Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Informática e Estatística
Curso de Especialização em Ciência da Computação
Convênio UFSC - UNOESC – Campus Xanxerê

Aplicações Cliente-Servidor via Web Usando Java

Leandro J. Komosinski

Xanxerê, maio-junho de 2002.

Este material didático, formado pelo presente texto e pelos programas exemplo, foi concebido especificamente para o curso de especialização em Ciência da Computação no âmbito do convênio USC - UNOESC (Campus Xanxerê).

O código fonte dos exemplos apresentados no texto podem ser baixados acessando o endereço `http://www.inf.ufsc.br/~leandro/curso-webapps/`.

Se você deseja usar o material para outra finalidade por favor entre em contato como autor no seguinte endereço:

Prof. Leandro J. Komosinski

Departamento de Informática e Estatística
Centro Tecnológico
Universidade Federal de Santa Catarina
CEP 88040-900
Florianópolis / SC
--
e-mail   : leandro@inf.ufsc.br
home page : http://www.inf.ufsc.br/~leandro
fone   : 48 331 7508

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A finalidade desta apostila é dar suporte à disciplina **Aplicações Cliente-Servidor via Web Usando Java** para o curso de Especialização em Ciência da Computação oferecido pela Universidade Federal de Santa Catarina (UFSC) à Universidade do Oeste de Santa Catarina - Campus Xanxerê no ano de 2002.

O assunto tratado ao longo deste texto envolve, em diferentes graus, várias áreas da Computação tais como o paradigma de Programação Orientada a Objetos (POO), Sistemas Distribuídos, Banco de Dados, linguagens de programação (Java) e linguagens de marcação (HTML, JSP, XML, etc).

A compreensão, em profundidade, de todas estas áreas é algo que requer um tempo razoável de vivência prática. Assim, a presente disciplina deve ser encarada como um ponto de partida. Os únicos pré-requisitos são ter alguma experiência com POO e conhecer o conceito de linguagem de marcação.

Como mostrado ao longo do texto, o desenvolvimento de aplicações para web necessita de profissionais com diferentes perfis. Há o perfil do programador clássico e o perfil do projetista de interfaces (chamado de *web designer*). O primeiro, a rigor, não precisa conhecer nada a respeito das tarefas e habilidades do segundo e vice-versa. Por exemplo, um programador Java não precisa conhecer HTML e o *web designer*, que conhece tudo sobre HTML, não precisa saber programar.

É claro que na prática a divisão entre programador e projetista não é tão radical assim. Frequentemente ambos os profissionais conhecem um pouco (ou mesmo bastante) o serviço do outro.

1.1 Conhecendo o Terreno

Analisando-se o nome da disciplina podemos ter uma idéia inicial sobre o seu conteúdo e a ênfase adotada. Esta percepção ajudará na leitura dos demais capítulos onde os conceitos descritos a seguir serão redefinidos de forma mais completa.

O nome da disciplina é formado a partir de três conceitos independentes. São eles:

- Aplicações Cliente-Servidor.
- Aplicações para Web.
- Linguagem Java.

1.1.1 Aplicações Cliente-Servidor

Aplicações cliente-servidor são programas que seguem o **modelo cliente-servidor**. Neste modelo, um programa serve (envia) dados, **quanto solicitado**, para outro programa. O primeiro programa é chamado de **programa servidor** e o segundo de **programa cliente**.

Por exemplo, os dados enviados pelo programa servidor, ou simplesmente "servidor", podem ser informações a respeito de um usuário de uma operadora de cartão de crédito. O programa cliente, ou simplesmente "cliente", neste caso, está embutido nas máquinas leitoras de cartão e solicita ao servidor autorização para que seja realizado o pagamento.

Outro exemplo são aplicações cliente-servidor onde o servidor fornece ao cliente páginas HTML[1]. Este tipo de programa é também chamado de **servidor web**. O programa cliente, neste exemplo, pode ser qualquer programa que processe dados codificados em HTML.

Por exemplo, o Apache, desenvolvido pela Fundação Apache, é o servidor web mais utilizado na Internet. Como clientes, os mais populares são o Internet Explorer©, da Microsoft, o Netscape©, da AOL e o Opera©, da Opera.

1.1.2 Aplicações para Web

Aplicações para web são muito semelhantes às aplicações do tipo sevidor web. A diferença está na geração dinâmica dos conteúdos a serem enviados ao programa cliente. Em outras palavras, a solicitação do cliente **gera algum tipo de processamento no servidor que é específico para cada aplicação**.

1.1.3 Linguagem Java

Java é uma linguagem de programação surgida nos anos 1990 que implementa os conceitos do paradigma da orientação a objetos[2].

[1] Na verdade, como mostrado na seção 2.2, o que define uma aplicação web é o fato da comunicação entre cliente e servidor ser realizada por meio do protocolo HTTP. Outros formatos de arquivo podem ser enviados como, por exemplo XML

[2] Ao contrário do que algumas pessoas imaginam, Java e JavaScript são linguagens diferentes. Também não é correto afirmar que Java foi feita para a Internet.

Quando pensamos em Java como linguagem de programação estamos, na verdade, nos referindo ao que é chamado *Java 2 Platform, Standard Edition (J2SE).* Com o J2SE é possível compilar e executar programas. Estes programas podem ou não usar uma vasta biblioteca de classes que acompanha o J2SE. Além disso, muitas outras bibliotecas podem ser obtidas de graça na Internet uma vez que é bastante comum a adoção da filosofia de **software de código aberto**.

Além de J2SE, Java apresenta-se nas versões *Java 2 Platform, Micro Edition (J2ME)* e *Java 2 Platform, Enterprise Edition (J2EE).* As edições J2ME e J2EE definem, respectivamente, arquiteturas de software para serem executados em dispositivos de mão (celulares, palm tops, etc.) e arquiteturas de software para grandes aplicações corporativas.

A temática sobre aplicações para web está na alçada de J2EE. Aplicações web com fortes requisitos de segurança, transacionalidade e concorrência devem ser desenvolvidas utilizando-se a arquitetura J2EE completa, que pode envolver o uso de até 13 tecnologias[3]. Já para aplicações mais simples, a opção recomendada é desenvolvê-las utilizando-se apenas um subconjunto destas tecnologias.

A conceituação e implementação de aplicações web usando o padrão J2EE completo é bastante complexa[RAJ02]. Em função desta característica, ela não é recomendável quando a equipe de desenvolvimento é formada por pessoas com pouca ou nenhuma experiência em Java. No contexto desta disciplina, portanto, são apresentadas e discutidas apenas algumas das tecnologias do padrão J2EE.

As três tecnologias J2EE discutidas neste curso – JavaBeans (Capítulo 4), JSP (Capítulos 5 e 6) e JDBC (Capítulo 7) – são consideradas essenciais. Com elas é perfeitamente possível desenvolver aplicações para web bastante sofisticadas. A divisão por capítulos reflete essencialmente uma preocupação didática pois na prática as tecnologias são usadas simultaneamente.

Uma simples pesquisa na Internet revela a existência de milhares de páginas ou sites tratando das tecnologias de JavaBeans, JSP e JDBC. Um exemplo interessante é o site JDance[4].

As aplicações web discutidas e a serem implementadas nesta disciplina serão simples, plenamente funcionais e estarão baseadas no subconjunto de tecnologias J2EE discutido ao longo do curso.

Java não é, obviamente, o único caminho para o desenvolvimento de aplicações para web. Outras iniciativas, como ".Net"da Microsoft ou CORBA da OMG têm igual propósito [Rom99]. A opção pelo caminho Java para esta disciplina segue, certamente, alguns critérios pessoais (experiência do professor, percepção sobre o impacto das tecnologias Java no mercado de

[3]Rigorosamente falando, uma aplicação caracterizada como J2EE utiliza, no mínimo, a tecnologia de *Enterprise JavaBeans (EJB).*

[4]Disponível em `http://www.jdance.com/jspjavabeans.shtm`.

trabalho e existência de excelentes tecnologias de apoio baseadas em software de código aberto).

A existência da palavra "Java" no nome da disciplina caracteriza a ênfase em questões mais práticas do que teóricas. A teorização apresentada nesta disciplina visa apenas dar suporte à prática[5].

1.2 Dinâmica e Avaliação da Disciplina

A disciplina está baseada na **discussão** e **experimentação prática** de uma série de exemplos – alguns bastante simples e outros mais complexos. Todos os exemplos apresentados nesta apostila foram copiados após terem sido executados em computador. Assim, não deve haver problemas se você quiser copiá-los e fazer suas próprias experiências.

Toda bibliografia contida neste texto constitui-se de artigos que têm em comum a priorização dos aspectos práticos do conteúdo tratado. Como livro texto de apoio à disciplina recomendo o livro de Francisco Bombim Júnior [Jún02]. Este livro, além de tratar do assunto de aplicações para web, traz uma revisão sobre Java. Recomendo, para aqueles que não conhecem a linguagem, a leitura dos capítulos sobre este assunto antes de ler os capítulos que envolvem exemplos práticos de aplicações para web.

A avaliação da discplina será feita através de um trabalho em grupo de 4 ou 5 pessoas. O trabalho constitui-se no desenvolvimento de uma aplicação para web completa. A natureza do trabalho bem como o prazo para desenvolvê-lo será determinado em comum acordo com a turma.

[5] A fundamentação teórica é vista em maior profundidade e abrangência na disciplina "Sistemas Distribuídos".

Capítulo 2

Aplicações para Web

Toda aplicação para web está baseada no **modelo cliente-servidor**. A finalidade deste capítulo é mostrar este modelo bem como a arquitetura de software dele decorrente.

2.1 O Modelo Cliente-Servidor

O modelo cliente-servidor é um tipo de sistema distribuído[1]. Neste modelo, as tarefas que o software deve realizar são divididas em dois grupos. Esta divisão implica na criação de dois programas. Fala-se, então, em **programa servidor** e **programa cliente**.

A característica principal do modelo cliente-servidor é que a comunicação, via algum protocolo, entre o programa cliente e o programa servidor sempre é realizada por iniciativa do primeiro. A finalidade do programa servidor é atender os pedidos feitos pelo programa cliente.

A adoção do modelo cliente-servidor representa, num certo sentido, uma volta ao passado. Durante muitos anos, na história da Computação, todo o processamento era realizado em um computador central (chamado "main-frame") e acessado a partir de vários terminais. Mais tarde, com o barateamento dos computadores ocorreu um movimento chamado "downsizing", onde o processamento centralizado foi distribuído para vários computadores menores e muito mais baratos.

Com a Internet, fenômeno recente em termos da história da Computação, começaram a se definir os papéis de fornecedor de informações/serviços[2] e de consumidor destas informações/serviços. Há uma tendência, com isso, do processamento voltar a se concentrar em um único ponto.

[1] Um sistema distribuído é um programa de computador que é executado em dois ou mais computadores.

[2] Um exemplo interessante de fornecedor de serviços é o site `http://www.ps2pdf.com` onde o cliente envia um arquivo no formato PostScript (.ps) e recebe o arquivo convertido para o formado PDF (.pdf).

Tomando como referência o modelo cliente-servidor, há duas arquiteturas de software normalmente utilizadas. A arquitetura em duas camadas e a arquitetura em N camadas.

2.1.1 Arquitetura em Duas Camadas

A arquitetura em duas camadas é composta por exatamente dois programas, como mostra a figura 2.1. Um programa corresponde ao programa cliente e outro ao programa servidor. Embora pareça uma divisão bastante óbvia, ela pode apresentar algums problemas. Uma solução possível para estes problemas é a adoção da arquitetura de N camadas [Cha00], discutida na seção 2.1.2.

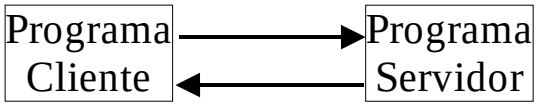

Figura 2.1: Arquitetura Cliente-Servidor em Duas Camadas

Os softwares servidor e cliente estão localizados, normalmente, em computadores diferentes e, claro, interligados através de alguma rede (tipicamente a Internet ou uma Intranet). O software servidor é concebido para atender vários clientes simultaneamente. Este cenário é esquematizado na Figura 2.2.

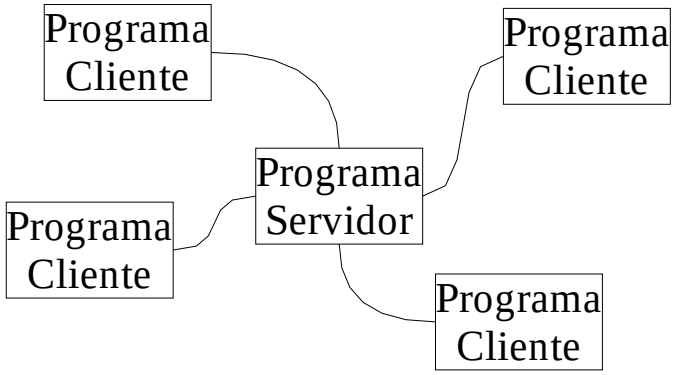

Figura 2.2: Cenário de uma Aplicação Cliente-Servidor em 2 Camadas

2.1.2 Arquitetura em N Camadas

A transformação de uma arquitetura em 2 camadas para uma de N camadas se dá pela divisão da camada associada ao programa servidor [Gou00]. Além

disso, outra mudança que costuma aparecer é o enxugamento da camada cliente[3].

O artigo de Pawlan [Paw01], embora esteja relacionado à plataforma J2EE, é uma excelente fonte de informação sobre a terminologia e conceituação de aplicações baseadas na arquitetura em N camadas.

A arquitetura em N camadas é mostrada na Figura 2.3. Ela indica que cada tarefa de responsabilidade do servidor é realizada por um programa específico. O exemplo típico é deixar para um Sistema Gerenciador de Base de Dados (SGBD) a tarefa de armazenar e recuperar dados.

Figura 2.3: Arquitetura Cliente-Servidor em N Camadas

A principal vantagem desta arquitetura é que, pelo menos em tese, cada camada pode ser substituída sem interferir nas demais. Por exemplo, a troca de banco de dados não deveria afetar o programa que trata das solicitações feita pelos clientes.

