer nenhum, um ou mais relacionamentos entre EJBs deste tipo. Define outro elemento, <ejb-relation>, no qual são configurados cada relacionemento entre dois EJBs.

```
<relationships>
<description></description>
<ejb- relation>
   <description></description>
   <ejb- relation- name></ejb- relation- name>
   <ejb- relationship- role>
      <description></description>
      <ejb- relationship- role- name></ejb- relationship- role-
name>
      <multiplicity></multiplicity>
      <relationship- role- source>
         <description></description>
         <ejb- name></ejb- name>
      </relationship- role- source>
      <cmr- field>
         <description></description>
         <cmr- field- name></cmr- field- name>
         <cmr- field- type></cmr- field- type>
      </cmr- field>
      </ejb- relationship- role>
   <ejb- relationship- role>
   <description></description>
   <ejb- relationship- role- name></ejb- relationship- role- name>
   <multiplicity></multiplicity>
   <cascade- delete></cascade- delete>
   <relationship- role- source>
      <description></description>
      <ejb- name></ejb- name>
   </relationship- role- source>
   <cmr- field>
      <description></description>
      <cmr- field- name></cmr- field- name>
      <cmr- field- type></cmr- field- type>
   </cmr- field>
   </ejb- relationship- role>
</ejb- relation>
<relationships>
```

A seguir vemos uma breve explicação dos atributos do elemento <relationships>.

**<description>** : descrição do relacionamento entre EJB Entity Bean CMP.

**<ejb- relation>** : define o relacionamento entre dois EJB Entity Bean CMP.

Sub-elemento <ejb- relation> do elemento <relationships> que define define o relacionamento entre dois EJB Entity Bean CMP.

**<description>**: descrição do relaciomento entre dois EJB Entity Bean CMP.

**<ejb- relation- name>**: nome do relacionemnto.

**<ejb- relationship- role>**: elemento que deve ser configurado para cada um dos dois EJBs que possuem o relacionemnto.

Sub-elemento <ejb- relationship- role> do elemento <ejb- relation> que descreve um regra de relacionamento dentre dois EJBs.

**<description>**: descrição da regra de relacionamento.

**<ejb- relationship- role- name>**: nome da regra de relacionamento.

**<multiplicity>**: multiplicidade do relacionamento. Pode ser One ou Many.

**<cascade- delete>**: define se o relacionamento de um para muitos de um EJB Entity Bean CMP é dependente da vida do outro EJB. Sendo assim, só pode ser configurado para o EJB Entity Bean com multiplicidade um (One) no relacionamento.

**<relationship- role- source>**: define qual EJB estará relacionando com este.

**<cmr- field>** : define qual o campo utilizado para o relacionamento. Apresenta atributos que definem a descrição do campo de relaciomento, o nome do campo propriamente dito e o tipo do campo.

### A.2.4 <assembly- descriptor>

O elemento <assembly-descriptor> define as regras de segurança, a permissão de acesso (execução) dos métodos dos EJBs, atributos de transação para cada método de cada EJBe a lista dos métodos excluídos do deploy.

```
<assembly-descriptor>
   <security-role>
      <description />
      <role-name></role-name>
   </security-role>
   <method-permission>
      <role-name></role-name>
   <method>
      <description />
      <ejb-name></ejb-name>
      <method-intf></method-intf>
      <method-name></method-name>
      <method- params ></method-params>
   </method>
   </method-permission>
   <container-transaction>
   <description />
   <method>
      <ejb-name></ejb-name>
      <method-name></method-name>
   </method>
   <trans-attribute></trans-attribute>
   </container-transaction>
</assembly-descriptor>
```

**<security-role>** : define a regra de segurança utilizada para acesso aos EJBs. Como atributos define uma descrição da regra de segurança e o nome da regra propriamente dita.

**<method-permission>** : quais métodos podem ser acessados por quem a partir da regra de segurança definida em <security-role>. Para cada método, informa-se o nome do EJB que o contém, além de informar se este método é acessado para alguma interface, o seu nome e os parâmetros que ele recebe.

Pode-se definir uma regra para todos os métodos de um EJB, utilizando o asterisco (*) no elemento <method-name>, por exemplo. Este elemento pode aparecer quantas vezes for necessário, para configurar uma regra de segurança para um método de um EJB.

**<container-transaction>** : define atributos de transação para cada método de cada EJB. Pode conter um atributo que descreve a transação e deve conter para cada método o nome do EJB e o nome do método que será utilizado um tipo de transação, configurada no elemento <trans-attribute> .

**<exclude-list>** : lista dos métodos excluídos do deploy. Apresenta um atributo com a descrição da lista de exclusão e um elemento que contém o nome de cada método de cada EJB que será excluído do deploy.

# Apêndice B

# API Enterprise JavaBeans

Este apêndice apresenta um guia de referência rápida para a API Enterprise JavaBeans. Observe que a API da plataforma J2EE é muita mais extensa, sendo que esta apresentada neste tópico é somente uma parte dela. Não visamos também explicar em detalhes a API EJB. Para detalhes consulte a documentação oficial da API e a especificação EJB 2.0.

Iremos apresentar o pacote javax.ejb explicando as suas interfaces e exceções.

## B.1 Interfaces

### B.1.1 EJBContext

A interface EJBContext é extendida pelos contextos de cada tipo de EJB, isto é, pelas interfaces SessionContext, EntityContext e MessageDrivenContext. Fornece informações de segurança, transação e dados sobre as interfaces Home (Local e Remote) do EJB, sendo possível utilizá-la para criar, destruir ou localizar um objeto EJB por exemplo.

```
public interface EJBContext {
    EJBHome getEJBHome();
    EJBLocalHome getEJBLocalHome();
    Properties getEnvironment();
    Identity getCallerIdentity();
```

```
    Principal getCallerPrincipal();
    boolean isCallerInRole(Identity role);
    boolean isCallerInRole(String roleName);
    UserTransaction getUserTransaction() throws
IllegalStateException;
    void setRollbackOnly() throws IllegalStateException;
    boolean getRollbackOnly() throws IllegalStateException;
}
```

### B.1.2 EJBHome

Esta interface deve ser extendida pelas interfaces Home (acesso Remoto - Home) dos componentes EJBs implementados. Ela oferece serviços para criação, remoção e localização de objetos EJB. Oferece informações sobre o metadados do EJB e o handle do EJB, que pode ser utilizado para futuras chamadas de métodos ao objeto, sem a necessidade de realizar a sua localização por exemplo.

```
public interface EJBHome extends Remote {
   void remove(Handle handle) throws RemoteException,
RemoveException;
   void remove(Object primaryKey) throws RemoteException,
RemoveException;
   EJBMetaData getEJBMetaData() throws RemoteException;
   HomeHandle getHomeHandle() throws RemoteException;
}
```

### B.1.3 EJBLocalHome

Deve ser extendida pelas interfaces Home (acesso Local - LocalHome) dos componentes EJBs. Apresenta serviços de criação, remoção e localização dos objetos EJB.

```
public interface EJBLocalHome {
    void remove(Object primaryKey) throws RemoveException,
EJBException;
}
```

### B.1.4 EJBLocalObject

A interface EJBLocalObject deve ser extendida pela interface Local (interface Remote - acesso Local) para os EJBs que provêm acessos locais. Apresenta métodos para obter referência para a interface LocalHome e obter a chave-primária (Primary Key) do objeto caso o mesmo seja um EJB Entity Bean. Além de prover um método para destruir a instância do objeto e um serviço que avalia se o objeto atual, isto é, o EJBLocalObject é idêntico a outro informado.

Quando definir a interface Local para um componente EJB, a mesma deve conter os métodos de negócio que estarão disponíveis para o cliente local.

```
public interface EJBLocalObject {
    public EJBLocalHome getEJBLocalHome() throws EJBException;
    public Object getPrimaryKey() throws EJBException;
    public void remove() throws RemoveException, EJBException;
    boolean isIdentical(EJBLocalObject obj) throws EJBException;
}
```

### B.1.5 EJBMetaData

Permite que o cliente acesse informações dos metadados do EJB. Esta interface não é muito utilizada, mas pode ser obtida através da chamada ao método ejbHome.getEJBMetaData(). As informações contidas nesta interface e fornecidas ao cliente remoto em forma de um objeto serializado, pode ser utilizado para obter dinamicamente informações sobre o EJB.

```
public interface EJBMetaData {
    EJBHome getEJBHome();
    Class getHomeInterfaceClass();
    Class getRemoteInterfaceClass();
    Class getPrimaryKeyClass();
    boolean isSession();
    boolean isStatelessSession();
}
```

### B.1.6 EJBObject

Esta interface deve ser extendida pela interface Remote (acesso remoto) dos componentes EJBs. Apresenta uma visão dos serviços oferecidos pelo EJB para o cliente remoto. A interface Remote definida para cada EJB, deve conter as assinaturas dos métodos de negócio que o EJB implementa e as quais o cliente remoto terá acesso.