2.2 Aplicações Cliente-Servidor via Web

A expressão "aplicação cliente-servidor via web" ou simplesmente "aplicação para web" significa uma aplicação cliente-servidor, com 2 ou N camadas, onde a comunicação entre o programa cliente e o programa servidor é realizada através do protocolo HTTP[4]. A série de quatro artigos de Mahmoud [Mah01a, Mah01b, Mah01c, Mah01d] apresenta um excelente panorama sobre a recente história das aplicações para web (no primeiro artigo) e sobre as tendências atuais (nos três artigos restantes).

Na camada cliente o programa normalmente utilizado é algum browser que, por definição, interpreta dados codificados em HTML e os mostra para o usuário em um formato legível.

Observe que, até aqui, o conceito de aplicação para web é idêntico ao conceito de servidor web. Um servidor web, como o popular Apache[5], tem como finalidade enviar arquivos codificados em HTML para que algum browser possa exibir para o usuário.

[3] Recentemente tem aparecido a expressão "thin client" que significa que o programa da camada cliente não faz nenhum tipo de processamento, limitando-se a exibir as informações recebidas do servidor.

[4] Hypertext Transfer Protocol, http://www.w3.org/Protocols/.

[5] Disponível em http://www.apache.org

O conceito de aplicação web se diferencia do conceito de servidor web quando os arquivos codificados em HTML são gerados em tempo real. O programa servidor, literalmente, gera os arquivos HTML dinamicamente.

O que acontece, portanto, é que toda solicitação feita por um cliente irá gerar algum processamento – específico da aplicação – no servidor. Isto é diferente do que acontece com os servidores web onde o processamento sempre é igual (recuperar e enviar o conteúdo de uma página indicada pelo usuário).

2.2.1 Componentes para Aplicações Web

O programa servidor de aplicação para web **NÃO PRECISA SER DESENVOLVIDO POIS JÁ ESTÁ PRONTO**.

Graças às modernas técnicas de engenharia de software, como a técnica de *frameworks*, as tarefas comuns a todas as aplicações já foram implementadas e testadas. Resta ao desenvolvedor das aplicações criar os **componentes de software** que são específicos para cada aplicação.

No contexto Java, os programas servidores são chamados de ***containers*** ou **servidores JSP**. O Tomcat (veja Apêndice B) é um exemplo de servidor JSP.

Capítulo 3

Desenvolvimento de Aplicações para Web

Definirse uma aplicação adotará uma arquitetura de 1, 2, 3 ou N camadas resolve apenas uma parte do problema de desenvolver software. Em cada um dos casos há, ainda, a indefinição sobre a **forma de interação** dos elementos de cada uma das camadas.

A experiência acumulada têm mostrado que para aplicações simples o desenvolvedor têm um grande grau de liberdade para conceber e estruturar a aplicação. O cilo básico de execução formado pela tríade requisição-processamento-resposta do modelo cliente-servidor pode ser implementado sem maiores dificuldades. Já para as aplicações mais complexas os riscos do desenvolvedor ser "tragado" em um emaranhado de algoritmos é grande.

A solução adotada, quase que por unanimidade, para administrar as interações em aplicações mais complexas passa pelo uso de vários padrões de projeto[1]. O padrão mais empregado é chamado MVC.

Este capítulo apresenta, ainda que informalmente, o padrão MVC e também seu uso no desenvolvimento de aplicações para web em Java. O capítulo termina com uma seção sobre como organizar os diversos tipos de arquivos existentes em uma aplicação típica de modo a produzir efetivamente protótipos consistentes.

3.1 O Padrão de Projeto MVC

O padrão MVC, que significa **Modelo-Visualização-Controle**, reafirma a importância da clara separação entre as questões de interface (visualização dos dados por parte do usuário) e de representação dos dados do problema sendo modelado.

A problemática envolvida na visualização dos dados de um problem anão tem, ou pelo menos não deveria ter, nenhuma relação com a representação

[1]No original, *design patterns*.

destes dados. Da mesma forma, a problemátia envolvida na representação dos dados não tem nehuma relação com a visualização dos mesmos. Falta, portanto, um elo de ligação entre estas duas partes. Este elo, chamado de controle, sabe como fazer com que o modelo e a visualização "conversem entre si" sem com isso criar dependências entre eles.

A Figura 3.1 esquematiza as relações entre os elementos MVC. Noque que o controle permite a comunicação nos dois sentidos entre o modelo e a visualização. Isto significa que sempre que há mudança nos dados esta deve ser propagada até a visualização para que o usuário tome conhecimento. Da mesma forma, o usuário pode ser o elemento provocador da mudança (a mudança nas informações que estão sendo mostradas deve se refletir no modelo).

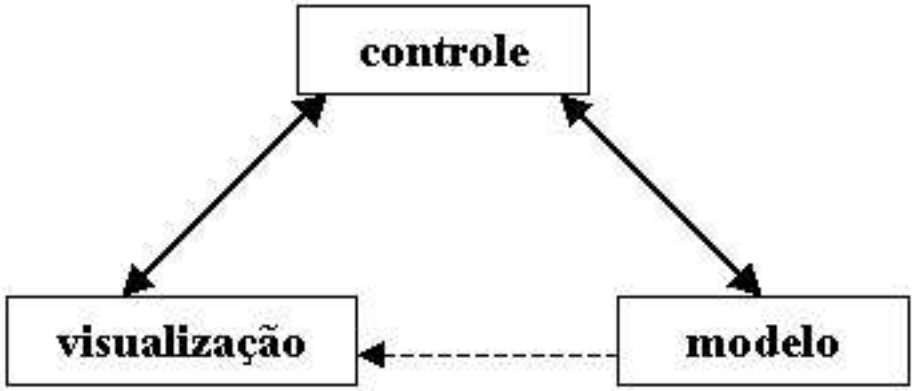

Figura 3.1: O Padrão de Projeto MVC

3.2 Arquitetura de Modelo 2

Existem diversas formas de organizar uma aplicação para web baseada em Java. No entanto, a forma que está sendo adotada quase por unanimidade é a chamada **Arquitetura de Modelo 2** [Sec99] mostrado na Figura 3.2. Este modelo explora o *design pattern* MVC visando uma clara separação entre a lógica de resolução do problema e a interface da aplicação [Mar01, Ada01].

O Modelo 2 tem a seguinte dinâmica de eventos. O usuário, através de seu browser, requisita uma página (evento 1) ao servidor JSP. A requisição é recebida por um servlet ou página JSP que atua como controlador. A requisição pode implicar na instanciação ou uso (evento 2) de um JavaBean que implementa a lógica de resolução do problema. Este JavaBean pode acessar outros sistemas de informação como banco de dados ou sistemas legados. O controlador então direciona (evento 3) o fluxo de execução para

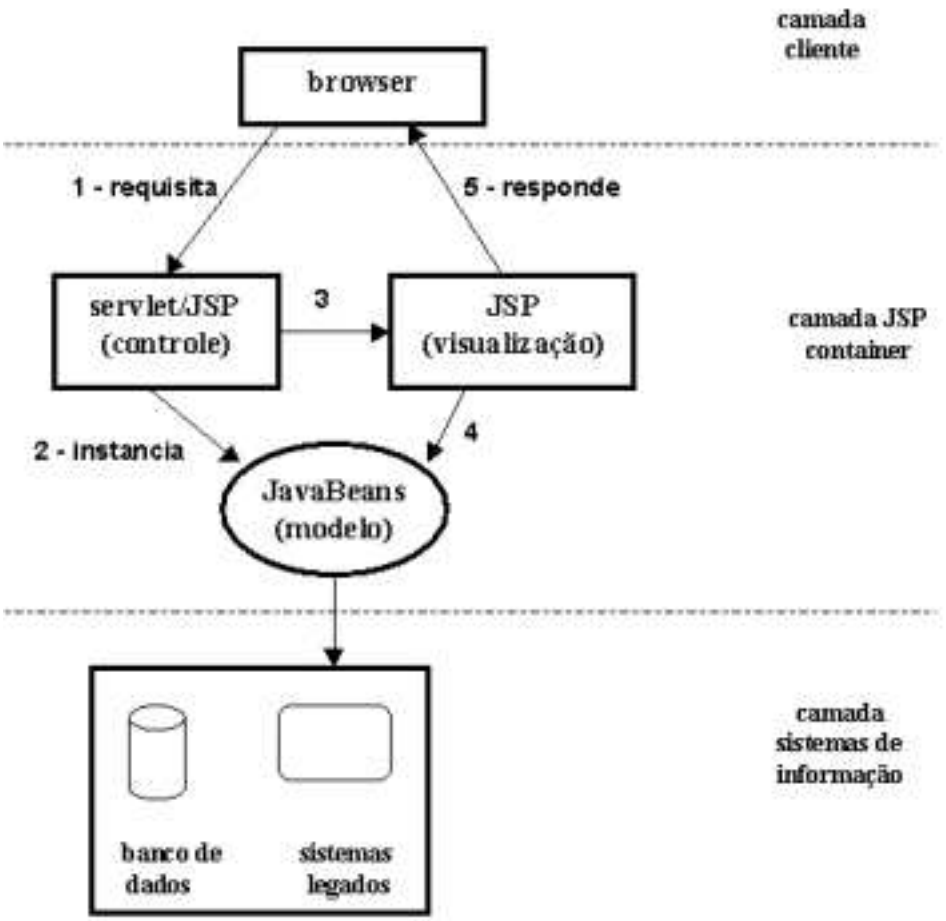

Figura 3.2: Arquitetura de Modelo 2

uma página JSP responsável por montar a página requisitada pelo usuário. Esta página JSP pode acessar (evento 4) o JavaBean para obter informações (conteúdo dinâmico). Finalmente a página JSP é enviada (evento 5) de volta para ser exibida pelo browser.

3.3 Dinâmica de Desenvolvimento

O desenvolvimento de aplicações para web implica no gerenciamento simultâneio de vários arquivos. Muitas tarefas precisam ser realizadas até ser possível gerar todos os componentes que devem ser inseridos no servidor. Uma forma de administrar esta complexidade é organizar os diversos arquivos em subdiretórios.

Cada ferramenta de desenvolvimento costuma organizar os subdiretórios

de uma forma particular. Com o intuito de não nos atermos a nenhuma ferramenta em particular, sugere-se uma forma de organização própria[2].

A Tabela 3.1 mostra a estrutura de diretórios recomendada. Todos os subdiretórios são relativos ao diretório onde a aplicação está sendo desenvolvida.

Subdiretório	Finalidade
\src\html	conter todas os arquivos `.html`, `.jsp`, `.css`, `.gif`, etc. Estes arquivos podem estar organizados em qualquer estrutura de subdiretórios.
\src\java	conter todos os programas fontes das classes Java (arquivos `.java`) usados na apliccação.
\classes	conter todos os programas compilados das classes Java (arquivos `.class`) usados na aplicação.
\meta	conter o arquivo XML que contém o descritor de instalação (*deployment descriptor*) da aplicação.
\jars	conter as bibliotecas (arquivos `.jar`) usadas na aplicação.
\dist	conter o arquivo `.war` que, por definição, contém todos os arquivos que fazem parte da aplicação. O nome deste arquivo representa o nome da aplicação. É este arquivo que deve ser instalado no container.

Tabela 3.1: Estrutura de diretórios para desenvolvimento de aplicações web

[2]A organização sugerida está inspirada, na verdade, para facilitar o uso da ferramenta Ant (veja Apêndice A)

Capítulo 4

JavaBeans

O paradigma de Programação Orientada a Objetos (POO) tem na **reusabilidade de software** um dos seus principais fundamentos. Os sistemas têm se tornando cada vez mais complexos e, ao mesmo tempo, devem ser desenvolvidos em cada vez menos tempo[1].

Diversas técnicas vêm sendo desenvolvidas nos últimos anos para tratar esta questão. Em Java, a abordagem adotada foi o desenvolvimento de **componentes de software**, chamados de **JavaBeans**.

O objetivo deste capítulo é mostrar como desenvolver JavaBeans no contexto das aplicações para web. Os JavaBeans são amplamente utilizados em conjunto com a tecnologia de JavaServer Pages (JSP), discutida no Capítulo 5.

A motivação inicial associada aos JavaBeans, no entanto, estava mais próxima do conceito de programação visual, isto é, na concepção e implementação da interface gráfica das aplicações. O ambiente Delphi da Borland, por exemplo, é um dos casos mais bem sucedidos desta abordagem na concepção de software.

4.1 Definição

Um JavaBean é uma classe cuja estrutura e sintaxe obedecem a um conjunto de regras. Estas regras são adotadas **voluntariamente** pelo desenvolvedor da classe.

Qualificar uma classe como sendo um JavaBean é exclusivamente uma questão de compromisso assumido pelo desenvolvedor. O compilador não é capaz de detectar se uma classe é ou não é um JavaBean. Do ponto de vista

[1]Tem sido frequente o uso da expressão em inglês *time-to-market* para representar a idéia de que um produto pode perder seu valor se não for desenvolvido em um determinado período de tempo. Este período é atrelado em muito maior grau a fatores comercias do que a fatores técnicos.

técnico, isto implica que um JavaBean não extende nenhuma superclasse nem implementa nenhuma interface.

Por outro lado, se uma classe for um JavaBean então editores compatíveis com o padrão JavaBeans conseguem manipular visualmente instâncias destas classes permitindo que elas sejam configuradas em tempo de projeto.

4.2 Regras

As regras que definem um JavaBean estão descritas na Tabela 4.1.

4.3 Exemplo: Um JavaBean para JSP

Embora os JavaBeans usados para modelar objetos gráficos e os usados em conjunto com páginas JSP compartilhem a mesma definição conceitual, existem duas diferenças práticas.

Primeiro, JavaBeans manipulados por editores gráficos precisam implementar a interface `java.io.Serializable` pois são armazenados em arquivos enquanto que os JavaBeans para JSP normalmente só existem em memória.

Segundo, nos JavaBeans que modelam objetos gráficos há uma tendência dos nomes dos métodos `setXX` e `getXX` coincidirem com os nomes dos atributos da classe pois estes representam, quase sempre, objetos visuais que necessitam ser configurados.