Por esta interface, o cliente pode ter acesso a referência para a interface Home do EJB, obter a chave-primária no caso de EJB Entity Bean, remover a instância do objeto, obter o Handler deste objeto que contém informações sobre ele e verificar se algum objeto é idêntico ao objeto atual.

```
public interface EJBObject extends Remote {
    public EJBHome getEJBHome() throws RemoteException;
    public Object getPrimaryKey() throws RemoteException;
    public void remove() throws RemoteException, RemoveException;
    public Handle getHandle() throws RemoteException;
    boolean isIdentical(EJBObject obj) throws RemoteException;
}
```

### B.1.7 EnterpriseBean

A interface EnterpriseBean é a interface genérica de cada tipo de EJB. Ela é extendida pelas interfaces SessionBean, EntityBean e MessageDrivenBean. Além de ser serializada é utilizada como uma interface marker, isto é, informa que a interface é realmente um EJB.

```
public interface EnterpriseBean extends java.io.Serializable {
}
```

### B.1.8 EntityBean

Quando implementamos EJB EntityBean, devemos implementar esta interface. Apresenta os métodos de ciclo de vida do EJB, executados pelo container nos momentos apropriados.

Define os métodos para configurar e desconfigurar o contexto associado ao EJB Entity Bean, método de remoção

da entidade persistente, ativação da entidade caso tenha sido passivada pelo container, passivação da entidade em caso de falta de recursos ou porque muitos EJBs estão instanciados por exemplo, carrega uma entidade do meio persistente para a memória quando o container precisa atualizar os dados do objeto, ou atualiza os dados no meio persistente de acordo com os valores dos atributos constantes em memória.

Além destes métodos, se faz necessário a implementação do método ejbCreate() já mencionado anteriormente, o qual irá criar a instância do objeto em memória e persistir o mesmo.

```
public interface EntityBean extends EnterpriseBean {
    public void setEntityContext(EntityContext ctx)
            throws EJBException, RemoteException;
    public void unsetEntityContext() throws EJBException,
RemoteException;
    public void ejbRemove() throws RemoveException, EJBException,
RemoteException;
    public void ejbActivate() throws EJBException, RemoteException;
    public void ejbPassivate() throws EJBException, RemoteException;
    public void ejbLoad() throws EJBException, RemoteException;
    public void ejbStore() throws EJBException, RemoteException;
}
```

### B.1.9 EntityContext

Apresenta a interface específica do EJBContext para um EJB Entity Bean. Ela é associada ao objeto após a criação da sua instância.

```
public interface EntityContext extends EJBContext {
    EJBLocalObject getEJBLocalObject() throws IllegalStateException;
    EJBObject getEJBObject() throws IllegalStateException;
    Object getPrimaryKey() throws IllegalStateException;
}
```

### B.1.10 Handle

Um Handle é uma referência persistente de um componente EJB. Ela é implementada por todos os EJB Handles. Muito útil quando a aplicação necessita persistir a referência para um objeto EJB e recuperá-la posteriormente.

```
public interface Handle extends java.io.Serializable {
    public EJBObject getEJBObject() throws RemoteException;
}
```

### B.1.11 HomeHandle

Tal como o Handle é uma referência persistente para a interface Remote de um EJB, a HomeHandle é uma referência persistente para interface Home de um EJB. Também é útil para manter a referência da interface Home de um EJB, persistente em algum meio e recuperá-la posteriomente para acessar a interface Home.

```
public interface HomeHandle extends java.io.Serializable {
    public EJBHome getEJBHome() throws RemoteException;
}
```

### B.1.12 MessageDrivenBean

Esta interface deve ser implementada por cada EJB Message-Driven Bean. Define os métodos de ciclo de vida deste tipo de EJB. Observe que estes métodos não são executados pelos clientes, pois como este componente tem um comportamento de execução assíncrona por meio de mensagens (JMS), os métodos são executados pelo container no recebimento de mensagens dos clientes.

Observe que o único método de negócio que a implementação de um EJB Message-Driven Bean deve oferecer é o método onMessage(), implementado da API JMS. Este método será invocado pelo container no recebimento de mensagens dos clientes e deve executar a lógica de negócio para o EJBem questão.

```
public interface MessageDrivenBean extends EnterpriseBean {
```

```
    void setMessageDrivenContext(MessageDrivenContext ctx) throws
EJBException;
    void ejbRemove() throws EJBException;
}
```

### B.1.13 MessageDrivenContext

Apresenta a interface específica de EJBContext para um EJB Message-Driven Bean. Também associa o contexto ao EJB, depois que sua instância é criada.

```
public interface MessageDrivenContext extends EJBContext {
}
```

### B.1.14 SessionBean

Deve ser implementada por todo EJB Session Bean. Apresenta os métodos do ciclo de vida do EJB, tais como associação do contexto ao EJB, remoção da instância do objeto, ativação e passivação do objeto pelo container, além de definir o método ejbCreate() que criará uma instância do objeto pelo container.

```
public interface SessionBean extends EnterpriseBean {
    void setSessionContext(SessionContext ctx) throws EJBException,
RemoteException;
    void ejbRemove() throws EJBException, RemoteException;
    void ejbActivate() throws EJBException, RemoteException;
    void ejbPassivate() throws EJBException, RemoteException;
}
```

### B.1.15 SessionContext

Esta é a interface específica de EJBContext para um EJB Session Bean. Também é associada a instância do objeto após a sua criação.

```
public interface SessionContext extends EJBContext {
    EJBLocalObject getEJBLocalObject() throws IllegalStateException;
    EJBObject getEJBObject() throws IllegalStateException;
}
```

### B.1.16 SessionSyncronization

Esta interface pode ser implementada por EJB Session Bean Stateful, quando deseja-se sincronizar o estado transacional do objeto. Cada método é executado a cada etapa das fases de transação.

```
public interface SessionSynchronization {
    public void afterBegin() throws EJBException, RemoteException;
    public void beforeCompletion() throws EJBException,
RemoteException;
    public void afterCompletion(boolean committed) throws
EJBException,
            RemoteException;
}
```

## B.2 Exceções

### B.2.1 AccessLocalException

Exceção lançada caso o chamador não tem acesso para executar o método. Utilizado para objetos locais.

### B.2.2 CreateException

Deve ser definida no método de criação da instância de cada EJB. Ela é lançada quando ocorrer uma falha na criação da instância de um objeto EJB.

### B.2.3 DuplicateKeyException

Esta exceção é lançada quando não pode-se criar um objeto, pois a sua chave já existe e está sendo utilizada por outro objeto. Normalmente é utilizada no método create() do EJB Entity Bean.

### B.2.4 EJBException

Informa ao objeto chamador que ocorreu uma falha não recuperável, ou um erro inesperado. É lançada quando

ocorre erros deste tipo nos métodos de ciclo de vida ou nos métodos de negócio. É a única exceção do pacote javax.ejb que extende a exceção de runtime - RuntimeException - e não permite ao chamador se recuperar do erro. As exceções NoSuchEntityException, NoSuchObjectLocalException, AccessLocalException e TransactionRequiredLocalException e TransactionRolledbackLocalException extendem esta exceção.

### B.2.5 FinderException

Deve ser utilizada em todos os métodos finders, normalmente usados em EJB Entity Bean. São lançadas quando um erro de localização do objeto ocorre.

### B.2.6 NoSuchEntityException

É lançada pelo container quando em um EJB Entity Bean é executado um método, para o qual não existe o objeto em questão. Pode ser utilizada pelos métodos de negócio do EJB e pelos métodos de ciclo de vida ejbLoad() e ejbStore() de um EJB Entity Bean.

### B.2.7 NoSuchObjectLocalException

Parecida com a exceção NoSuchEntityException, diferindo que esta é lançada para objeto locais que não exitem mais.

### B.2.8 ObjectNotFoundException

É lançada por um método finder, para avisar que um objeto não existe. Deve-se utilizar esta exceção quando um método finder retornar somente um objeto. No caso de vários objetos, deve-se retornar uma coleção nula.

### B.2.9 RemoveException

A exceção RemoveException é lançada quando o container não pode remover a instância do objeto. É utilizada no método ejbRemove().

### B.2.10 TransactionRequiredLocalException

Informa que a requisição não encontrou uma transação e a mesma era requerida.