O exemplo a seguir mostra como usar JavaBeans no contexto das aplicações para web com a tecnologia de JSP. O artigo de Duane [FK00] é uma excelente fonte de informação sobre as relações entre JavaBeans e JSP.

O JavaBean `EstudanteBean` modela um estudante caracterizado por seu nome, três notas e a média destas notas. As Figuras 4.1 e 4.2[2] mostram o código Java correspondente ao referido JavaBean.

Observe, na Figura 4.1, os métodos `setNota1`, `setNota2` e `setNota3`. Estes métodos, juntamente com os seus complementares `getNota1`, `getNota2` e `getNota3` na Figura 4.2, fazem com que o usuário deste JavaBean pense em termos das propriedades **nota1**, **nota2** e **nota3**.

[2] A divisão em duas figuras é motivada por razões estritamente didáticas. Há somente um arquivo `EstudanteBean.java`.

Regra	Descrição
1 - Construtor	Deve haver um construtor sem nenhum argumento. Se o JavaBean não possui nenhum construtor declarado explicitamente então o interpretador Java considera, já que faz parte da definição da linguagem, a existência de um construtor implícito sem argumentos que não faz nada.
2 - Persistência	Todo JavaBean pode implementar a interface `java.io.Serializable`. Esta regra não precisa ser necessariamente obedecida. Neste caso a instância do JavaBean fica impossibilitado de ser salvo em arquivo ou ser enviado de um computador para outro numa aplicação distribuída.
3 - Acesso aos atributos	Se o JavaBean possui um atributo na forma `tipo atributo;` então o acesso ao valor deste atributo deve ser feito através do método `public tipo getAtributo()`.
4 - Mudança no valor dos atributos	Se o JavaBean possui um atributo na forma `tipo atributo;` então a mudança do valor deste atributo deve ser feito através do método `public void setAtributo(tipo valor)`.
5 - Propriedades	Um JavaBean pode ter, além de seus atributos, propriedades. Estas são caracterizadas pela existência de métodos nas formas `public void setPropriedade(tipo valor)` e/ou `publit tipo getPropriedade()`.
5 - Encapsulamento de atributos	Todos os atributos devem estar encapsulados, isto é, devem ser qualificados como `private` ou, ao menos, `protected`.

Tabela 4.1: Regras de Definição de JavaBeans

```
public class EstudanteBean {
  private String nome;
  private float[] notas;
  private float media;

  public EstudanteBean() {
    nome     = "";
    notas    = new float [3];
    notas[0] = 0.0f;
    notas[1] = 0.0f;
    notas[2] = 0.0f;
    media    = 0.0f;
  }

  public void setNome(String valor) {
    nome = valor;
  }

  public void setNota1(float valor) {
    notas[0] = valor;
    calculeMedia();
  }

  public void setNota2(float valor) {
    notas[1] = valor;
    calculeMedia();
  }
  public void setNota3(float valor) {
    notas[2] = valor;
    calculeMedia();
  }

  protected void calculeMedia() {
    media = (notas[0] + notas[1] + notas[2]) / 3;
  }
```

Figura 4.1: Um JavaBean modelando um estudante (parte a).

```
  public String getNome() {
    return nome;
  }

  public float getMedia() {
    return media;
  }

  public float getNota1() {
    return notas[0];
  }

  public float getNota2() {
    return notas[1];
  }

  public float getNota3() {
    return notas[2];
  }
}
```

Figura 4.2: Um JavaBean modelando um estudante (parte b).

Capítulo 5

JavaServer Pages (JSP): Sintaxe Básica

O desenvolvimento de aplicações para web depende de alguma tecnolgia capaz de gerar conteúdos dinamicamente. Aplicações baseadas em Java utilizam a tecnologia de JavaServer Pages (JSP) [Jún02, jGu00] para esse fim.

Neste capítulo, a tecnolgia de JSP é mostrada inicialmente através de exemplos (seção 5.2) e, só então, sua sintaxe é formalmente apresentada.

5.1 Aplicação para Web Baseada em JSP

Aplicaçoões para web baseadas na tecnologia de JSP são organizadas seguindo a arquitetura mostrada na Figura 5.1.

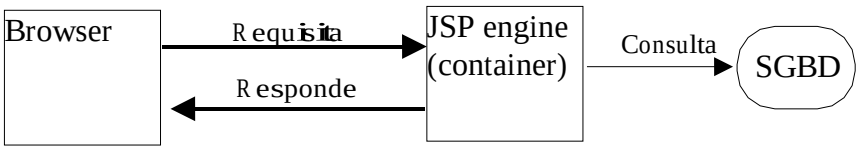

Figura 5.1: Arquitetura de Aplicação Web Baseada em JSP

Nesta arquitetura o programa cliente normalmente é o próprio browser rodando no computador do usuário. Já o programa servidor é alguma implementação da especificação da **máquina JSP (JSP engine)**, também chamado de **container**. Completando a arquitetura, embora seja opcional, é comum a integração entre aplicações e algum banco de dados. Reforçando o conceito de modelo cliente-servidor em 3 camadas, o container e o banco de dados não precisam estar necessariamente rodando em um mesmo computador.

O programa cliente pode ser qualquer browser capaz de exibir um texto codificado em HTML. Este programa funciona, portanto, como a interface da aplicação.

Há, no entanto, diferenças importantes entre os principais browsers do mercado quanto à extensão e forma como a especificação HTML é implementada. Recomenda-se, portanto, que o texto HTML gerado pelo container seja relativamente simples. Deve-se evitar o processamento no próprio browser[1].

Existem diversas implementações do container JSP disponíveis no mercado, tanto na modalidade de código fechado e pago[2] como de código aberto e sem nenhum custo. Uma das implementações de código aberto mais conhecidas é o **Tomcat** produzido pela Fundação Apache[3]. O **Tomcat**, descrito no Anexo B, é utilizado nos exemplos discutidos ao longo deste curso.

O acesso ao banco de dados é realizado através da tecnologia JDBC, apresentada no Capítulo 7. Todos os principais fornecedores de banco de dados oferecem drivers JDBC. Com estes drivers, um programa escrito em Java tem acesso, via SQL, ao banco de dados. Nos exemplos apresentados neste curso utiliza-se o driver para MySQL[4]. Ambos, o driver e o banco de dados, são gratuitos.

Neste contexto, utilizar uma aplicação implica numa interação entre o browser do usuário e o container. O usuário fornece um endereço (URL) ao browser e este requisita ao container o documento correspondente. O container, ao receber a requisição, responde ao browser enviando o documento requisitado. O conteúdo deste documento pode ser estático (quando o URL termina com `.html`) ou dinâmico (quando o URL termina com `.jsp`). Neste caso, algum processamento é realizado, incluindo-se aí um possível acesso ao banco de dados. O resultado deste processamento é incorporado ao documento que sempre está codificado em HTML.

5.1.1 Estrutura de Diretórios

Uma aplicação para web baseada em Java é composta por um conjunto de arquivos de diversos formatos. Tipicamente os formatos encontrados são HTML, JSP, `.gif`, `.jpg`, classes Java (extensão `.class`), bibliotecas Java (extensão `.jar`), XML, etc.

Todos os arquivos devem estar organizados em uma estrutura de diretórios específica. Esta estrutura é mostrada na Tabela 5.1. O nome `aplic` representa o nome da aplicação. Cada aplicação dentro do container deve ter um nome único.

[1] Esta recomendação deve ser vista com cautela. Na prática observa-se que muitos sites de grandes empresas só funcionam se o usuário estiver usando algum browser específico.

[2] Por exemplo, gigantes da Informática como IBM, Borland, BEA, oferecem suas versões.

[3] Disponível em `http://jakarta.apache.org/tomcat`.

[4] Dsiponível em `http://www.mysql.com`.

Diretório	Finalidade
aplic	Contém todos os arquivos `.html`, `.jsp`, `.gif`, etc. que compõem a aplicação. Estes arquivos podem estar organizados, por sua vez, em qualquer estrutura de diretórios.
aplic\WEB-INF	Contém o arquivo `web.xml`. Este arquivo é discutido na seção 5.2.1.1.
aplic\WEB-INF\classes	Contém as classes Java (arquivos `.class`) usadas na aplicação.
aplic\WEB-INF\lib	Contém as bibliotecas (arquivos `.jar`) usadas na aplicação.

Tabela 5.1: Estrutura de Diretórios para Aplicação Web em Java.

5.2 Aplicações Exemplo Iniciais

Uma página JSP é basicamente um texto escrito em HTML e acrescido de alguns elementos sintáticos escritos em JSP. A parte escrita em HTML corresponde ao conteúdo estático da página e a parte escrita em JSP corresponde à parte dinâmica.

5.2.1 exemplo1: Hora Atualizada Manualmente

Como uma primeira aproximação de JSP, considere a aplicação para web **exemplo1**. Esta aplicação tem como objetivo mostrar para o usuário uma página em que aparece a hora atual. A interface desta aplicação é mostrada na Figura 5.2.

Figura 5.2: Interface da aplicação exemplo1

Vamos considerar, inicialmente, como construir esta aplicação SEM utilizar a tecnologia JSP. A aplicação, neste caso, é formada pelos arquivos indicados na Tabela 5.2.

Arquivo	Finalidade
exemplo1\index.html	Gerar a página inicial da aplicação, detalhado na Figura 5.3.
exemplo1\WEB-INF\web.xml	Descritor de instalação[5] da aplicação, detalhado na Figura 5.4.

Tabela 5.2: Arquivos que formam a aplicação exemplo1.

5.2.1.1 Descritor de Instalação

Várias características de uma aplicação para web podem ser configuradas através do arquivo `web.xml`. Cada aplicação deve ter necessariamente um descritor de instalação, mesmo que ele não esteja configurando nada. Neste caso a aplicação usa as configurações padrão definidas pelo container.

O conteúdo mínimo do arquivo descritor de instalação é mostrado na Figura 5.4[6].

```
<html>
<head>
<title> Aplicação web - exemplo1 </title>
</head>
<body>
<h1> Aplicaçao web - exemplo1 </h1>
<p>A hora atual é 17:33 hs.</p>
</body>
</html>
```

Figura 5.3: Aplicação exemplo1: arquivo index.html.

```
<!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>
```

Figura 5.4: Aplicação exemplo1: arquivo web.xml.

[6]Este assunto, configuração do descritor de instalação, não é tratado neste curso.

A aplicação **exemplo1** é perfeitamente funcional, isto é, pode ser executada a partir dos seus arquivos `index.html` e `web.xml`. O problema com esta aplicação é que o **conteúdo** do arquivo `index.html` precisa ser atualizado manualmente se quisermos garantir a consistência da hora informada pelo servidor.

5.2.2 exemplo2: Hora Atualizada Automaticamente

A aplicação `exemplo2`, cuja interface é mostrada na Figura 5.5, tem a mesma finalidade da aplicação `exemplo1`. Porém, agora, a indicação da hora atual é realizada automaticamente. Isto é possível com a introdução de elementos de JSP.

Figura 5.5: Interface da aplicação exemplo2

A aplicação é formada pelos arquivos indicados na Tabela 5.3.

Arquivo	Finalidade
exemplo2\index.jsp	Gerar a página inicial da aplicação, detalhado na Figura 5.6.
exemplo2\WEB-INF\web.xml	Descritor de instalação da aplicação, detalhado na Figura 5.4. Note que este arquivo é igual ao usado na aplicação `exemplo1`.

Tabela 5.3: Arquivos que formam a aplicação exemplo2.

Observe, no arquivo `index.jsp`, que foram incluídas expressões delimitadas pelos símbolos `<%` e `%>`. Estas expressões correspondem a **fragmentos de código Java** e definem a parte dinâmica da página.

As mudanças no arquivo `index.jsp` da aplicação `exemplo2` em relação ao arquivo `index.html` da aplicação `exemplo1` são:

- a extensão passou de `.html` para `.jsp`. Esta nova extensão indica que

```
<html>
<head>
<title> Aplicação web - exemplo2 </title>
</head>
<body>
<%!
java.util.Calendar agora;
int hora;
int minuto;
%>

<%
agora = new java.util.GregorianCalendar();
hora = agora.get(java.util.Calendar.HOUR);
minuto = agora.get(java.util.Calendar.MINUTE);
%>

<h1>Aplicação web - exemplo2</h1>

<p>A hora atual é <%= hora + ":" + minuto%> hs.</p>

</body>
</html>
```

Figura 5.6: Aplicação exemplo2: arquivo index.jsp.

o documento possui algum conteúdo que precisa ser gerado dinamicamente pelo servidor.

- expressões delimitadas pelos símbolos `<%!` e `%>` definem variáveis Java. No caso em questão, estão sendo definidas três variáveis (agora – um objeto da classe java.util.Calendar – e hora e minuto – variáveis do tipo inteiro).
- logo após a definição das variáveis há um fragmento de código Java (delimitado pelos símbolos `<%` e `%>`). É este fragmento que, quando executado, determinará a hora e o minuto atual no servidor. De modo geral, pode-se escrever qualquer algoritmo usando-se a linguagem Java.
- a expressão delimitada pelos símbolos `<%=` e `%>` é avaliada pelo container, seu valor é incorporado ao código HTML existente.

Se o usuário olhar o código fonte da página `index.jsp` que está sendo

exibida pelo seu browser ele encontrará **apenas código HTML**. Esta característica é extremamente interessante pois o fragmento de código Java usado para gerar dinamicamente as informações **não fica disponível para o usuário**[7].

5.2.3 exemplo3: Hora Atualizada Automaticamente via JavaBeans

A aplicação `exemplo3`, cuja interface é mostrada na Figura 5.7, tem a mesma finalidade da aplicação `exemplo2`. Porém, agora, a indicação da hora atual é realizada automaticamente via um **JavaBean**.

Figura 5.7: Interface da aplicação exemplo3

Esta aplicação é formada pelos arquivos indicados na Tabela 5.4.

Arquivo	Finalidade
exemplo3\index.jsp	Gerar a página inicial da aplicação, detalhado na Figura 5.8.
exemplo3\WEB-INF\web.xml	Descritor de instalação da aplicação (detalhado na Figura 5.4). Note que este arquivo é igual ao usado nas aplicaçções `exemplo1` e `exemplo2`.
exemplo3\WEB-INF\classes\ ljk\beans\HoraAtualBean.class	JavaBean capaz de determinar a hora atual (detalhado na Figura 5.9).

Tabela 5.4: Arquivos que formam a aplicação exemplo3.

Observe como o arquivo `index.jsp` ficou menor e mais fácil de ser entendido. Outro aspecto importante é a separação, agora mais bem definida,

[7] A título de comparação, quando a linguagem JavaScript é usada para gerar conteúdos dinâmicos o fragmento de código usado para esse fim fica visível para o usuário.

entre as tarefas do web designer – que conhece HTML e possivelmente não conhece Java – e do desenvolvedor da parte lógica da aplicação – que pode não conhecer HTML e conhece Java.

```
<jsp:useBean id="agora"
             class="ljk.beans.HoraAtualBean"
             scope="session"/>
<html>
<head>
<title>Aplicação web - exemplo3</title>
</head>
<body>
<h1>Aplicação web - exemplo3</h1>

<p>A hora atual é <%= agora.getHoraAtual()%> hs.</p>

</body>
</html>
```

Figura 5.8: Aplicação exemplo3: arquivo index.jsp.

Observe que o arquivo `index.jsp` inicia com a tag `<jsp:useBean>`. Esta tag, definida na linguagem JSP, especifica:

id="agora" um objeto chamado `agora`. Este objeto é capaz de criar, como valor de retorno do método `getHoraAtual()`, um string que representa a hora atual.

class="ljk.beans.HoraAtualBean" a classe do objeto `agora`.

scope="session" o ciclo de vida do objeto `agora`. Neste caso, está indicando que o objeto existe durante toda sessão[8].

[8] O conceito de sessão é explicado no Capítulo 6.

```
package ljk.beans;

import java.util.Calendar;

public class HoraAtualBean {
    private Calendar agora;
    private int hora;
    private int minuto;

    public HoraAtualBean() {
    }

    public String getHoraAtual() {
        agora = new java.util.GregorianCalendar();
        hora = agora.get(java.util.Calendar.HOUR);
        minuto = agora.get(java.util.Calendar.MINUTE);
        return (hora + ":" + minuto);
    }
}
```

Figura 5.9: Aplicação3: arquivo ljk.beans.HoraAtualBean.java.

5.2.4 exemplo4: Dobrando valor

A aplicação `exemplo4` tem como finalidade calcular o dobro do número digitado pelo usuário. Ela mostra uma das formas de uso mais frequentes da tecnologia JSP, qual seja, o processamento de formulários preenchidos pelo usuário.

Ao acessar a aplicação, o usuário deve preencher um formulário, digitando um número inteiro no campo especificado (veja a Figura 5.10). Este formulário é enviado ao servidor que extrai o número digitado e retorna para o usuário uma nova página, como mostra a Figura 5.11, que indica o número digitado e o dobro deste número.

Figura 5.10: Formulário preenchido pelo usuário da aplicação exemplo4

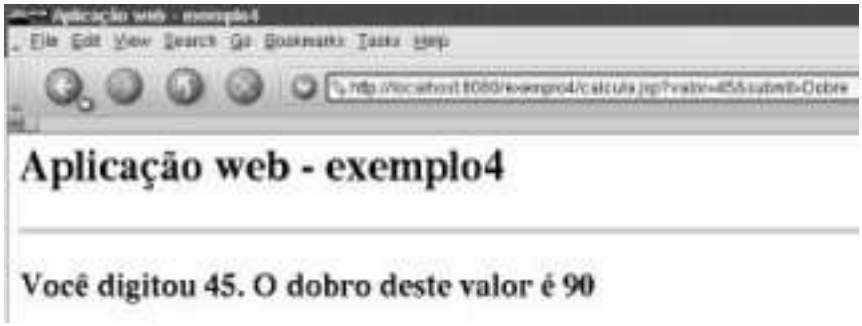

Figura 5.11: Resposta do servidor da aplicação exemplo4

A aplicação é formada pelos arquivos indicados na Tabela 5.5.

Observe, na Figura 5.13, que o javabean está utilizando uma biblioteca, a classe `ljk.dobrador.Dobrador` que está definida no arquivo `.jar` no subdiretório `\lib` da aplicação.

Observe que no arquivo `index.html` (Figura 5.12) o formulário possui o campo "valor". O número inteiro digitado pelo usuário neste campo é processado no arquivo `calcula.jsp` (Figura 5.14) através da expressão `request.getParameter("valor");`.

Arquivo	Finalidade
exemplo4\index.html	Gerar a página inicial da aplicação, detalhado na Figura 5.12.
exemplo4\calcula.jsp	Gerar a página retornada ao usuário, detalhado na Figura 5.14.
exemplo4\WEB-INF\web.xml	Descritor de instalação da aplicação, detalhado na Figura 5.4. Note que este arquivo é igual ao usado nas aplicaçções `exemplo1`, `exemplo2` e `exemplo3`.
exemplo4\WEB-INF\classes\ ljk\exemplo\ExemploBean.class	JavaBean capaz de calcular o dobro de um número inteiro, detalhado na Figura 5.13.
exemplo4\WEB-INF\lib\ lib_exemplo.jar	Classe Java que só possui um método que retorna um número inteiro que é o dobro do número passado como argumento.

Tabela 5.5: Arquivos que formam a aplicação exemplo4.

```
<html>
<head>
<title>Aplicação web - exemplo4</title>
</head>
<body>
<h1>Aplicação web - exemplo4</h1>

<form type=get action=calcula.jsp>

Ditite um número <input type=text name=valor> <br>

<INPUT TYPE=submit name=submit Value="Dobre"> </form>
</body>
</html>
```

Figura 5.12: Aplicação exemplo4: arquivo index.html.

```
package ljk.exemplo;

import ljk.dobrador.Dobrador;

public class ExemploBean {
    public int getDobro(int valor) {
    return Dobrador.dobre(valor);
    }
}
```

Figura 5.13: Aplicação exemplo4: arquivo ljk.exemplo.ExemploBean.java.

```
<jsp:useBean id="dobrador" class="ljk.exemplo.ExemploBean"/>
<html> <head> <title>Aplicação web - exemplo4</title> </head>
<body>

<%! String param;%>
<%! int numero;%>
<% param = request.getParameter("valor");%>
<% numero = java.lang.Integer.parseInt(param);%>

<h1> Aplicação web - exemplo4</h1> <hr>
<h2>Você digitou <%= param %>. O dobro deste valor
é <%= dobrador.getDobro(numero)%> </h2>
</body>
</html>
```

Figura 5.14: Aplicação exemplo4: arquivo calcula.jsp.

5.3 Sintaxe

A linguagem utilizada na tecnologia JSP envolve as categorias sintáticas de **diretivas**, **scripts** e **ações padrão**.

5.3.1 Diretivas

Diretivas são mensagens destinadas ao container JSP que configuram algumas propriedades da página. Elas seguem a sintaxe `<%@ diretiva atributo="valor"%>`. Há três diretivas possíveis: `page`, `include` e `taglib`[9]. A cada diretiva podem estar associadas de 0 a N pares de atributo e valor.

5.3.1.1 page

A diretiva `page` define algumas propriedades da página que a contém. A Tabela 5.6 mostra algumas[10] das propriedades que podem ser definidas nesta diretiva.

Propriedade	Finalidade
`<%@page import="classes"%>`	Deixar disponível para os scripts da página o conjunto de classes indicadas como valor deste atributo. Corresponde à declaração `import` de Java.
`<%@page isErrorPage="true|false"%>`	Indicar se a página é página de erro. o valor default é `false`. Quando um erro de processamento ocorrer esta página será enviada ao cliente no lugar da mensagem de erro gerada pelo container JSP.
`<%@page errorPage="URL"%>`	Indicar qual página representa a página de erro.

Tabela 5.6: Propriedades da diretiva `page`

.

5.3.1.2 include

A diretiva `include` inclui na página o conteúdo de outros arquivos que podem estar tanto no formato HTML como JSP. Ela tem a seguinte sintaxe: `<%@include file="arquivo"%>`

[9] Neste curso só serão discutidas e utilizadas as diretivas page e include.
[10] A relação completa de propriedades pode ser encontrada em [Jún02].

Com esta diretiva é possível gerenciar melhor os conteúdos das páginas através do conceito de *templates* [Gea00]. Por exemplo, se todas as páginas de uma aplicação apresentam uma espécie de cabeçalho – o logotipo e endereço da empresa, por exemplo – então o fragmento de código deste cabeçalho pode ser colocado em um único arquivo e incluído nas páginas em que for necessário. Alterando apenas a página de cabeçalho teremos uma aplicação consistente (mesma aparência).

5.3.1.3 Aplicação exemplo5: uso de diretivas

A aplicação `exemplo5` mostra, na prática, como usar as diretivas apresentadas na seção 5.3.1.

Na aplicação `exemplo4` (seção 5.2.4, página 40), tudo funciona muito bem desde que o usuário digite um número inteiro. Se ele digitar qualquer outra coisa acontecerá um erro e, no lugar da página esperada, aparecerá a página de erro padrão do servidor JSP (mostrada na Figura 5.15).

Figura 5.15: Página de erro padrão do Tomcat.

A aplicação `exemplo5` é formada pelos arquivos indicados na Tabela 5.7. Dois pontos diferenciam esta aplicação da `exemplo4`. O primeiro ponto é a criação de um cabeçalho (definido pelo fragmento de código HTML, mostrado na Figura 5.16) usado em todas as páginas. O segundo ponto é a criação de uma página de erro, detalhada na Figura 5.19.

Arquivo	Finalidade
exemplo5\cabecalho.html	Gerar a o cabeçalho das páginas da aplicação, detalhado na Figura 5.16.
exemplo5\index.jsp	Gerar a página inicial da aplicação, detalhado na Figura 5.17.
exemplo5\calcula.jsp	Gerar a página retornada ao usuário caso não ocorra erro, detalhado na Figura 5.18.
exemplo5\erro.jsp	Gerar a página de erro retornada ao usuário quando ocorre um erro, detalhado na Figura 5.19.
exemplo5\WEB-INF\web.xml	Descritor de instalação da aplicação, detalhado na Figura 5.4. Note que este arquivo é igual ao usado nas aplicaçções exemplo anteriores.
exemplo5\WEB-INF\classes\ ljk\exemplo\ExemploBean.class	JavaBean capaz de calcular o dobro de um número inteiro (detalhado na Figura 5.13, página 42).
exemplo5\WEB-INF\lib\ lib_exemplo.jar	Classe Java que só possui um método que retorna um número inteiro que é o dobro do número passado como argumento.

Tabela 5.7: Arquivos que formam a aplicação exemplo5.

```
<h1> Aplicação web exemplo5 <br> Uso de Diretivas </h1>
```

Figura 5.16: Aplicação exemplo5: arquivo cabecalho.html.

```
<html>
<head>
<title>Aplicação web - exemplo5</title>
</head>
<body>
<%@include file="cabecalho.html"%>
<form type=get action=calcula.jsp>

Ditite um número <input type=text name=valor>
<br>

<INPUT TYPE=submit name=submit Value="Dobre">
</form>
</body>
</html>
```

Figura 5.17: Aplicação exemplo5: arquivo index.jsp.

```
<%@page errorPage="erro.jsp"%>
<jsp:useBean id="dobrador" class="ljk.exemplo.ExemploBean"/>
<html>
<head>
<title>Aplicação web - exemplo5</title>
</head>
<body>

<%! String param;%>
<%! int numero;%>
<% param = request.getParameter("valor");%>
<% numero = java.lang.Integer.parseInt(param);%>

<%@include file="cabecalho.html"%>

<h2>Você digitou <%= param %>. O dobro deste
valor é <%= dobrador.getDobro(numero)%> </h2>
</body>
</html>
```

Figura 5.18: Aplicação exemplo5: arquivo calcula.jsp.

```
<%@page isErrorPage="true"%>
<html>
<head><title>Aplicação web - exemplo5</title></head>

<body>
<%@include file="cabecalho.html"%>
<h3>Página de Erro - Você não digitou um número!</h3>
<h3><a href="index.jsp">Tente novamente</a></h3>
</body>
</html>
```

Figura 5.19: Aplicação exemplo5: arquivo erro.jsp.

5.3.2 Scripts

Scripts são fragmentos de código Java usados para auxiliar na geração de conteúdos dinâmicos das páginas. Eles são divididos em três classes: **declarações**, **expressões** e **scriptles**.

5.3.2.1 Declarações

As declarações servem para definir variáveis, com escopo limitado à página, usadas nas outras classes de scripts.

A sintaxe de uma declaração é idêntica à usada para declarar variáveis em um programa Java e delimitada pelos símbolos `<%!` e `%>`. Assim, por exemplo, a declaração Java `int pecasEmEstoque;` é declarada em JSP como `<%! int pecasEmEstoque;%>`.

5.3.2.2 Expressões

Expressões são expressões Java – normalmente o valor de uma variável ou valor de retorno de algum método de um objeto – que são convertidas para um string e incluídas na página de saída (aquela que será enviada ao cliente).

A sintaxe de uma expressão é idêntica à usada na linguagem Java e delimitada pelos símbolos `<%=` e `%>`. Por exemplo, suponha que na página haja uma declaração do tipo `int pecasEmEstoque = 0;`. Se quisermos incluir esta informação na página que o cliente irá visualizar no seu browser então bastará usarmos a expressão `<p>Atualmente há <%= pecasEmEstoque %> peças no estoque.<p>`.

Um aspecto interessante das expressões é que, uma vez convertidas para string, deixam literalmente de existir na página que será enviada ao cliente. Assim, por exemplo, se um usuário for observar o código fonte da página recebida do servidor ele verá o código HTML `<p>Atualmente há 0 peças no estoque.</p>`. **Não há como saber que parte da página HTML foi gerada dinamicamente.**

5.3.2.3 Scriptlets

Scriptlets são fragmentos de código Java delimitados pelos símbolos `<%` e `%>`. Eles possibilitam a criação de algoritmos que expressam a lógica do problema a ser resolvido. Por exemplo, veja a aplicação `exemplo6`.