### B.2.11 TransactionRolledbackLocalException

Indica que a transação foi marcada com rollback ou estava sendo executado o rollback, mas ocorreu algum erro nesta operação.

# Apêndice C
# Aplicação J2EE – Exemplo

A seguir apresentamos o código completo da aplicação exemplo. Esta aplicação consiste de cinco EJBs, sendo um para cada tipo de EJB definido pela especificação EJB 2.0. Então serão:

EJB Session Bean Stateless - SalesSupportBean

EJB Session Bean Stateful - SalesBasketBean

EJB Entity Bean BMP - UserBean

EJB Entity Bean CMP - ProductBean

Message-Driven Bean - UserNotifierBean

Observe que existe um deployment descriptor ejb-jar.xml para cada EJB nesta aplicação exemplo. Isto não é necessário nem possível em uma aplicação J2EE, na qual façam parte dela todos esses EJBs. Para isso é preciso criar um único deployment descritor ejb-jar.xml, no qual incluímos todas as informações de todos os EJBs contidos na aplicação J2EE em questão. Este arquivo é mostrado no final deste apêndice.

Iremos apresentar também o arquivo específico para o servidor de aplicações Jboss, utilizado para criar as tabelas do banco de dados utilizado para persistir os objetos da aplicação. Este arquivo, chamado de jbosscmp-jdbc.xml está mostrado no final deste apêndice também. Salientamos que este arquivo não faz parte da especificação J2EE e é um arquivo complementar e de uso exclusivo para o servidor de aplicações Jboss.

Vejamos o código completo do EJB Sesison Bean Stateless - SalesSupportBean. Apresentamos as interfaces Home e Remote, SalesSupportHome e SalesSupport respectivamente, a implementação do EJB SalesSupportBean e por fim o deployment descriptor ejb-jar.xml deste EJB.

### Interface Home : SalesSupportHome.

```
package com.book.project.ejb.session;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

// Inteface Home, obtém uma referência para a interface Remote, esta
// ultima que fornece os serviços de suporte a venda dos produtos.
public interface SalesSupportHome extends javax.ejb.EJBHome {
    public SalesSupport create() throws CreateException, RemoteException;
}
```

### Interface Remote : SalesSupport.

```
package com.book.project.ejb.session;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

// Interface Remote do EJB <code>SalesSupport</code>.
// Apresenta serviços de cálculo de desconto e parcelas para um produto em venda.
public interface SalesSupport extends javax.ejb.EJBObject {
    public java.lang.Integer calcDiscount(Integer value, Integer range)
            throws RemoteException;
    public java.lang.Integer calcPiece(Integer value, Integer times)
            throws RemoteException;
}
```

### Session Bean Stateless : SalesSupportBean.

```
package com.book.project.ejb.session;
```

```
import javax.ejb.*;

// Apresenta os serviços para cálculo de frete, desconto, valores de parcela
// dependendo da quantidade de vezes que o valor vai ser pago.
// Estes serviços são utilizados enquanto estiver sendo realizada a venda dos produtos.
public class SalesSupportBean implements SessionBean {

    //Contexto do Session Bean.
    private SessionContext sessionContext;

    // Cria uma instância do objeto pelo container.
    // @throws CreateException exceção na criação da instãncia do objeto.
    public void ejbCreate() throws CreateException {
    }

    // Remoção da instância do objeto pelo container.
    public void ejbRemove() {
    }

    // Utilizado pelo container para ativar o objeto que está passivo.
    public void ejbActivate() {
    }

    // Utilizado pelo container para tornar passivo um objeto ativo.
    public void ejbPassivate() {
    }

    // Configura o contexto do EJB Session Bean.
    // @param sessionContext contexto do EJB Session Bean.
    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }

    // Calcula um desconto para o valor informado, de acordo com a percentual informada.
    // @param value valor a ser calculado o desconto.
```

```
    // @param perc percentual utilizado para cálculo do desconto.
    // @return valor do desconto concedido.
    public java.lang.Integer calcDiscount(Integer value, Integer perc) {
        return new Integer(value.intValue() * perc.intValue());
    }

    // Cálculo o valor das parcelas, de acordo com o valor total
informado e com
    // a quantidade de vezes também informada.
    // @param value valor total da venda.
    // @param times quantidade de parcelas desejada.
    // @return valor de cada parcela.
    public java.lang.Integer calcPiece(Integer value, Integer times) {
        return new Integer(value.intValue() / times.intValue());
    }
}
```

## Deployment Descriptor : SalesSupportBean.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-
jar_2_0.dtd">
<ejb-jar>
    <enterprise-beans>
        <session>
            <display-name>SalesSupport</display-name>
            <ejb-name>SalesSupport</ejb-name>

<home>com.book.project.ejb.session.SalesSupportHome</home>

<remote>com.book.project.ejb.session.SalesSupport</remote>
            <ejb-
class>com.book.project.ejb.session.SalesSupportBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
        </enterprise-beans>
        <assembly-descriptor>
            <container-transaction>
                <method>
                    <ejb-name>SalesSupport</ejb-name>
```

```
            <method- name>*</method- name>
        </method>
        <trans- attribute>Required</trans- attribute>
    </container- transaction>
</assembly- descriptor>
</ejb- jar>
```

No próximo código, apresentamos o EJB Session Bean Stateful - SalesBasketBean. Inicialmente seguem s interfaces Home e Remote, SalesBasketHome e SalesBasket respectivamente, a implementação do EJB SalesBasketBean e por fim o deployment descriptor deste EJB.

### Interface Home : SalesBasketHome.

```
package com.book.project.ejb.session;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

// Inteface Home, obtém uma referência para a interface Remote, esta
// ultima que fornece os serviços para o carrinho de compras.
public interface SalesBasketHome extends javax.ejb.EJBHome {
    public SalesBasket create() throws CreateException, RemoteException;
}
```

### Interface Remote : SalesBasket.

```
package com.book.project.ejb.session;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

// Interface Remota da cesta de produtos, apresenta serviços que serão utilizados
// na venda dos produtos ao usuário.
// Estes serviços são: iniciar e finalizar venda, adicionar e remover produto
// da cesta de produtos, limpar a cesta, calcular o valor total dos produtos etc.
```

```
public interface SalesBasket extends javax.ejb.EJBObject {
    public void initSales(com.book.project.vo.UserVO user)
            throws RemoteException;
    public java.lang.Boolean finalizeSale() throws RemoteException;
    public void addProduct(com.book.project.vo.ProductVO product)
throws
            RemoteException;
    public java.lang.Boolean removeProduct
(com.book.project.vo.ProductVO
            product) throws RemoteException;
    public java.lang.Integer calcBasketPrice() throws RemoteException;
    public void freeBasket() throws RemoteException;
    public Boolean authenticateUser(com.book.project.vo.UserVO
user)
            throws RemoteException;
}
```

## Session Bean Stateful : SalesBasketBean.

```
package com.book.project.ejb.session;

import javax.ejb.*;
import java.util.*;
import com.book.project.vo.*;
import com.book.project.ejb.*;
import com.book.project.ejb.entity.*;

// Apresenta o serviço de carrinho, que mantém os produto em
venda
// de um determinado usuário.
// Estes produtos são mantidos em uma Map, e podem ser
atualizados,
// removidos ou inseridos no carrinho de compras.
// O usuário também é registrado como proprietário desta compra.
public class SalesBasketBean implements SessionBean {

    // Contexto do Session Bean.
    private SessionContext sessionContext;

    // Cesta de produtos.
    private Map basket;

    // Usuário dono desta compra.
```

```
    private UserVO user;

    // Criação da instância deste objeto pelo container.
    // @throws CreateException exceção na criação da instância do
EJB.
    public void ejbCreate() throws CreateException {
        this.basket = new HashMap();
    }

    // Remoção da instância do EJB, pelo container.
    public void ejbRemove() {
    }

    // Utilizado pelo container para ativar o objeto que está passivo.
    public void ejbActivate() {
    }

    // Utilizado pelo container para tornar passivo um objeto ativo.
    public void ejbPassivate() {
    }

    // Configura o contexto do EJB Session Bean.
    // @param sessionContext contexto do EJB Session Bean.
    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }

    // Inicia a compra dos produtos para o usuário informado.
    // @param user usuário que iniciará a compra dos produtos.
    // @return verdadeiro caso a compra seja iniciada com sucesso.
    public void initSales(UserVO user) {
        this.user = user;
    }

    // Finaliza a compra.
    // Esta operação persiste os dados da venda, através do uso de
EJB Entity Bean
    // com acesso local e notifica o usuário do sucesso da sua
compra.
```