Para se fazer um comentário – prática recomendável em termos de documentação do scriptlet – deve-se usar a sintaxe `<%-- comentário --%>`.

5.3.2.4 Aplicação exemplo6: uso de scripts

O objetivo da aplicação `exemplo6` é calcular a média de três notas de um aluno. O usuário digita as três notas em um formulário, mostrado na

Figura 5.20, e as envia ao servidor. As notas são processadas (cálculo da média) e um página, mostrada na Figura 5.21, é retornada ao cliente. Se a média for maior ou igual a 5.75 então a página retornada indica que o aluno está aprovado.

Figura 5.20: Interface da aplicação exemplo6: formulário

Figura 5.21: Interface da aplicação exemplo6: resposta do servidor

A Tabela 5.8 indica os arquivos que fazem parte da aplicação `exemplo6`.

Arquivo	Finalidade
exemplo6\cabecalho.html	Gerar o cabeçalho das páginas da aplicação. Segue o mesmo princípio do cabeçalho da aplicação `exemplo5`, detalhado na Figura 5.16 (página 45).
exemplo6\index.jsp	Gerar a página inicial da aplicação, detalhado na Figura 5.22.
exemplo6\calculaMedia.jsp	Gerar a página retornada ao usuário caso não tenha ocorrido erro de digitação, detalhado na Figura 5.23.
exemplo6\erro.jsp	Gerar a página de erro retornada ao usuário quando ocorre um erro de digitação das notas. Segue o mesmo princípio da página de erro da aplicação `exemplo5`, detalhado na Figura 5.19 (página 47).
exemplo6\WEB-INF\web.xml	Descritor de instalação da aplicação, detalhado na Figura 5.4 (página 34). Note que este arquivo é igual ao usado nas aplicaçções exemplo anteriores.

Tabela 5.8: Arquivos que formam a aplicação exemplo6.

Observe que no arquivo `calculaMedia.jsp` (Figura 5.23) a maior parte do que está escrito diz respeito ao processo de obter as notas, calcular a média e decidir se esta média deixa o estudante na situação de aprovado ou reprovado.

Toda esta lógica associada às notas e média está escrita na forma de scripts, isto é, fragmentos de código Java. Esta forma de abordar a questão tem duas características negativas. Primeiro, os fragmentos de código não são facilmente reutilizáveis pois, para tal, baseiam-se no método *recorte-e-cole*. Segundo, o projetista (*web designer*) e o programador *web developer*) devem trabalhar em conjunto, sendo que o projetista depende dos scripts do programador para projetar as páginas da aplicação.

Uma das formas de eliminar, ou pelo menos diminuir, os problemas cidatos é usar ações padrão[11], juntamente com os javabeans, para "limpar" a página.

Assim, de modo geral, sugere-se que os scriptlets sejam usados com muita

[11] No original, *standard actions*.

```
<html>
<head>
<title>Aplicação web - exemplo6</title>
</head>
<body>
<%@include file="cabecalho.html"%>
<form type=get action=calculaMedia.jsp>
<table border="1">
<tr><th>Nota 1</th><td><input type=text name=nota1></td></tr>
<tr><th>Nota 2</th><td><input type=text name=nota2></td></tr>
<tr><th>Nota 3</th><td><input type=text name=nota3></td></tr>
</table>
<br>

<INPUT TYPE=submit name=submit Value="Calcule Média">
</form>
</body>
</html>
```

Figura 5.22: Aplicação exemplo6: arquivo `index.jsp`

cautela e evitados sempre que possível.

```
  <%@page errorPage="erro.jsp"%>
<html>
<head>
<title>Aplicação web - exemplo6</title>
</head>
<body>

<%!
String param;
float nota1;
float nota2;
float nota3;
float media;
String situacao;
%>

<%-- request é um objeto implícito --%>
<%
 param = request.getParameter("nota1");
 nota1 = java.lang.Float.parseFloat(param);
 param = request.getParameter("nota2");
 nota2 = java.lang.Float.parseFloat(param);
 param = request.getParameter("nota3");
 nota3 = java.lang.Float.parseFloat(param);

 media = (nota1 + nota2 + nota3) / 3;
 if (media < 5.75f)
   situacao = "reprovado";
 else
   situacao = "aprovado";
%>

<%@include file="cabecalho.html"%>
<font size=+1>
<p>Suas três nota foram: <%= nota1 %>, <%= nota2 %>
e <%= nota3 %>.</p>
<p> Sua média, portanto, foi <%= media %>.</p>
<p> Com esta média você está <%= situacao %>.
</font>

</body>
</html>
```

Figura 5.23: Aplicação exemplo6: arquivo `calculaMedia.jsp`

5.3.3 Ações Padrão

As ações padrão permitem a execução de ações relacionadas aos aspectos dinâmicos de uma página **utilizando-se tags ao invés de scripts**. Estas ações incentivam a reusabilidade de código e facilitam o trabalho do *web designer*.

Com relação à sintaxe das ações padrão comentadas a seguir, somente é mostrada a forma mais comum e/ou simples. A relação com todas as possibilidades pode ser obtida em [Jún02].

5.3.3.1 <jsp:useBean>

Esta tag possibilita a criação de um novo objeto Java ou a utilização de um objeto previamente criado. Este objeto, que deve ser um JavaBean (apresentado no Capítulo 4), encapsula uma parte da lógica de resolução do problema tratado pela aplicação. Assim, a página que contém esta tag reforça a noção de orientação a objetos como paradigma de programação da aplicação.

A forma mais comum de uso desta tag é

```
<jsp:useBean id="objeto" class="classe" scope="escopo"/>
```

Os valores dos atributos da tag são

objeto símbolo que representa o nome do objeto.

classe classe do objeto

escopo escopo do objeto (comentado a seguir).

O scopo de um objeto pode ser `page`, `request`, `session` ou `application`. O significado de cada um dos escopos é explicado em detalhes no Capítulo 6. Um objeto com escopo `session`, por exemplo, pode representar o conjunto de produtos que um usuário está comprando. Já um objeto com escopo `application` pode representar, por exemplo, o número de visitantes da aplicação desde que ela entrou em operação[12].

Os objetos somente serão criados se não houver um objeto com o mesmo id e escopo. Se houver, a declaração referencia o objeto já existente.

Se após a **criação** do objeto for necessário inicializá-lo com valores específicos então a sintaxe a ser utilizada é a mostrada a seguir. A inicialização é feita via `scriptlets` ou via tag `<jsp:setProperty>`.

```
<jsp:useBean id="objeto" class="classe" scope="escopo">
                    inicialização
<jsp:useBean/>
```

[12] Quando o container é encerrado todos os objetos, de qualquer escopo, são destruídos. Neste caso, suas informações devem ser armazenadas em outro local, como um banco de dados.

5.3.3.2 <jsp:setProperty>

Esta tag define valores para as propriedades (atributos) de um JavaBean. As formas mais comum de utilização são

```
<jsp:setProperty name="objeto" property="*"/>

<jsp:setProperty name="objeto"
                 property="prop" param="parametro"/>

<jsp:setProperty name="objeto"
                 property="prop" value="<%= expressao%>"
```

Na primeira forma, o símbolo * como valor do atributo `property` significa que serão alterados o valor das propriedades do JavaBean `objeto` cujos nomes forem iguais ao nome dos parâmetros contidos no objeto `request`[13]. Por exemplo, se o JavaBean possui os métodos `setNome` e `setIdade` e objeto `request` possuir os parâmetros `nome` e `idade`[14] então, automaticamente, os valores dos parâmetros passarão como argumento dos métodos. A conversão dos tipos é feita automaticamente. Se não for possível converter (por exemplo, a tentativa de converter uma letra para um número) então o servidor JSP gerará uma exceção.

Na segunda forma, o atributo `prop` do JavaBean `objeto` terá seu valor modificado pelo valor do parâmetro `parametro` contido no objeto `request`.

Na terceira forma, o atributo `prop` do JavaBean `objeto` terá seu valor modificado para o valor da expressão.

5.3.3.3 <jsp:getProperty>

Esta tag permite colocar o valor de um atributo de um JavaBean na página que será enviada ao cliente. Sua sintaxe é

```
<jsp:getProperty name="objeto" property="prop"/>
```

Por exemplo, suponha que uma página JSP contenha a seguinte declaração

```
<jsp:useBean id="umAluno" class="Aluno" scope="session"/>
```

Suponha ainda que o objeto `umAluno` possui o método

```
public float getMedia()
```

que retorna a média das suas notas. Podemos colocar esta informação em uma página através da seguinte frase que combina HTML e JSP:

[13] Este objeto é descrito na seção 6.2.1, página 67.

[14] Os nomes devem ser exatamente estes.

```
...
<h1> Média Final </h1>
<p> Você tem a nota <jsp:getProperty name="umAluno"
    property="media"/> como média de suas notas.</p>
...
```

5.3.3.4 <jsp:include>

Esta tag permite a inclusão de páginas (HTML ou JSP) na página que será enviada ao cliente. Observe que ela é semelhante à diretiva `<%@ include file= ... %>` (descrita na página 43).

A sintaxe desta tag é

```
<jsp:include page="arquivo"/>
```

A principal diferença entre as duas formas de inclusão de arquivos está no momento em que ela é realizada. Com a diretiva o arquivo é incluído em tempo de compilação da página. Já com a ação o arquivo é processado em tempo de execução e só então seu resultado é incluído na página que contém a ação. Com isso, os objetos implícitos `request` e `session` estão acessíveis para a página a ser incluída.

Além dos objetos `request` e `session`, a tag permite a inclusão de novos parâmetros para serem "consumidos" pela página incluída. Neste caso a sintaxe da tag é

```
<jsp:include page="arquivo">
    <jsp:param name="nome" value="valor"/>
    ...
<jsp:include/>
```

5.3.3.5 <jsp:forward>

Esta tag permite que a requisição feita por um cliente seja repassada para uma outra página (HTML ou JSP). Por exemplo, se o usuário tentar acessar uma página da aplicação sem ter sido verificado se ele está autorizado a fazê-lo então pode-se usar a tag para desviar a execução para uma página de login.

Sua sintaxe é

```
<jsp:forward page="novaPagina"/>
```

A decisão sobre para qual página a execução será desviada pode ser determinada em tempo de execução como mostra o fragmento de código a seguir.

```
...
<%! String destino; %>
...
<jsp:forward page='<%= destino %>'/>
```

A página que que irá tratar a requisição do cliente depende, agora, do valor da variável destino. O valor desta variável pode ser determinado, por exemplo, como resultado da execução de um scriptlet.

5.3.3.6 <jsp:param>

Esta tag permite a definição de parâmetros (e seus respectivos valores) que são utilizados nas tags `<jsp:include>` e `<jsp:forward>`.

A sintaxe para atribuir um valor a um nome é

```
<jsp:param name="nome" value="valor"/>
```

5.3.3.7 Aplicação exemplo7: uso de ações padrão

A aplicação `exemplo7` tem a mesma finalidade da aplicação `exemplo6`, isto é, determinar se uma aluno está aprovado ou não em função da média de suas notas. A diferença está no uso, agora, de ações padrão.

Quando o usuário digita três notas cuja média significa aprovação para o estudante, como mostra a Figura 5.24, a aplicação retorna a página de aprovação com as notas e a média, como mostra a Figura 5.25.

Figura 5.24: exemplo7, usuário digitando notas de aprovação.

Figura 5.25: exemplo7, página indicando aprovação.

Quando o usuário digita três notas cuja média significa reprovação para o estudante, como mostra a Figura 5.26, a aplicação retorna a página de reprovação com as notas e a média, como mostra a Figura 5.27.

Os arquivos que formam a aplicação `exemplo7` estão indicados na Tabela 5.9.

Figura 5.26: exemplo7, usuário digitando notas de reprovação.

Figura 5.27: exemplo7, página indicando reprovação.

Arquivo	Finalidade
exemplo7\cabecalho.html	Gerar o cabeçalho das páginas da aplicação. Segue o mesmo princípio do cabeçalho da aplicação `exemplo5`, detalhado na Figura 5.16 (página 45).
exemplo7\index.jsp	Gerar a página inicial da aplicação onde o usuário digita as três notas. Este arquivo é idêntico ao usado na aplicação `exemplo6`, detalhado na Figura 5.22 (página 51).
exemplo7\calculaMedia.jsp	Controlar a aplicação (veja discussão a seguir). O arquivo é detalhado na Figura 5.28.
exemplo7\erro.jsp	Gerar a página de erro retornada ao usuário quando ocorre um erro de digitação das notas. Segue o mesmo princípio da página de erro da aplicação `exemplo5`, detalhado na Figura 5.19 (página 47).
exemplo7\WEB-INF\web.xml	Descritor de instalação da aplicação, detalhado na Figura 5.4 (página 34). Este arquivo é igual ao usado nas aplicações exemplo anteriores.
exemplo7\aprovado.jsp	Gerar a página retornada ao usuário caso o aluno seja aprovado, detalhado na Figura 5.29.
exemplo7\reprovado.jsp	Gerar a página retornada ao usuário caso o aluno seja reprovado, detalhado na Figura 5.30.
exemplo7\mostraNotas.jsp	Gerar uma tabela HTML com as três notas, detalhado na Figura 5.31.
exemplo7\classes\ljk\ AlunoBean.class	JavaBean responsável por guardar as notas do estudante e calcular a sua média, detalhado na Figura 5.32.

Tabela 5.9: Arquivos que formam a aplicação exemplo7.

```
<%@page errorPage="erro.jsp"%>
<jsp:useBean id="umAluno" class="ljk.AlunoBean" scope="request"/>
<jsp:setProperty name="umAluno" property="*"/>

<%! String situacao,cor;%>
<%
  if (umAluno.getMedia() >= 5.75f) {
     situacao = "aprovado.jsp";
     cor = "#99ff99";  // verde
  }
  else {
     situacao = "reprovado.jsp";
     cor = "#ff6666";  // vermelho
  }
%>

<jsp:forward page='<%=situacao%>'>
  <jsp:param name="cor" value='<%=cor%>'/>
</jsp:forward>
```

Figura 5.28: Aplicação exemplo7: arquivo calculaMedia.jsp

O arquivo `calculaMedia.jsp`, Figura 5.28, assume, com o uso das ações padrão, um papel radicalmente diferente daquele assumido na aplicação `exemplo6`. No lugar de retornar uma página para o usuário, sua nova função é **controlar a lógica da resolução do problema**.