```
    // @return verdadeiro caso a compra tenha sido realizada com
sucesso.
    // @throws Exception exceção na criação do Entity Bean de
Produtos.
    public java.lang.Boolean finalizeSale() throws Exception {
        ProductLocalHome localHome = (ProductLocalHome)
                SalesServiceLocator.getInstance().getLocalHome
                (SalesServiceLocator.PRODUCT_BEAN);
        try {
            for (Iterator i = basket.entrySet().iterator(); i.hasNext();) {
                Map.Entry product = (Map.Entry) i.next();
                ProductVO vo = (ProductVO) product.getValue();
                localHome.create(vo.getName(), vo.getId(),
vo.getDescription(),
                    new Integer(vo.getPrice()));
            }
            return new Boolean(true);
        }catch (CreateException ex) {
            throw new Exception("Error while trying to finalize Sale. " +
ex);
        }
    }

    // Adiciona um produto no carrinho de compras.
    // @param product produto a ser inserido no carrinho de
compras.
    public void addProduct(com.book.project.vo.ProductVO product) {
        this.basket.put(product, product);
    }

    // Remove um produto do carrinho de compras.
    // @param product produto a ser removido do carrinho de
compras.
    // @return verdadeiro caso a remoção seja efetivada, falso caso
contrário.
    public java.lang.Boolean removeProduct
(com.book.project.vo.ProductVO
            product) {
        if (this.basket.remove(product) != null)
            return new Boolean(true);
        return new Boolean(false);
    }
```

```
    // Calcula o valor dos produtos contidos na cesta e retorna este
valor.
    // @return valor total do produtos contidos na cesta.
    public Integer calcBasketPrice() {
        int value = 0;
        for (Iterator i = basket.entrySet().iterator(); i.hasNext(); ) {
            Map.Entry entry = (Map.Entry) i.next();
            value =+ ((ProductVO) entry.getValue()).getPrice();
        }
        return new Integer(value);
    }

    // Remove os produtos da cesta de compras.
    public void freeBasket() {
        this.basket.clear();
    }

    // Valida e autentica o usuário, verificado se o mesmo é um
usuário válido.
    // Esta operação deveria ser mais completa, mas não é o intuito
deste exemplo
    // validar esta operação, sendo assim, esta operação será bastante
simples.
    // @param user usuário a ser autenticado.
    // @return verdadeiro caso seja validado, falso caso contrário.
    public Boolean authenticateUser(UserVO user) {
        return new Boolean(true);
    }
}
```

## Deployment Descriptor : SalesBasketBean.

```
<?xml version=”1.0" encoding=”UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC “-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN” “http://java.sun.com/dtd/ejb-
jar_2_0.dtd”>
<ejb-jar>
    <enterprise-beans>
        <session>
            <display-name>SalesBasket</display-name>
            <ejb-name>SalesBasket</ejb-name>
```

```
<home>com.book.project.ejb.session.SalesBasketHome</home>

<remote>com.book.project.ejb.session.SalesBasket</remote>
        <ejb- class>com.book.project.ejb.session.SalesBasketBean</ejb-
class>
        <session- type>Stateful</session- type>
        <transaction- type>Container</transaction- type>
      </session>
   </enterprise- beans>
   <assembly- descriptor>
      <container- transaction>
         <method>
            <ejb- name>SalesBasket</ejb- name>
            <method- name>*</method- name>
         </method>
         <trans- attribute>Required</trans- attribute>
      </container- transaction>
   </assembly- descriptor>
</ejb- jar>
```

Vemos a seguir os códigos do EJB Entity Bean BMP - UserBean. Para este, além das interfaces Home e Remote para acesso remoto, dispomos das interfaces para acesso local. Então seguem as interfaces UserHome, User, UserLocalHome e UserLocal. Depois a implementação do EJBUserBean e por fim o seu deployment descriptor.

### Interface Home : UserHome.

```
package com.book.project.ejb.entity;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

// Interface <code>Home</code> utilizada no acesso remoto.
// Provê o método para criação da entidade e localização da entidade
// dada a sua chave primária.
public interface UserHome extends javax.ejb.EJBHome {
```

```
    public User create(String name, Integer cpf, String address, String
email)
            throws CreateException, RemoteException;
    public User findByPrimaryKey(Integer cpf) throws FinderException,

            RemoteException;
}
```

### Interface Remote : User.

```
package com.book.project.ejb.entity;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

// Interface <code>Remote</code> do bean de entidade usuário
utilizado para
// acesso remoto. Provê acesso aos dados deste entidade.
public interface User extends javax.ejb.EJBObject {
    public String getName() throws RemoteException;
    public Integer getCpf() throws RemoteException;
    public String getAddress() throws RemoteException;
    public String getEmail() throws RemoteException;
    public void setName(String name) throws RemoteException;
    public void setAddress(String address) throws RemoteException;
    public void setEmail(String email) throws RemoteException;
}
```

### Interface LocalHome : UserLocalHome.

```
package com.book.project.ejb.entity;

import javax.ejb.*;
import java.util.*;

// Interface <code>Home</code> utilizada para acesso local.
// Provê os métodos de criação da entidade usuário e localização
// do objeto através de sua chave primária.
public interface UserLocalHome extends javax.ejb.EJBLocalHome {
    public UserLocal create(String name, Integer cpf, String address,
String email)
            throws CreateException;
```

```
    public UserLocal findByPrimaryKey(Integer cpf) throws
FinderException;
}
```

## Interface Local : UserLocal.

```
package com.book.project.ejb.entity;

import javax.ejb.*;
import java.util.*;

// Interface <code>Remote</code> utilizada para acesso local e
que provê os
// métodos para configurar e obter os valores dos atributos do
usuário.
public interface UserLocal extends javax.ejb.EJBLocalObject {
    public String getName();
    public Integer getCpf();
    public String getAddress();
    public String getEmail();
    public void setName(String name);
    public void setAddress(String address);
    public void setEmail(String email);
}
```

## Entity Bean BMP : UserBean.

```
package com.book.project.ejb.entity;

import java.rmi.*;
import java.sql.*;
import javax.ejb.*;
import javax.naming.*;

// EJBEntity Bean BMP,responsável por mantêr os dados de usuário em
um meio persistente.
// <p>
// Apresenta os métodos de configuração e obtenção dos valores dos
atributos
// da entidade, criação, remoção, atualização e localização da
entidade
// no meio de persistência.
public class UserBean implements EntityBean {
```

```
    // Contexto do EJB Entity Bean.
    private EntityContext entityContext;

    // Conexão com o BD.
    private transient Connection connection = null;

    // Query de inserção do objeto no BD.
    private final String INSERT_QUERY =
          "INSERT INTO USER(Name, Cpf, Address, Email) VALUES
(?, ?, ?, ?)";

    // Query de atualização do objeto no BD.
    private final String UPDATE_QUERY =
        "UPDATE USER SET Name = ?, Address = ?, Email = ? WHERE
Cpf = ?";

    // Query de remoção do objeto no BD.
    private final String DELETE_QUERY = "DELETE FROM USER WHERE
Cpf = ?";

    // Query de obtenção do objeto do BD.
    private final String SELECT_QUERY = "SELECT Name, Cpf,
Address,
        Email FROM USER WHERE Cpf = ?";

    // Query utilizada para obter os objetos dado sua chave primária.
    private final String FIND_BY_PK = "SELECT Name, Cpf, Address,
        Email FROM USER WHERE Cpf = ?";

    // Nome do usuário.
    private String name;

    // CPF do usuário.
    private Integer cpf;

    // Endereço do usuário.
    private String address;

    // Email do usuário.
    private String email;
```

```
    // Utilizado pelo container para criar o objeto usuário.
    // @param name nome do usuário.
    // @param cpf cpf do usuário.
    // @param address endereço do usuário.
    // @param email email do usuário.
    // @return chave primária da entidade usuário.
    // @throws CreateException exceção na criação desta instância.
    public Integer ejbCreate(String name, Integer cpf, String address,
String email)
            throws CreateException {
        try {
            connection = getConnection();
            try {
                PreparedStatement stmt = connection.prepareStatement
(INSERT_QUERY);
                try {
                    stmt.setString(1, name);
                    stmt.setInt(2, cpf.intValue());
                    stmt.setString(3, address);
                    stmt.setString(4, email);
                    stmt.execute();
                }
                finally {stmt.close();}
            }
            finally {connection.close();}
        }catch (Exception e){
            throw new CreateException();
        }
        setName(name);
        setCpf(cpf);
        setAddress(address);
        setEmail(email);
        return getCpf();
    }