A tag `<jsp:useBean>` cria uma instância do JavaBean `ljk.AlunoBean`. Em seguida, a tag `<jsp:setProperty>` **automaticamente** passa os valores das três notas definidas pelo usuário no formulário da página `index.jsp` para a instância do JavaBean. Isto é possível porque no formulário os campos relativos às três notas têm nomes compatíveis com os métodos do JavaBean. Por exemplo, o valor do campo `nota1` do formulário é passado como parâmetro do método `public void setNota1(float valor)`. A seguir o valor da variável `situação` é definido em função da média obtida com o JavaBean. Por fim, a tag `<jsp:forward>` encaminha a requisição do cliente para a página indicada vela variável `situação` juntamente com mais um parâmetro chamado `cor`.

O arquivo `aprovado.jsp`, Figura 5.29, define a página que será enviada ao cliente quando o aluno for aprovado. Observe o uso das tags `<jsp:include>` e `<jsp:getProperty>`. A primeira inclui o resultado do processamento do arquivo `mostraNotas.jsp` (descrito a seguir). A segunda recupera o valor da média do objeto `umAluno`. Note que as duas expressões a seguir são

```
<jsp:useBean id="umAluno" class="ljk.AlunoBean" scope="request"/>

<html>
<head>
<title>Aplicação web - exemplo7</title>
</head>
<body>

<%@include file="cabecalho.html"%>
<h2> Aluno Aprovado </h2>
<font size=+1>
<jsp:include page="mostraNotas.jsp"/>

<p> Sua média, portanto, foi
    <jsp:getProperty name="umAluno" property="media"/>.</p>
</font>
</body>
</html>
```

Figura 5.29: Aplicação exemplo7: arquivo aprovado.jsp

equivalentes.

```
<jsp:getProperty name="umAluno" property="media"/>

<%= umAluno.getMedia()%>
```

O arquivo `reprovado.jsp`, Figura 5.30, define a página que será enviada ao cliente quando o aluno for reprovado. Ele segue a mesma lógica do arquivo `aprovado.jsp` descrito na Figura 5.29.

O arquivo `mostraNotas.jsp`, Figura 5.31, mostra um exemplo de interação entre HTML e JSP. Observe que o texto é formado por tags HTML e inserções de expressões e tags JSP. Note ainda que não há, neste arquivo, nenhuma indicação de onde o seu conteúdo será utilizado. A cor de fundo da tag `<th>` é definida pelo parâmetro `cor` que foi adicionado ao objeto implícito `request`.

```
<jsp:useBean id="umAluno" class="ljk.AlunoBean" scope="request"/>

<html>
<head>
<title>Aplicação web - exemplo7</title>
</head>
<body>

<%@include file="cabecalho.html"%>
<h2> Aluno Reprovado </h2>
<font size=+1>
<jsp:include page="mostraNotas.jsp"/>

<p> Sua média, portanto, foi
    <jsp:getProperty name="umAluno" property="media"/>.</p>
</font>
</body>
</html>
```

Figura 5.30: Aplicação exemplo7: arquivo reprovado.jsp

```
<jsp:useBean id="umAluno" class="ljk.AlunoBean" scope="request"/>

<h3> Suas Notas </h3>

<table border=1 >
  <%
   int i = 1;
   String nota;
   for (i = 1; i <= 3; i++) {
     nota = "nota" + i;
  %>
     <tr>
       <th bgcolor=<%= request.getParameter("cor")%>> Nota <%= i %>
       </th>
       <td><%= umAluno.getNota(i)%>
       </td>
     </tr>
   <%
   }
   %>
</table>
```

Figura 5.31: Aplicação exemplo7: arquivo mostraNotas.jsp

```
package ljk;

public class AlunoBean {
  private float[] notas;
  private float media;

  public AlunoBean() {
    notas = new float [3];
    notas[0] = 0.0f;
    notas[1] = 0.0f;
    notas[2] = 0.0f;
    media = 0.0f;
  }

  protected void calculeMedia() {
    media = (notas[0] + notas[1] + notas[2])/3;
  }

  public void setNota1(float valor) {
    notas[0] = valor;
    this.calculeMedia();
  }

  public void setNota2(float valor) {
    notas[1] = valor;
    this.calculeMedia();
  }
  public void setNota3(float valor) {
    notas[2] = valor;
    this.calculeMedia();
  }

  public float getNota(int i) {
    return notas[i-1];
  }

  public float getMedia() {
    return media;
  }
}
```

Figura 5.32: Aplicação exemplo7: arquivo AlunoBean.java

Capítulo 6

JSP : Uso de Objetos

O protocolo HTTP foi desenvolvido para que qualquer pessoa, de uma forma muito simples, possa recuperar arquivos, geralmente no formato HTML, localizados em computadores distantes conectados à internet. Simplificando as coisas, ele poderia ser descrito pelo seguinte algoritmo:

1. O programa cliente solicita, via protocolo de rede TCP/IP, ao programa servidor um arquivo identificado pela string `http://...`

2. O programa servidor, ao receber a solicitação do programa cliente, analisa o string, carrega o arquivo correspontente na memória e o envia para o cliente.

Este cenário, comum nos primeiros anos[1] da web, foi aos poucos sendo modificado. Logo percebeu-se que seria possível e interessamte não só buscar informações em computadores distantes mas também enviar informações para estes mesmos computadores.

Surge então, neste contexto, novos conceitos e, consequentemente, novas necessidades para os quais o protocolo HTTP não foi originalmente concebido. Por exemplo, uma **sessão** é um período de tempo delimitado – que pode durar segundos, minutos ou mesmo horas – em que os programas cliente e servidor ficam trocando informações que só fazem sentido para aquela sessão.

O exemplo típico onde o conceito de sessão aparece é quando se acessa um site para realizar uma compra de algum produto. A compra é um processo relativamente demorado onde os itens adquiridos vão sendo colocados em um "carrinho virtual". Terminada a compra, isto é, encerrada a sessão, o carrinho deve ser dispensado ("esvaziado").

Outro exemplo é o problema da autenticação do usuário. O programa servidor, por exemplo, só autoriza o cliente a acessar determinadas páginas

[1] O protocolo HTTP surgiu em um laboratório de pesquisa na Suíça no início dos anos 1990.

ou realizar determinadas atividades se ele estiver perfeitamente identificado e autorizado (tipicamente fornecendo algum código de identificação acompanhado de uma senha).

O problema central nos dois exemplos apresentados é que o protocolo HTTP não foi feito para guardar informações, mas sim simplesmente receber/enviar informações (o conteúdo de um arquivo).

A tecnologia JSP disponibiliza aos desenvolvedores de aplicações para web a possibilidade de criar e/ou acessar objetos armazenados no programa servidor. Os objetos podem ser criados especificamente para uma aplicação. A especificação JSP define também um conjunto de objetos pré-definidos (os chamados **objetos implícitos**). Com estes objetos é possível sanar as dificuldades inerentes ao protocolo HTTP.

O objetivo deste capítulo é apresentar estes objetos implícitos e utilizá-los em uma aplicação exemplo que usa, ainda, objetos específicos.

6.1 Escopo dos Objetos

Os objetos, tanto os criados explicitamente em uma página JSP como os disponibilizados pelo próprio servidor (objetos implícitos), possuem diferentes graus de visibilidade. Esta visibilidade, conhecida como escopo, define em que contextos um objeto está disponível para ser usado.

Uma forma intuitiva de compreender a noção de escopo é perceber que em um site de compras, por exemplo, os itens que estão sendo comprados por um usuário em um computador não devem ser vistas por outros usuários. Da mesma forma, a informação sobre quantos e quais objetos estão disponíveis para venda deve estar disponível para todos os usuários.

Existem quatro escopos possíveis, descritos a seguir.

6.1.1 page

Objetos com escopo **page** (página) somente estão acessíveis na página que os criou. Objetos com este escopo possuem referências no objeto implícito `pageContext`. Este é o caso dos objetos criados através de scriptlets e ações padrão com atributo `<jsp:useBean ... scope="page"/>`.

6.1.2 request

Objetos com escopo **request** (requisição) somente estão acessíveis nas páginas processando a mesma requisição onde foram criados. Objetos com este escopo possuem referências no objeto implícito `request`. Este é o caso dos objetos criados através de scriptlets e incluídos em `request` e ações padrão com atributo `<jsp:useBean ... scope="request"/>`.

6.1.3 session

Objetos com escopo **session** (sessão) somente estão acessíveis nas páginas processando requisições que estão na mesma sessão[2]. Objetos com este escopo possuem referências no objeto implícito `session`. Este é o caso dos objetos criados através de scriptlets e incluídos em `session` e ações padrão com atributo `<jsp:useBean ... scope="session"/>`.

6.1.4 application

Objetos com escopo **application** (aplicação) estão acessíveis enquanto a aplicação estiver carregada no servidor JSP e este estiver rodando. Objetos com este escopo possuem referências no objeto implícito `application`. Este é o caso dos objetos criados através de scriptlets e incluídos em `application` e ações padrão com atributo `<jsp:useBean ... scope="application"/>`.

Todos os usuários têm acesso aos mesmos objetos com escopo de aplicação. Isto pode causar problemas de consistência quando dois ou mais usuários tentam ler ou modificar o mesmo objeto. Para isso é reponsabilidade do programador garantir o acesso exclusivo através dos mecanismos disponibilizados pela linguagem java (palavra reservada `synchronized`).

6.2 Objetos Implícitos

No desenvolvimento de páginas JSP estão disponíveis uma série de objetos que podem ser usados diretamente, isto é, sem a necessidade de serem criados explicitamente. Estes objetos são criados automaticamente pelo servidor JSP. As seções que seguem apresentam os principais (mais comumente usados) objetos implícitos.

6.2.1 Objeto `request`

O objeto `request` pertence à classe

```
javax.servlet.http.HttpServletRequest
```

A principal finalidade do objeto `request` é armazenar dados relativos à solicitação feita pelo cliente. Alguns de seus métodos são descritos na Tabela 6.1.

6.2.2 Objeto `response`

O objeto `response` pertence à classe

[2]No servidor JSP Tomcat, uma sessão tem duração de 30 minutos como valor default. Este tempo pode ser ajustado para qualquer outro valor.

Método	Descrição
`String getParameter(String nome)`	Retorna o valor do parâmetro `nome` ou `null` caso o parâmetro não esteja definido.
`void setAtribute(String nome, Object valor)`	Inclui o objeto `valor` associando a ele o nome `nome`.
`Object getAtribute(String nome)`	Retorna o objeto associado ao nome `nome`.
`String getRemoteHost()`	Retorna o nome do host do cliente que solicitou a página.

Tabela 6.1: Alguns métodos do objeto request

`javax.servlet.http.HttpServletResponse`

A principal finalidade do objeto `response` é representar a página que será enviada ao cliente. Alguns de seus métodos são descritos na Tabela 6.2.

Método	Descrição
`<%= expressao =%>)`	O valor da `expressão`, convertido para string, é inserido na página de resposta.
`void addCookie(Cookie umCookie)`	Armazena `umCookie` no browser.
`void sendRedirect(String URL)`	Retorna o comanddo `redirect` para o browser. Este imediatamente solicita o endereço `URL` para o servidor.

Tabela 6.2: Alguns métodos do objeto response

6.2.3 Objeto `session`

O objeto `session` pertence à classe

`javax.servlet.http.HttpSession`

O objeto `session` é usado para armazenar objetos cuja existência está limitada à duração da sessão. Alguns dos métodos deste objeto estão descritos na Tabela 6.3.

Método	Descrição
`void setAtribute(String nome, Object valor)`	Inclui o objeto `valor` associando a ele o nome `nome`.
`Object getAtribute(String nome)`	Retorna o objeto associado ao nome `nome`.
`void removeAtribute(String nome)`	Remove o objeto associado ao nome `nome`.
`void invalidate()`	Encerra a sessão presente e apaga todos os objetos armazenados como atributos.
`void setMaxInactiveInterval(int segundos)`	Determina por quantos segundos uma sessão é mantida válida sem que haja nova requisição do cliente.
`int getMaxInactiveInterval()`	Retorna quantos segundos uma sessão é mantida válida sem que haja nova requisição do cliente.

Tabela 6.3: Alguns métodos do objeto session

6.2.4 Objeto `application`

O objeto `application` pertence à classe

```
javax.servlet.ServletContext
```

O objeto `application` é usado para armazenar objetos cuja existência está vinculada à duração da própria aplicação. Alguns dos métodos deste objeto estão descritos na Tabela 6.4.

Método	Descrição
`void setAtribute(String nome, Object valor)`	Inclui o objeto `valor` associando a ele o nome `nome`.
`Object getAtribute(String nome)`	Retorna o objeto associado ao nome `nome`.
`void removeAtribute(String nome)`	Remove o objeto associado ao nome `nome`.

Tabela 6.4: Alguns métodos do objeto application

6.3 Exemplo: Gerenciamento de Sessão

O uso de objetos, tanto os implícitos quanto os criados pela própria aplicação, permitem que a consistência da aplicação seja mantida. Como exemplo, considere a aplicação "Jogo das Bandeiras" descrito a seguir. Esta aplicação faz uso intessivo dos dois tipos de objetos (implícitos e criados).

6.3.1 Descrição da Aplicação Jogo das Bandeiras

O objetivo da aplicação é adivinhar o nome do país das bandeiras que são mostradas na tela. As funcionalidades da aplicação são:

- O usuário só acessa a aplicação mediante senha[3].
- A aplicação mantém um registro sobre quantas vezes o usuário usou a aplicação[4].
- Em cada sessão de uso da aplicação, são registradas o número de respostas e o número de respostas corretas.
- O usuário pode jogar indefinidamente. Ele encerra sua sessão clicando no botão apropriado.
- As bandeiras são mostradas aleatoriamente a partir de um conjunto previamente definido[5].
- O uso irregular da aplicação implica no imediato encerramento da sessão. Uso irregular, neste contexto, significa uma tentativa de acessar uma página fornecendo o endereço manualmente no browser diferente dos definidos para a aplicação.

6.3.2 Interface da Aplicação Jogo das Bandeiras

A página de acesso à aplicação `flags` é mostrada na Figura 6.1. O usuário deve fornecer um nome de usuário e senha válidos.

Se o usuário digitar um nome e/ou senha inválidos então o servidor responderá com a página mostrada na Figura 6.2.

Após entrar com nome e senha válidos, o jogo começa e o usuário deve responder o nome do país indicado na página mostrada na Figura 6.3. Observe que página indica o nome do usuário assim como seus dados (número

[3]Por uma questão de simplicidade, os nomes dos usuários e suas respectivas senhas são armazenadas em uma classe Java. Numa situação real, estas informações provavelmente estariam armazenadas em um banco de dados.

[4]Como os dados dos usuários são mantidos em memória, sempre que o servidor JSP é encerrado esta informação é perdida.

[5]Como no caso dos usuários, as informações sobre as bandeiras estão armazenadas em memória (em uma classe Java).

Figura 6.1: Página de login da aplicação flags.

Figura 6.2: Página indicando login inválido na aplicação flags.

de partidas já disputadas, número de tentativas nesta sessão e número de acertos nesta sessão).

6.3.3 Modelagem da Aplicação flags

Os arquivos que utilizam a tenologia JSP (aqueles com extensão `.jsp`) estão descritos na Tabela 6.5. Os arquivos HTML (aqueles com extensão `.html`) estão descritos na Tabela 6.6. Finalmente, os JavaBeans (arquivos Java) usados na aplicação `flags` são descritos na Tabela 6.7. Observe que nesta tabela consta também o arquivo `servlet.jar` relativo aos objetos implícitos.

Figura 6.3: Página principal do jogo na aplicação flags.

Arquivo	Finalidade
flags\index.jsp	Gerar a página de login da aplicação, descrito na Figura 6.4.
flags\loginInvalido.jsp	Gerar a página indicando que o login foi inválido, descrito na Figura 6.5.
flags\paginaInicial.jsp	Gerar a página principal do jogo onde é mostrada uma bandeira para o usuário adivinhar o nome do país correspondente, descrito na Figura 6.6.
flags\autenticador.jsp	Autenticar o usuário. Ele impede que a página gerada pelo arquivo `paginaInicial.jsp` seja enviada para o cliente se o usuário não se identificou inicialmente, descrito na Figura 6.7.
flags\controlador.jsp	Controlar o fluxo de execução da aplicação, descrito na Figura 6.8.

Tabela 6.5: Arquivos `.jsp` da aplicação flags

6.3.3.1 Arquivo `index.jsp`

O arquivo `index.jsp`, mostrado na Figura 6.4, gera a página de login da aplicação `flags`.

```
<html>
<head>
<%@include file="titulo.html"%>
</head>
<body>
<%@include file="cabecalho.html"%>

<h2> Identifique-se</h2>
<form type=post action=controlador.jsp>
  <%@include file="menuPrincipal.html"%><br>
  <input type="hidden" name="opcao" value="login">
  <input type=submit value="Entre">
</form>
</body>
</html>
```

Figura 6.4: Aplicação flags: arquivo index.jsp.

6.3.3.2 Arquivo `loginInvalido.jsp`

O arquivo `loginInvalido`, mostrado na Figura 6.5, gera a página que indica que o usuário digitou o nome e/ou a senha errado(s). .

6.3.3.3 Arquivo `paginaInicial`

O arquivo `paginaInicial.jsp`, mostrado na Figura 6.6, gera a página onde o jogo acontece.

A primeira linha deste arquivo, que contém a tag JSP `<jsp:include>`, é responsável, e somente ela, pela autenticação do usuário. Basta, portanto, incluir esta linha em qualquer arquivo que precise ser protegida contra seu uso indevido.

6.3.3.4 Arquivo `autenticador.jsp`

O arquivo `autenticador.jsp`, mostrado na Figura 6.7, é usado para impedir o uso indevido de alguma página.

```
<html>
<head>
<%@ include file="titulo.html"%>
</head>
<body>
<%@include file="cabecalho.html"%>

<h2> Login Inválido</h2>
<h3><a href="index.jsp">Tente Novamente</a></h3>
</body>
</html>
```

Figura 6.5: Aplicação flags: arquivo loginInvalido.jsp.

O autenticador funciona da seguinte maneira. Caso na sessão atual não esteja definido um objeto com o nome `jogAtual` ou na requisição do cliente não esteja definido o parâmetro `opcao` então está caracterizado o uso indevido da página que inclui o arquivo `autenticador.jsp`. Inclui-se, então, o atributo `acessoIlegal` na requisição do cliente e desvia-se o fluxo de execução para a página responsável pelo controle da lógica da aplicação (`controlador.jsp`).

6.3.3.5 Arquivo `controlador.jsp`

O arquivo `controlador.jsp`, mostrado na Figura 6.8, exerce o papel de controlador da aplicação, seguindo assim o Modelo 2 (descrito na seção 3.2, página 22).

O controlador funciona da seguinte maneira. Com exceção do arquivo `index.jsp`, todas as requisições do cliente são encaminhadas para a página `controlador.jsp`. Esta página, encaminha a requisição (e o objeto sessão) para o javabean `oCtrl`. É este objeto quem efetivamente analisa a requisição, executa o(s) algoritmo(s) pertinente(s) e determina qual página deverá ser retornada para o cliente.

O objeto `oCtrl` pode ainda indicar que a requisição do cliente precisa ser redirecionada para outra página. Isto acontece quando o usuário tenta acessar uma página da aplicação ilegalmente.

```
<jsp:include page="autenticador.jsp"/>
<jsp:useBean id="jogAtual" class="ljk.flags.sess.JogadorAtualBean"
                           scope="session"/>
<jsp:useBean id="oJogo" class="ljk.flags.app.JogoDasBandeirasBean"
                           scope="application"/>

<% session.setAttribute("bandAtual", oJogo.sorteieBandeira());%>

<html>
<head>
<%@ include file="titulo.html"%>
</head>
<body>
<%@include file="cabecalho.html"%>

<h2> Bem vindo ao Jogo <%= jogAtual.getOJogador().getNome()%></h2>

<table border="1">
  <tr bgcolor="yellow">
     <th>Partidas</th><th>Acertos</th><th>Tentativas</th></tr>
  <tr><td><%= jogAtual.getOJogador().getNumPartidas()%></td>
      <td><%= jogAtual.getNumAcertos()%></td>
      <td align="center"><%= jogAtual.getNumTentativas()%></td>
  </tr>
</table>

<h3> Objetivo do Jogo </h3>
<p>O objetivo do jogo é acertar o nome do país que possui a
bandeira indicada abaixo. </p>

<img src="./flags/<jsp:getProperty name="bandAtual"
                       property="img"/>" border="2">
<form type=post action controlador.jsp>
  <input type=text name=pais size=20>
  <input type=hidden name=opcao value=verificar>
  <input type=submit value="Verificar Resposta">
</form>
<form type=post action controlador.jsp>
   <input type="hidden" name="opcao" value="sair">
   <input type="submit" value="Sair do Jogo">
</form>
</body>
</html>
```

Figura 6.6: Aplicação flags: arquivo paginaInicial.jsp.

```
<% if ((session.getAttribute("jogAtual") == null) ||
       (request.getParameter("opcao") == null)) {
     request.setAttribute("acessoIlegal","sim");%>
     <jsp:forward page="controlador.jsp"/>
   <%}
%>
```

Figura 6.7: Aplicação flags: arquivo autenticador.jsp.

```
<jsp:useBean id="oJogo" class="ljk.flags.app.JogoDasBandeirasBean"
                        scope="application"/>
<jsp:useBean id="oCtrl" class="ljk.flags.ctrl.ControladorBean"
                        scope="session">
  <% oCtrl.conhecaJogo(oJogo); %>
</jsp:useBean>

<% oCtrl.processeOpcao(request,session);
 if (oCtrl.getRedirecionar()) {
      response.sendRedirect("http://"
                            + request.getServerName()
                            + ":" + request.getServerPort()
                            + "/flags/"
                            + oCtrl.getProxPagina());
   }
   else {%>
     <jsp:forward page='<%= oCtrl.getProxPagina()%>'/>
   <%}
%>
```

Figura 6.8: Aplicação flags: arquivo controlador.jsp.

Arquivo	Finalidade
flags\cabecalho.html	Arquivo que contém o cabeçalho – na forma de fragmento de código HTML – das páginas da aplicação. Este arquivo segue o princípio dos cabeçalhos dos exemplos mostrados no Capítulo 5.
flags\titulo.html	Arquivo, não mostrado nesta apostila, que contém apenas a tag HTML `<title>`. Serve para mostrar que as diretivas JSP podem aparecer também no preâmbulo de uma página HTML.
flags\menuPrincipal.html	Arquivo que contém os campos do formulário usado no login da aplicação, descrito na Figura 6.9.
flags\flags*.gif	O conjunto de arquivos de imagem das bandeiras.

Tabela 6.6: Arquivos `.html` da aplicação flags

6.3.3.6 Arquivo `menuPrincipal.html`

O arquivo `menuPrincipal.html`, mostrado na Figura 6.9, define os campos e o formato do formulário usado para login da aplicação.

```
<table border="1">
  <tr bgcolor="yellow"><th>Nome</th><th>Senha</th></tr>
  <tr><td><input type="text" name="nome" size="20"></td>
      <td><input type="password" name="senha" size="10"></td>
  </tr>
</table>
```

Figura 6.9: Aplicação flags: arquivo menuPrincipal.html.

6.3.3.7 Arquivo `BandeiraBean.java`

O arquivo `BandeiraBean.java`, mostrado na Figura 6.10, representa uma bandeira na aplicação `flags`.

Arquivo	Finalidade
flags\...\BandeiraBean.class	Representar uma bandeira, descrito na Figura 6.10.
flags\...\JogadorBean.class	Representar um jogador cadastrado na aplicação, descrito na Figura 6.11.
flags\...\BandeirasBean.class	Representar o conjunto de bandeiras cadastradas na aplicação, descrito na Figura 6.12.
flags\...\JogadoresBean.class	Representar o conjunto de jogadores cadastrados na aplicação, descrito na Figura 6.13.
flags\...\JogoDasBandeirasBean.class	Representar o jogo das bandeiras, descrito na Figura 6.14.
flags\...\JogadorAtualBean.class	Representar o jogador que está efetivamente jogando, descrito na Figura 6.15.
flags\...\ControladorBean.class	Auxiliar o arquivo `controlador.jsp` no gerenciamento da lógia da aplicação, descrito nas Figuras 6.16 e 6.17.
flags\WEB-INF\lib\servlet.jar	Conter a implementação das classes dos objetos implícitos. Este arquivo é necessário pois os javabeans se comunicam com estes objetos.

Tabela 6.7: Arquivos `.java` da aplicação flags

6.3.3.8 Arquivo `JogadorBean.java`

O arquivo `JogadorBean.java`, mostrado na Figura 6.11, representa um jogador cadastrado na aplicação `flags`.

6.3.3.9 Arquivo `BandeirasBean.java`

O arquivo `BandeirasBean.java`, mostrado na Figura 6.12, representa o conjunto de bandeiras cadastradas na aplicação `flags`.

6.3.3.10 Arquivo `JogadoresBean.java`

O arquivo `JogadoresBean.java`, mostrado na Figura 6.13, representa o conjunto de jogadores cadastrados na aplicação `flags`.

```
package ljk.flags.app;

public class BandeiraBean {
  private String pais;
  private String img;

  public BandeiraBean(String pais, String img) {
    this.pais = pais;
    this.img = img;
  }

  public String getPais() {
    return pais;
  }

  public String getImg() {
    return img;
  }
}
```

Figura 6.10: Aplicação flags: arquivo BandeiraBean.java.

6.3.3.11 Arquivo `JogoDasBandeirasBean.java`

O arquivo `JogoDasBandeirasBean.java`, mostrado na Figura 6.14, representa o jogo das bandeiras na aplicação `flags`.

6.3.3.12 Arquivo `JogadorAtualBean.java`

O arquivo `JogadorAtualBean.java`, mostrado na Figura 6.15, representa o o jogador que está acessando a aplicação `flags`. Observe na Figura 6.6 (página 76) que a tag `<jsp:useBean>` define uma instância desta classe com escopo de sessão (`... scope="session"...`). Este escopo, portanto, permite que vários usuários acessem simultaneamente a aplicação[6].

6.3.3.13 Arquivo `ControladorBean.java`

O arquivo `ControladorBean.java`, mostrado nas Figuras 6.16 e 6.17, representa um objeto cuja finalidade é executar a lógica do jogo das bandeiras na aplicação `flags`.