    // Executado após a criação da instância.
    // Pode ser utilizado para realizar alguma operação neste
momento.
```

```
    // Utilizado para configurar os relacionamentos para o EJB Entity
Bean CMP.
    // @param name nome do usuário.
    // @param cpf cpf do usuário.
    // @param address endereço do usuário.
    // @param email email do usuário.
    // @throws CreateException exceção na criação desta instância.
    public void ejbPostCreate(String name, Integer cpf, String address,
String email)
            throws CreateException {
    }

    // Remoção da instância do EJB, pelo container.
    // @throws RemoveException exceção na remoção do objeto.
    public void ejbRemove() throws RemoveException {
        try {
            connection = getConnection();
            try {
                PreparedStatement stmt = connection.prepareStatement
(DELETE_QUERY);
                try {
                    stmt.setInt(1, this.cpf.intValue());
                    stmt.execute();
                }
                finally {stmt.close();}
            }
            finally {connection.close();}
        } catch (Exception e){
            throw new RemoveException("Error removing element.");
        }
    }

    // Configura o nome do usuário.
    // @param name nome do usuário.
    public void setName(String name) {
        this.name = name;
    }

    // Configura o cpf do usuário.
    // @param cpf cpf do usuário.
```

```
public void setCpf(Integer cpf) {
    this.cpf = cpf;
}

// Configura o endereço do usuário.
// @param address endereço do usuário.
public void setAddress(String address) {
    this.address = address;
}

// Configura o email do usuário.
// @param email email do usuário.
public void setEmail(String email) {
    this.email = email;
}

// Obtém o nome do usuário.
// @return nome do usuário.
public String getName() {
    return name;
}

// Obtém o cpf do usuário.
// @return cpf do usuário.
public Integer getCpf() {
    return cpf;
}

// Obtém o endereço do usuário.
// @return endereço do usuário.
public String getAddress() {
    return address;
}

// Obtém o email do usuário.
// @return email do usuário.
public String getEmail() {
    return email;
}
```

```
    // Realiza a localização do objeto pela sua chave primária.
    // @param cpf cpf do usuário.
    // @return chave primária do objeto em memória.
    // @throws FinderException exceção em caso de erro ao localizar
objeto.
    public Integer ejbFindByPrimaryKey(Integer cpf) throws
FinderException {
        try {
            connection = getConnection();
            try {
                PreparedStatement stmt = connection.prepareStatement
(FIND_BY_PK);
                try {
                    stmt.setInt(1, cpf.intValue());
                    ResultSet rs = stmt.executeQuery();
                    try {
                        if (rs.next()) {
                            this.setName(rs.getString(1));
                            this.setAddress(rs.getString(2));
                            this.setEmail(rs.getString(3));
                            this.setCpf(new Integer(rs.getInt(4)));
                        }
                    }
                    finally {rs.close();}
                }
                finally {stmt.close();}
            }
            finally { connection.close(); }
        } catch (Exception e){
            throw new FinderException("Error finding object by id: " +
cpf);
        }
        return this.cpf;
    }


    // Utilizado pelo container para carregar os dados do objeto do
meio
    // de persistência para a memória.
    public void ejbLoad() {
```

```
        try {
            connection = getConnection();
            try {
                PreparedStatement stmt = connection.prepareStatement
(SELECT_QUERY);
                try {
                    stmt.setInt(1, this.cpf.intValue());
                    ResultSet rs = stmt.executeQuery();
                    try {
                        if (rs.next()) {
                            this.setName(rs.getString(1));
                            this.setAddress(rs.getString(2));
                            this.setEmail(rs.getString(3));
                        }
                    }
                    finally {rs.close();}
                }
                finally {stmt.close();}
            }
            finally {connection.close();}
        }catch (Exception e){
            throw new EJBException("Error loading objects.");
        }
    }

    // Utilizado pelo container para persistir os dados constantes em
memória
    // para um meio de persistência.
    public void ejbStore() {
        try {
            connection = getConnection();
            try {
                PreparedStatement stmt = connection.prepareStatement
(UPDATE_QUERY);
                try {
                    stmt.setString(1, this.name);
                    stmt.setString(2, this.address);
                    stmt.setString(3, this.email);
                    stmt.setInt(4, this.cpf.intValue());
                    stmt.execute();
```

```
            }
            finally { stmt.close(); }
        }
        finally { connection.close(); }
    } catch (Exception e){
        throw new EJBException("Error persisting objects.");
    }
}

// Utilizado pelo container para ativar o objeto que está passivo.
public void ejbActivate() {
}

// Utilizado pelo container para tornar passivo um objeto ativo.
public void ejbPassivate() {
}

// Desconfigura o contexto do EJB Entity Bean.
public void unsetEntityContext() {
    this.entityContext = null;
}

// Configura o contexto do EJB Entity Bean.
// @param entityContext contexto do EJB Entity Bean.
public void setEntityContext(EntityContext entityContext) {
    this.entityContext = entityContext;
}

// Cria uma conexão com o banco de dados padrão.
// @return conexão do banco de dados.
private Connection getConnection() {
    Connection conn = null;
    try {
        InitialContext ctx = new InitialContext();
        conn = (Connection) ctx.lookup("java:/DefaultDS");
    } catch (Exception e){
        throw new RuntimeException(e.getMessage());
    }
    return conn;
```

```
    }
}
```

**Deployment Descriptor : UserBean.**

```
<?xml version=”1.0" encoding=”UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC “-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN” “http://java.sun.com/dtd/ejb-
jar_2_0.dtd”>
<ejb-jar>
   <enterprise-beans>
      <entity>
         <display-name>User</display-name>
         <ejb-name>User</ejb-name>
         <home>com.book.project.ejb.entity.UserHome</home>
         <remote>com.book.project.ejb.entity.User</remote>
         <local-
home>com.book.project.ejb.entity.UserLocalHome</local-home>
         <local>com.book.project.ejb.entity.UserLocal</local>
         <ejb-class>com.book.project.ejb.entity.UserBean</ejb-
class>
         <persistence-type>Bean</persistence-type>
         <prim-key-class>java.lang.String</prim-key-class>
         <reentrant>False</reentrant>
         <abstract-schema-name>User</abstract-schema-name>
      </entity>
   </enterprise-beans>
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>User</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Required</trans-attribute>
      </container-transaction>
   </assembly-descriptor>
</ejb-jar>
```

O próximo código apresenta o EJB Entity Bean CMP - ProductBean. Este EJB apresenta também as interfaces Home, Remote, Local e LocalHome que são respectivamente, ProductHome, Product, ProductLocal e

ProductLocalHome. Neste Entity Bean usamos uma classe de chave-primária específica, isto é, definimos uma nova classe para este fim que é a ProductPK e segue na seqüência. Em seguida temos a implementação do EJB ProductBean. Observe que como definimos este EJB como Entity Bean CMP, a implementação do EJB ProductBean não apresenta codificação para os métodos de persistência do objeto. Como sabemos, isto é feito pelo próprio container do servidor de aplicações. Bem, por fim apresentamos o deployment descriptor deste EJB.

**Interface Home : ProductHome.**

```
package com.book.project.ejb.entity;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

// Interface <code>Home</code> utilizada no acesso remoto.
// Provê o método para criação da entidade e localização da entidade
// dada a sua chave-primária.
public interface ProductHome extends javax.ejb.EJBHome {
    public Product create(String name, String description, Integer
price,
            ProductPK productPK) throws CreateException,
RemoteException;
    public Product findByPrimaryKey(ProductPK pk)
            throws FinderException, RemoteException;
}
```

**Interface Remote : Product**.

```
package com.book.project.ejb.entity;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

// Interface <code>Remote</code> do bean de entidade produto
utilizado para
// acesso remoto. Provê acesso aos dados desta entidade.
```

```
public interface Product extends javax.ejb.EJBObject {
    public String getName() throws RemoteException;
    public String getDescription() throws RemoteException;
    public Integer getPrice() throws RemoteException;
    public ProductPK getProductPK() throws RemoteException;
    public void setName(String name) throws RemoteException;
    public void setDescription(String description) throws
RemoteException;
    public void setPrice(Integer price) throws RemoteException;
    public void setProductPK(ProductPK productPK) throws
RemoteException;
}
```

## Interface LocalHome : ProductLocalHome.

```
package com.book.project.ejb.entity;

import javax.ejb.*;
import java.util.*;

// Interface <code>Home</code> utilizada para acesso local.
// Provê os métodos de criação da entidade produto e localização
// do objeto através de sua chave- primária.
public interface ProductLocalHome extends javax.ejb.EJBLocalHome {
    public ProductLocal create(String name, String description,
            Integer price, ProductPK productPK) throws
CreateException;
    public ProductLocal findByPrimaryKey(ProductPK pk) throws
FinderException;
}
```

## Interface Local : ProductLocal.

```
package com.book.project.ejb.entity;

import javax.ejb.*;
import java.util.*;

// Interface <code>Remote</code> utilizada para acesso local e
que provê os
// métodos para configurar e obter os valores dos atributos do
produto.
public interface ProductLocal extends javax.ejb.EJBLocalObject {
```

```
    public String getName();
    public String getDescription();
    public Integer getPrice();
    public ProductPK getProductPK();
    public void setName(String name);
    public void setDescription(String description);
    public void setPrice(Integer price);
    public void setProductPK(ProductPK productPK);
}
```

**PrimaryKey : ProductPK.**

```
package com.book.project.ejb.entity;

import java.io.*;

// Classe que define uma chave primária para a entidade produto.
public class ProductPK implements Serializable {
    // Identificador único do objeto produto.
    private String id;

    // Construtor padrão.
    public ProductPK() {
    }

    // Construtor customizado.
    // @param id identificador do produto.
    public ProductPK(String id){
        this.id = id;
    }

    // Obtém a chave-primária Id da entidade produto.
    // @return chave-primária Id da entidade produto.
    public String getId() {
        return this.id;
    }

    // Determina se o objeto atual é igual ao objeto a ser comparado.
    // @param obj objeto a ser comparado.
    // @return true caso o objeto seja igual, false caso contrário.
```

```
    public boolean equals(Object obj) {
        if (obj != null) {
            if (this.getClass().equals(obj.getClass())) {
                ProductPK that = (ProductPK) obj;
                return (((this.id == null) && (that.id == null)) || (this.id !
= null
                && this.id.equals(that.id)));
            }
        }
        return false;
    }

    // Define o método hashCode().
    // @return chave de hash.
    public int hashCode() {
        return id.hashCode();
    }
}
```

## Entity Bean CMP : ProductBean.

```
package com.book.project.ejb.entity;

import javax.ejb.*;

// Entity Bean CMP, responsável por mantêr os dados dos produtos
em um meio persistente.
// <p>
// Os métodos de negócio deste EJBsão implementados pelo próprio
container e os métodos
// finders e select devem ser definidos por meio de EQL no
deployment descriptor.
// Observe que o método findByPrimaryKey é definido como padrão.
// <p>
// Podemos definir em um Entity Bean, tanto CMP quanto BMP,
// uma classe que define a chave-primária do EJB.
// Faremos isto para este EJB para exemplificarmos o seu uso.
// A classe de chave-primária do produto será ProductPK.
abstract public class ProductBean implements EntityBean {
    // Contexto do EJB Entity Bean.
    private EntityContext entityContext;
```

```
    // Utilizado pelo container para criar o objeto produto.
    // @param name nome do produto.
    // @param id identificador único do produto.
    // @param description descrição do produto.
    // @param price preço do produto.
    // @return chave primária do produto.
    // @throws CreateException exceção na criação desta instância.
    public ProductPK ejbCreate(java.lang.String name, java.lang.String
description,
            java.lang.Integer price,
com.book.project.ejb.entity.ProductPK
            productPK) throws CreateException {
        setName(name);
        setDescription(description);
        setPrice(price);
        setProductPK(productPK);
        return getProductPK();
    }

    // Executado após a criação da instância.
    // Pode ser utilizado para realizar alguma operação neste
momento.
    // Utilizado para configurar os relacionamentos para o EJB Entity
Bean CMP.
    // @param name nome do produto.
    // @param id identificador único do produto.
    // @param description descrição do produto.
    // @param price preço do produto.
    // @throws CreateException exceção na criação desta instância.
    public void ejbPostCreate(java.lang.String name, java.lang.String
description,
            java.lang.Integer price,
com.book.project.ejb.entity.ProductPK
            productPK) throws CreateException {
    }

    // Remoção da instância do EJB, pelo container.
    // @throws RemoveException exceção na remoção do objeto.
    public void ejbRemove() throws RemoveException {
    }
```

```
    // Utilizado pelo container para carregar os dados do objeto do
meio
    // de persistência para a memória.
    public void ejbLoad() {
    }

    // Utilizado pelo container para persistir os dados constantes em
memória
    // para um meio de persistência.
    public void ejbStore() {
    }

    // Utilizado pelo container para ativar o objeto que está passivo.
    public void ejbActivate() {
    }

    // Utilizado pelo container para tornar passivo um objeto ativo.
    public void ejbPassivate() {
    }

    // Desconfigura o contexto do EJB Entity Bean.
    public void unsetEntityContext() {
        this.entityContext = null;
    }

    // Configura o contexto do EJB Entity Bean.
    // @param entityContext contexto do EJB Entity Bean.
    public void setEntityContext(EntityContext entityContext) {
        this.entityContext = entityContext;
    }

    // Configura o nome do produto.
    // Implementado pelo container para o Entity Bean CMP.
    // @param name nome do produto.
    public abstract void setName(java.lang.String name);

    // Configura a descrição do produto.
    // Implementado pelo container para o Entity Bean CMP.
    // @param description descrição do produto.
```

```
    public abstract void setDescription(java.lang.String description);

    // Configura o preço do produto.
    // Implementado pelo container para o Entity Bean CMP.
    // @param price preço do produto.
    public abstract void setPrice(java.lang.Integer price);

    // Configura a chave primária do produto.
    // @param productPK chave primária do produto.
    public abstract void setProductPK
(com.book.project.ejb.entity.ProductPK
            productPK);

    // Obtém o nome do produto.
    // Implementado pelo container para o Entity Bean CMP.
    // @return nome do produto.
    public abstract java.lang.String getName();

    // Obtém a descrição do produto.
    // Implementado pelo container para o Entity Bean CMP.
    // @return descrição do produto.
    public abstract java.lang.String getDescription();

    // Obtém o preço do produto.
    // Implementado pelo container para o Entity Bean CMP.
    // @return preço do produto.
    public abstract java.lang.Integer getPrice();

    // Obtém a chave primária do produto.
    // @return chave primária do produto.
    public abstract com.book.project.ejb.entity.ProductPK
getProductPK();

}
```

## Deployment Descriptor : ProductBean.

```
<?xml version=”1.0" encoding=”UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC “-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN” “http://java.sun.com/dtd/ejb-
jar_2_0.dtd”>
```

```
<ejb- jar>
   <enterprise- beans>
      <entity>
         <display- name>Product</display- name>
         <ejb- name>Product</ejb- name>

<home>com.book.project.ejb.entity.ProductHome</home>
         <remote>com.book.project.ejb.entity.Product</remote>
         <local-
home>com.book.project.ejb.entity.ProductLocalHome</local- home>
         <local>com.book.project.ejb.entity.ProductLocal</local>
         <ejb- class>com.book.project.ejb.entity.ProductBean</ejb-
class>
         <persistence- type>Container</persistence- type>
         <prim- key-
class>com.book.project.ejb.entity.ProductPK</prim- key- class >
         <reentrant>False</reentrant>
         <cmp- version>2.x</cmp- version>
         <abstract- schema- name>Product</abstract- schema-
name>
         <cmp- field>
            <field- name>name</field- name>
         </cmp- field>
         <cmp- field>
            <field- name>description</field- name>
         </cmp- field>
         <cmp- field>
            <field- name>price</field- name>
         </cmp- field>
         <cmp- field>
            <field- name>productPK</field- name>
         </cmp- field>
         <primkey- field>productPK</primkey- field>
      </entity>
   </enterprise- beans>
   <assembly- descriptor>
      <container- transaction>
         <method>
            <ejb- name>Product</ejb- name>
            <method- name>*</method- name>
```

```
            </method>
            <trans- attribute>Required</trans- attribute>
        </container- transaction>
    </assembly- descriptor>
</ejb- jar>
```

Apresentamos a seguir, o código do EJB Message- Driven Bean - UserNotifierBean. A implementação do EJB segue logo a seguir. Após segue a implementação da mensagem utilizada para notificar o EJB MDB UserMessage e por fim o deployment descriptor deste EJB.

### Message- Driven Bean : UserNotifierBean.

```
package com.book.project.ejb.mdb;

import java.util.*;
import javax.ejb.*;
import javax.jms.*;
import javax.mail.*;
import javax.naming.*;
import javax.mail.internet.*;
import com.book.project.vo.*;

// Responsável por notificar o usuário das novas atualizações do
estado da venda de
// seu produto, tais como produto localizado, produto enviado,
venda finalizada etc.
// O método onMessage() é responsável por receber um mensagem
do tipo TextMessage
// e enviar um email para o usuário com esta mensagem.
public class UserNotifierBean implements MessageDrivenBean,
MessageListener {
    // Contexto do Message- Drive Bean.
    private MessageDrivenContext messageDrivenContext;

    // Instância o objeto no servidor pelo container.
    public void ejbCreate() {
}

// Remove a instância do objeto no servidor.
public void ejbRemove() {
```

```
}

// Valida a mensagem recebida e envia uma notificação ao usuário
com
// a mensagem texto recebida como parâmentro.
// @param msg mensagem a ser notificada.
public void onMessage(javax.jms.Message msg) {
    try {
        if (msg instanceof UserMessage) {
            String message = ((UserMessage) msg).getMessage();
            UserVO user = ((UserMessage) msg).getUser();
            this.sendEmail(user.getHost(), user.getPartialEmail(),
                "Sales Notification", message);
        }
    } catch (Exception ex) {
        System.err.println("Error sending email for user. " + ex);
    }
}

// Envia um email de notificação para o usuário.
// @param host host do usuário.
// @param to email do destinatário.
// @param subject título da notificação.
// @param text mensagem da notificação.
// @throws Exception exceção caso não consiga enviar o email.

    private void sendEmail(String host, String to, String subject, String
text)
        throws Exception {
        try {
            // Obtém as propriedades do sistema
            Properties props = System.getProperties();
            // Configura o servidor de email
            props.put("mail.smtp.host", host);
            // Obtém a sessão
            javax.mail.Session session = javax.mail.Session.getDefaultInstance(props, null);
            // Define a mensagem
            MimeMessage message = new MimeMessage(session);
            message.setFrom(new InternetAddress("Scheduler"));
```

```
            message.addRecipient
(javax.mail.Message.RecipientType.TO,
                new InternetAddress(to));
            message.setSubject(subject);
            message.setText(text);
            // Envia a mensagem
            Transport.send(message);
        }catch (Exception e){
            throw new Exception("Error sending mail!" + e);
        }
    }

    // Configura o contexto do EJB MDB.
    // @param messageDrivenContext contexto do MDB.
    public void setMessageDrivenContext(MessageDrivenContext
messageDrivenContext) {
        this.messageDrivenContext = messageDrivenContext;
    }
}
```

**Mensagem : UserMessage.**

```
package com.book.project.ejb.mdb;

import java.io.*;
import com.book.project.vo.*;

// Objeto que contém a mensagem a ser entregue para o usuário.
// O usuário também é definido neste objeto.
public class UserMessage implements Serializable {

    // Usuário o qual será enviado a mensagem.
    private UserVO user;

    // Mensagem a ser enviada para o usuário.
    private String message;

    // Construtor padrão.
    public UserMessage(){
    }
```

```
    // Construtor customizado.
    // @param user usuário o qual será enviado a mensagem.
    // @param message mensagem a ser enviada para o usuário.
    public UserMessage(UserVO user, String message){
        this.user = user;
        this.message = message;
    }

    // Obtém a mensagem a ser enviada para o usuário.
    // @return mensagem a ser enviada para o usuário.
    public String getMessage() {
        return message;
    }

    // Configura a mensagem a ser enviada para o usuário.
    // @param message mensagem a ser enviada para o usuário.
    public void setMessage(String message) {
        this.message = message;
    }

    // Obtém o usuário o qual será enviado a mensagem.
    // @return usuário o qual será enviado a mensagem.
    public UserVO getUser() {
        return user;
    }

    // Configura o usuário o qual será enviado a mensagem.
    // @param user usuário o qual será enviado a mensagem.
    public void setUser(UserVO user) {
        this.user = user;
    }
}
```

## Deployment Descriptor : UserNotifierBean.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-
jar_2_0.dtd">
<ejb-jar>
    <enterprise-beans>
```

```
        <message- driven>
            <display- name>UserNotifier</display- name>
            <ejb- name>UserNotifier</ejb- name>
            <ejb-
class>com.book.project.ejb.mdb.UserNotifierBean</ejb- class>
            <transaction- type>Container</transaction- type>
            <message- driven- destination>
                <destination- type>javax.jms.Topic</destination- type>
                <subscription- durability>NonDurable</subscription-
durability>
            </message- driven- destination>
        </message- driven>
    </enterprise- beans>
    <assembly- descriptor>
        <container- transaction>
            <method>
                <ejb- name>UserNotifier</ejb- name>
                <method- name>*</method- name>
            </method>
            <trans- attribute>Required</trans- attribute>
        </container- transaction>
    </assembly- descriptor>
</ejb- jar>
```

Foi criado um EJB Session Bean Stateless - ProjecSupportBean - com o intuito de preparar o ambiente da aplicação exemplo. O código do EJB, tais como suas interfaces, implementação e deployment descriptors não foram comentadas pois é utilizado somente como um serviço auxiliar para a aplicação. Segue os códigos do EJB.

**Interface Home : ProjectSupportHome .**

```
package com.book.project.ejb.session;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

// Inteface Home, obtém uma referência para a interface Remote,
esta
// ultima que fornece os serviços de suporte a aplicação exemplo.
```

```
public interface ProjectSupportHome extends javax.ejb.EJBHome {
    public ProjectSupport create() throws CreateException,
RemoteException;
}
```

**Interface Remote : ProjectSupport.**

```
package com.book.project.ejb.session;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

// Interface Remote do EJB <code>ProjectSupportBean</code>.
// Apresenta serviços de suporte para a aplicação exemplo.

public interface ProjectSupport extends javax.ejb.EJBObject {
    public void prepareEnvironment() throws RemoteException;
    public void clearEnvironment() throws RemoteException;
}
```

**Session Bean Stateless : ProjectSupportBean.**

```
package com.book.project.ejb.session;

import java.sql.*;
import java.rmi.*;
import javax.ejb.*;
import javax.naming.*;

// EJB Session Bean Stateless utilizado como serviço de suporte
// para a aplicação exemplo.
// <p>
// Provê serviços para preparação do ambiente.
public class ProjectSupportBean implements SessionBean {

    // Contexto do Session Bean.
    private SessionContext sessionContext;

    // Criação da instância deste objeto pelo container.
    // @throws CreateException exceção na criação da instância do
EJB.
```

```
    public void ejbCreate() throws CreateException {
    }

    // Remoção da instância do EJB, pelo container.
    public void ejbRemove() {
    }

    // Utilizado pelo container para ativar o objeto que está passivo.
    public void ejbActivate() {
    }

    // Utilizado pelo container para tornar passivo um objeto ativo.
    public void ejbPassivate() {
    }

    // Configura o contexto do EJB Session Bean.
    // @param sessionContext contexto do EJB Session Bean.
    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }

    // Prepara o amibente para a execução da aplicação exemplo.
    public void prepareEnvironment() {
        System.out.println("Preparing environment.");
        try {
            InitialContext ctx = new InitialContext();
                javax.sql.DataSource ds = (javax.sql.DataSource)
                    ctx.lookup("java:/DefaultDS");
            Connection conn = ds.getConnection();
            try {
                PreparedStatement stmt = conn.prepareStatement
("CREATE TABLE USER
                        (NAME VARCHAR(40), CPF INTEGER(30), ADDRESS
VARCHAR(50),
                        EMAIL VARCHAR(30), CONSTRAINT PK_USER
PRIMARY KEY (CPF) )");
                try {
                    stmt.execute();
                }
                finally {
                    stmt.close();
```

```
            }
        }
        finally {
        conn.close();
        }
    }catch (Exception ex) {
        throw new EJBException("Error preparing environment. " +
ex.getMessage());
    }
    System.out.println("Do it.");
  }

  // Limpa o ambiente de execução da aplicação exemplo.
  public void clearEnvironment() {
    System.out.println("Clearing environment.");
    try {
      InitialContext ctx = new InitialContext();
      javax.sql.DataSource ds = (javax.sql.DataSource)
            ctx.lookup("java:/DefaultDS");
      Connection conn = ds.getConnection();
      try {
        PreparedStatement stmt = conn.prepareStatement
("DROP TABLE USER");
        try {
          stmt.execute();
        }
        finally {
          stmt.close();
        }
      }
      finally {
        conn.close();
      }
    }catch (Exception ex) {
        throw new EJBException("Error clearing environment. " +
ex.getMessage());
    }
    System.out.println("Do it.");
  }
}
```

## Deployment Descriptor : ProjectSupportBean.

```
<?xml version=”1.0" encoding=”UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC “-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN” “http://java.sun.com/dtd/ejb-
jar_2_0.dtd”>
<ejb-jar>
   <enterprise-beans>
      <session>
         <display-name>ProjectSupport</display-name>
         <ejb-name>ProjectSupport</ejb-name>

<home>com.book.project.ejb.session.ProjectSupportHome</home>

<remote>com.book.project.ejb.session.ProjectSupport</remote>
         <ejb-
class>com.book.project.ejb.session.ProjectSupportBean</ejb-class>
         <session-type>Stateless</session-type>
         <transaction-type>Container</transaction-type>
      </session>
   </enterprise-beans>
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>ProjectSupport</ejb-name>
            <method-name>*</method-name>
         </method>
            <trans-attribute>Required</trans-attribute>
      </container-transaction>
   </assembly-descriptor>
</ejb-jar>
```

## Deployment Descriptor da Aplicação Exemplo : ejb-jar.xml

```
<?xml version=”1.0" encoding=”UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC “-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN” “http://java.sun.com/dtd/ejb-
jar_2_0.dtd”>
<ejb-jar>
   <enterprise-beans>
      <session>
         <display-name>SalesSupport</display-name>
```

```
            <ejb- name>SalesSupport</ejb- name>

<home>com.book.project.ejb.session.SalesSupportHome</home>

<remote>com.book.project.ejb.session.SalesSupport</remote>
            <ejb-
class>com.book.project.ejb.session.SalesSupportBean</ejb- class>
            <session- type>Stateless</session- type>
            <transaction- type>Container</transaction- type>
        </session>
        <session>
            <display- name>SalesBasket</display- name>
            <ejb- name>SalesBasket</ejb- name>

<home>com.book.project.ejb.session.SalesBasketHome</home>

<remote>com.book.project.ejb.session.SalesBasket</remote>
            <ejb- class>com.book.project.ejb.session.SalesBasketBean</ejb-
class>
            <session- type>Stateful</session- type>
            <transaction- type>Container</transaction- type>
        </session>
        <session>
            <display- name>ProjectSupport</display- name>
            <ejb- name>ProjectSupport</ejb- name>

<home>com.book.project.ejb.session.ProjectSupportHome</home>

<remote>com.book.project.ejb.session.ProjectSupport</remote>
            <ejb-
class>com.book.project.ejb.session.ProjectSupportBean</ejb- class>
            <session- type>Stateless</session- type>
            <transaction- type>Container</transaction- type>
        </session>
        <entity>
            <display- name>User</display- name>
            <ejb- name>User</ejb- name>
            <home>com.book.project.ejb.entity.UserHome</home>
            <remote>com.book.project.ejb.entity.User</remote>
            <local-
home>com.book.project.ejb.entity.UserLocalHome</local- home>
```

```
            <local>com.book.project.ejb.entity.UserLocal</local>
            <ejb- class>com.book.project.ejb.entity.UserBean</ejb-
class>
            <persistence- type>Bean</persistence- type>
            <prim- key- class>java.lang.Integer</prim- key- class>
            <reentrant>False</reentrant>
            <abstract- schema- name>User</abstract- schema- name>
        </entity>
        <entity>
            <display- name>Product</display- name>
            <ejb- name>Product</ejb- name>

<home>com.book.project.ejb.entity.ProductHome</home>
            <remote>com.book.project.ejb.entity.Product</remote>
            <local-
home>com.book.project.ejb.entity.ProductLocalHome</local- home>
            <local>com.book.project.ejb.entity.ProductLocal</local>
            <ejb- class>com.book.project.ejb.entity.ProductBean</ejb-
class>
            <persistence- type>Container</persistence- type>
            <prim- key-
class>com.book.project.ejb.entity.ProductPK</prim- key- class>
            <reentrant>False</reentrant>
            <cmp- version>2.x</cmp- version>
            <abstract- schema- name>Product</abstract- schema-
name>
            <cmp- field>
                <field- name>name</field- name>
            </cmp- field>
            <cmp- field>
                <field- name>description</field- name>
            </cmp- field>
            <cmp- field>
                <field- name>price</field- name>
            </cmp- field>
            <cmp- field>
                <field- name>productPK</field- name>
            </cmp- field>
            <primkey- field>productPK</primkey- field>
        </entity>
        <message- driven>
```

```
            <display- name>UserNotifier</display- name>
            <ejb- name>UserNotifier</ejb- name>
            <ejb-
class>com.book.project.ejb.mdb.UserNotifierBean</ejb- class>
            <transaction- type>Container</transaction- type>
            <message- driven- destination>
               <destination- type>javax.jms.Topic</destination- type>
               <subscription- durability>NonDurable</subscription-
durability>
            </message- driven- destination>
         </message- driven>
      </enterprise- beans>
      <assembly- descriptor>
         <container- transaction>
            <method>
               <ejb- name>SalesSupport</ejb- name>
               <method- name>*</method- name>
            </method>
            <trans- attribute>Required</trans- attribute>
         </container- transaction>
         <container- transaction>
            <method>
               <ejb- name>SalesBasket</ejb- name>
               <method- name>*</method- name>
            </method>
            <trans- attribute>Required</trans- attribute>
         </container- transaction>
         <container- transaction>
            <method>
               <ejb- name>User</ejb- name>
               <method- name>*</method- name>
            </method>
            <trans- attribute>Required</trans- attribute>
         </container- transaction>
         <container- transaction>
            <method>
               <ejb- name>Product</ejb- name>
               <method- name>*</method- name>
            </method>
            <trans- attribute>Required</trans- attribute>
```

```
        </container- transaction>
        <container- transaction>
           <method>
              <ejb- name>UserNotifier</ejb- name>
              <method- name>*</method- name>
           </method>
         <trans- attribute>Required</trans- attribute>
        </container- transaction>
        <container- transaction>
           <method>
              <ejb- name>ProjectSupport</ejb- name>
              <method- name>*</method- name>
           </method>
           <trans- attribute>Required</trans- attribute>
        </container- transaction>
     </assembly- descriptor>
</ejb- jar>
```

## Deployment Descriptor específico para o JBoss : jboss.xml

```
<?xml version=”1.0" encoding=”UTF- 8"?>
<!DOCTYPE jboss PUBLIC ‘-//JBoss//DTD JBOSS 3.0//EN’
‘http://www.jboss.org/j2ee/dtd/jboss_3_0.dtd’>
<jboss>
   <enterprise- beans>
      <session>
         <ejb- name>SalesSupport</ejb- name>
         <jndi- name>SalesSupport</jndi- name>
      </session>
      <session>
         <ejb- name>SalesBasket</ejb- name>
         <jndi- name>SalesBasket</jndi- name>
      </session>
      <session>
         <ejb- name>ProjectSupport</ejb- name>
         <jndi- name>ProjectSupport</jndi- name>
      </session>
      <entity>
         <ejb- name>User</ejb- name>
         <jndi- name>UserRemote</jndi- name>
```

```
            <local-jndi-name>User</local-jndi-name>
        </entity>
        <entity>
            <ejb-name>Product</ejb-name>
            <jndi-name>ProductRemote</jndi-name>
            <local-jndi-name>Product</local-jndi-name>
        </entity>
        <message-driven>
            <ejb-name>UserNotifier</ejb-name>
            <destination-jndi-name>topic/testTopic</destination-
jndi-name>
        </message-driven>
    </enterprise-beans>
</jboss>
```

## Deployment Descriptor específico para o JBoss : jbosscmp-jdbc.xml

```
<?xml version=”1.0" encoding=”UTF-8"?>
<!DOCTYPE jbosscmp-jdbc PUBLIC ‘-//JBoss//DTD JBOSSCMP-JDBC
3.0//EN’ ‘http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_3_0.dtd’>
<jbosscmp-jdbc>
    <defaults>
        <datasource>java:/DefaultDS</datasource>
        <datasource-mapping>Hypersonic SQL</datasource-
mapping>
        <create-table>true</create-table>
        <remove-table>true</remove-table>
    </defaults>
    <enterprise-beans>
        <entity>
            <ejb-name>Product</ejb-name>
            <table-name>PRODUCT</table-name>
            <cmp-field>
                <field-name>name</field-name>
                <column-name>NAME</column-name>
            </cmp-field>
            <cmp-field>
                <field-name>description</field-name>
                <column-name>DESCRIPTION</column-name>
            </cmp-field>
```

```
            <cmp-field>
               <field-name>price</field-name>
               <column-name>PRICE</column-name>
            </cmp-field>
            <cmp-field>
               <field-name>productPK</field-name>
               <column-name>ID</column-name>
            </cmp-field>
         </entity>
      </enterprise-beans>
</jbosscmp-jdbc>
```