[6] Por uma questão de simplificação, observe que na versão atual da aplicação nada impede que dois usuários diferentes se loguem com o mesmo nome de usuário.

```
package ljk.flags.app;

public class JogadorBean {
  private String nome;
  private String senha;
  private int numPartidas;

  public JogadorBean(String nome, String senha) {
    this.nome = nome;
    this.senha = senha;
    numPartidas = 0;
  }

  public String getNome() {
    return nome;
  }

  public boolean ehSenha(String valor) {
    return valor.equals(senha);
  }

  public void incrNumPartidas() {
    numPartidas++;
  }

  public int getNumPartidas() {
    return numPartidas;
  }
}
```

Figura 6.11: Aplicação flags: arquivo JogadorBean.java.

```
package ljk.flags.app;

public class BandeirasBean {
  private BandeiraBean[] asBandeiras;
  private int numBandeiras;

  public BandeirasBean() {
    numBandeiras = 13;
    asBandeiras = new BandeiraBean [numBandeiras];
    asBandeiras[0] = new BandeiraBean("Argentina", "ar.gif");
    asBandeiras[1] = new BandeiraBean("Austrália", "au.gif");
    asBandeiras[2] = new BandeiraBean("Brasil",    "br.gif");
    asBandeiras[3] = new BandeiraBean("Canadá",    "ca.gif");
    asBandeiras[4] = new BandeiraBean("Chile",     "cl.gif");
    asBandeiras[5] = new BandeiraBean("China",     "cn.gif");
    asBandeiras[6] = new BandeiraBean("Cuba",      "cu.gif");
    asBandeiras[7] = new BandeiraBean("Dinamarca", "dk.gif");
    asBandeiras[8] = new BandeiraBean("Finlândia", "fi.gif");
    asBandeiras[9] = new BandeiraBean("Japão",     "jp.gif");
    asBandeiras[10] = new BandeiraBean("Portugal",  "pt.gif");
    asBandeiras[11] = new BandeiraBean("Rússia",    "ru.gif");
    asBandeiras[12] = new BandeiraBean("Uruguai",   "uy.gif");
  }

  public int getNumBandeiras() {
    return numBandeiras;
  }

  public BandeiraBean getAsBandeiras(int indice) {
    return asBandeiras[indice];
  }
}
```

Figura 6.12: Aplicação flags: arquivo BandeirasBean.java.

```
package ljk.flags.app;

public class JogadoresBean {
  private JogadorBean[] osJogadores;
  private int numJogadores;

  public JogadoresBean() {
    numJogadores = 3;
    osJogadores = new JogadorBean [numJogadores];
    osJogadores[0] = new JogadorBean("Fulano da Silva", "fuva");
    osJogadores[1] = new JogadorBean("Beltrano Santos", "beos");
    osJogadores[2] = new JogadorBean("Cicrano Souza",   "ciza");
  }

  public JogadorBean valideSenha(String nome, String senha) {
    JogadorBean oJogadorAtual = null;;
    int i = 0;
    while (i < numJogadores) {
      if ((nome.equals(osJogadores[i].getNome())) &&
          osJogadores[i].ehSenha(senha)) {
        oJogadorAtual = osJogadores[i];
        oJogadorAtual.incrNumPartidas();
        i = numJogadores;
      }
      i++;
    }
    return oJogadorAtual;
  }
}
```

Figura 6.13: Aplicação flags: arquivo JogadoresBean.java.

```
package ljk.flags.app;

import java.util.Random;

public class JogoDasBandeirasBean {
  private BandeirasBean asBandeiras;
  private JogadoresBean osJogadores;
  private int numBandeiras;
  private Random rand;

  public JogoDasBandeirasBean() {
    asBandeiras = new BandeirasBean();
    numBandeiras = asBandeiras.getNumBandeiras();
    osJogadores = new JogadoresBean();
    rand = new Random();
  }

  public JogadorBean valideLogin(String nome, String senha) {
    return osJogadores.valideSenha(nome, senha);
  }

  public BandeiraBean sorteieBandeira() {
    return asBandeiras.getAsBandeiras(rand.nextInt(numBandeiras));
  }
}
```

Figura 6.14: Aplicação flags: arquivo JogoDasBandeirasBean.java.

```
package ljk.flags.sess;

import ljk.flags.app.JogadorBean;

public class JogadorAtualBean {
  private JogadorBean oJogador;
  private int numAcertos;
  private int numTentativas;

  public JogadorAtualBean(JogadorBean umJogador) {
    oJogador = umJogador;
  }

  public JogadorBean getOJogador() {
    return oJogador;
  }

  public int getNumAcertos() {
    return numAcertos;
  }

  public int getNumTentativas() {
    return numTentativas;
  }

  public void incrNumTentativas() {
    numTentativas++;
  }

  public void incrNumAcertos() {
    numAcertos++;
  }
}
```

Figura 6.15: Aplicação flags: arquivo JogadorAtualBean.java.

```
package ljk.flags.ctrl;

import ljk.flags.app.*;
import ljk.flags.sess.JogadorAtualBean;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class ControladorBean {
  private JogoDasBandeirasBean oJogo;
  private HttpServletRequest request;
  private HttpSession session;
  private String proxPagina;
  private boolean redirecionar;

  public ControladorBean() {
    proxPagina = null;
    redirecionar = false;
  }

  public void conhecaJogo(JogoDasBandeirasBean oJogo) {
    this.oJogo = oJogo;
  }

  public void processeOpcao(HttpServletRequest req, HttpSession sess) {
    request = req;
    session = sess;
    redirecionar = false;
    if (request.getAttribute("acessoIlegal") != null)
         processeAcessoIlegal();
    else {
      String opcao = req.getParameter("opcao");
      if (opcao == null) {
          proxPagina = "index.jsp";
          redirecionar = true;
      }
      else
        if (request.getAttribute("acessoIlegal") != null)
           processeAcessoIlegal();
        else if (opcao.equals("login"))
          processeLogin();
        else if (opcao.equals("sair"))
          processeSair();
        else if (opcao.equals("verificar"))
          processeVerificar();
        else {
          proxPagina = "index.jsp";
          redirecionar = true;
        }
        }
  }
   ...
```

Figura 6.16: Aplicação flags: arquivo ControladorBean.java (a).

```
   ...
  protected void processeLogin() {
     JogadorBean oJogadorAtual;
     session.removeAttribute("jogAtual");
     proxPagina ="loginInvalido.jsp";
     String nome = request.getParameter("nome");
     String senha = request.getParameter("senha");
     if ((nome != null) && (senha != null)) {
       oJogadorAtual = oJogo.valideLogin(nome,senha);
       if (oJogadorAtual != null) {
         session.setAttribute("jogAtual", new JogadorAtualBean(oJogadorAtual));
         proxPagina = "paginaInicial.jsp";
       }
     }
   }

   protected void processeSair() {
      session.invalidate();
      redirecionar = true;
      proxPagina = "index.jsp";
   }

   protected void processeVerificar() {
     JogadorAtualBean oJogAtual = (JogadorAtualBean) session.getAttribute("jogAtual");
     String pais = request.getParameter("pais");
     if ((oJogAtual != null) && (pais != null)) {
       BandeiraBean bandAtual = (BandeiraBean) session.getAttribute("bandAtual");
       if (pais.equals(bandAtual.getPais()))
         oJogAtual.incrNumAcertos();
       oJogAtual.incrNumTentativas();
       proxPagina = "paginaInicial.jsp";
     }
     else {
         session.removeAttribute("jogAtual");
         proxPagina = "index.jsp";
         redirecionar = true;
     }
   }

   protected void processeAcessoIlegal() {
     proxPagina = "balalaco.jsp"; redirecionar = true;
   }

   public boolean getRedirecionar() {
     return redirecionar;
   }

   public String getProxPagina() {
     return proxPagina;
   }
}
```

Figura 6.17: Aplicação flags: arquivo ControladorBean.java (b).

Capítulo 7

JDBC

Nas aplicações para web é comum o uso de banco de dados. Frequentemente o banco tem, inclusive, existência anterior à aplicação. Do ponto de vista conceitual, o banco de dados é definico como uma das camadas nas arquiteturas cliente-servidor de N camadas.

O objetivo deste capítulo é mostrar, através de um exemplo, como integrar as tecnologias de **JavaBeans** (apresentada no Capítulo 4) e de **JSP** (apresentada no Capítulo 5) com a tecnologia de **JDBC** (Java Data Base Connectors). Uma visão rápida sobre JDBC pode ser obtida no artigo de Duane [Fie00].

7.1 Descrição da Aplicação Exemplo

A aplicação, batizada de **copa2002**, tem como objetivo administrar os dados relativos às partidas de futebol realizadas durante a Copa do Mundo de Futebol de 2002.

De cada partida são registradas as seguintes informações:

- número do jogo. Por exemplo, o primeiro jogo tem número 1, o segundo número 2 e assim por diante.
- data do jogo.
- nome dos países envolvidos na partida.
- placar do jogo.

O usuário da aplicação pode realizar as seguintes atividades:

- incluir os dados de um jogo.
- obter os dados dos jogos de um determinado país.
- obter os dados de todos os jogos.

7.2 Modelagem da Base de Dados

A partir da descrição do problema, é necessário criar um modelo computacional relativo à base de dados propriamente dita.

A base de dados `xxe`[1] armazena os dados na tabela `copa2002` definida conforme mostra a Tabela 7.1.

Campo	Tipo	Descrição
jogo	INT NOT NULL	Número do jogo. Chave.
data	DATE	Data do jogo.
pais1	VARCHAR(20)	Nome de um dos países.
pais2	VARCHAR(20)	Nome do outro país.
gols1	INT DEFAULT '0' NOT NULL	Número de gols do pais1
gols2	INT DEFAULT '0' NOT NULL	Número de gols do pais2

Tabela 7.1: Definição da tabela SQL para aplicação copa2002.

A criação da tabela `copa2002` é realizada através do seguinte comando SQL:

```
create table copa2002 (
  jogo int not null,
  data date,
  pais1 varchar(20),
  pais2 varchar(20),
  gols1 int default '0' not null,
  gols2 int default '0' not null,
  primary key (jogo))
```

7.3 Interface da Aplicação

A página inicial da aplicação `copa2002` é mostrada na Figura 7.1. O usuário faz sua escolha selecionando uma das opções modeladas como *radio button* e, em seguida, clicando sobre o botão rotulado com `Execute`.

Se durante a utilização da aplicação o banco de dados estiver inacessível por qualquer motivo então o usuário receberá do servidor a página mostrada na Figura 7.2.

7.3.1 Incluindo Novo Jogo

A página que permite ao usuário incluir novo jogo é mostrada na Figura 7.3. O usuário preenche os campos indicados e submete o formulário clicando

[1] Para criá-la, use o comando SQL[2] `create database xxe`.

Figura 7.1: copa2002 - Página incial da aplicação.

sobre o botão rotulado como `Inclua`. Caso o número do jogo já tenha sido cadastrado, o servidor retorna a página mostrada na Figura 7.4.

Figura 7.2: copa2002 - Página indicando banco de dados inacessível.

Figura 7.3: copa2002 - Página de inclusão de jogo.

Figura 7.4: copa2002 - Página indicando erro na inclusão de jogo.

7.3.2 Mostrando Todos os Jogos

A Figura 7.5 mostra a página enviada pelo servidor quando o usuário deseja ver todos os jogos já cadastrados.

Aplicação Copa 2002

Todos os Jogos

Jogo	Data	País	Placar	País
1	2002-05-31	França	2 x 1	Camarões
2	2002-06-01	Brasil	3 x 5	China
3	2002-06-03	Porto Rico	7 x 1	Brasil
4	2002-06-23	Brasil	6 x 2	Portugal
5	2002-06-06	Cuba	8 x 0	Portugal
6	2002-06-28	Itália	0 x 0	Coréia
7	2002-06-08	França	9 x 3	Japão
10	2002-06-23	Olaria	3 x 2	Polenteiro

Figura 7.5: copa2002 - Página mostrando todos os jogos.

7.3.3 Mostrando Jogos de Um País

A Figura 7.6 mostra a página enviada pelo servidor quando o usuário deseja ver todos os jogos de um país. O usuário deve digitar o nome de um país no campo apropriado e clicar no botão `Pesquise Por País`.

Após o formulário da Figura 7.6 ter sido enviado para o servidor, este retorna a página com todos os jogos do país solicitado, como mostra a Figura 7.7.

Figura 7.6: copa2002 - Página solicitando nome do país.

Figura 7.7: copa2002 - Página com os jogos de um país.

.

7.4 Modelagem da Aplicação

Por uma questão de clareza didática, os arquivos que fazem parte da aplicação `copa2002` são listados em três tabelas. Estes arquivos foram agrupados segundo sua extensão (`.jsp`, `.html` e `.java`). A Tabela 7.2 mostra a relação de arquivos do tipo `.jsp`. A Tabela 7.3 mostra a relação de arquivos do tipo `.html`. Finalmente a Tabela 7.4 mostra a relação de arquivos do tipo `.java`. O arquivo descritor de instalação (`web.xml`) é o mesmo usado nos exemplos apresentados no Capítulo 5 e, por isso, não é mostrado.

Arquivo	Finalidade
copa2002\index.jsp	Gerar a página inicial da aplicação, descrito na Figura 7.8.
copa2002\bdinacessivel.jsp	Gerar a página que indica que o banco de dados está inacessível, descrito na Figura 7.9.
copa2002\chaveDuplicada.jsp	Gerar a página indica erro na inclusão de novo jogo, descrito na Figura 7.10.
copa2002\inclusaoForm.jsp	Gerar a página usada para incluir novo jogo, descrito na Figura 7.11.
copa2002\procurarPorPais.jsp	Gerar a página que solicita o nome do país do qual o usuário deseja ver todos os jogos, descrito na Figura 7.12.
copa2002\todos.jsp	Gerar a página que mostra o resultado das pesquisas (todos os jogos cadastrados e todos os jogos de um determinado país), descrito na Figura 7.13.
copa2002\controlador.jsp	Controlar o fluxo de execução da aplicação, descrito na Figura 7.14.

Tabela 7.2: Arquivos `.jsp` da aplicação copa2002

Arquivo	Finalidade
copa2002\cabecalho.html	Conter o cabeçalho das páginas da aplicação. Este arquivo segue o princípio dos cabeçalhos dos exemplos mostrados no Capítulo 5.
copa2002\menuPrincipal.html	Conter as opções mostradas ao usuário na página inicial da aplicação, descrito na Figura 7.15.
copa2002\fichaJogo.html	Conter os campos do formulário usado para cadastrar novo jogo, descrito na Figura 7.16.
copa2002\fichaPesqPais.html	Conter os campos do formulário usado para pesquisar jogos de um país, descrito na Figura 7.17.

Tabela 7.3: Arquivos `.html` da aplicação copa2002

Arquivo	Finalidade
copa2002\classes \ljk\copa2002\bd\BD.class	Modelar o javabeanresponsável por acessar o banco de dados, descrito na Figura 7.18.

Tabela 7.4: Arquivos `.java` da aplicação copa2002

7.4.1 Arquivo `index.jsp`

O arquivo `index.jsp`, mostrado na Figura 7.8, gera a página incial da aplicação `copa2002`.

```
<html>
<head>
<title>Aplicação web - Copa 2002</title>
</head>
<body>
<%@include file="cabecalho.html"%>

<h2> Menu Principal</h2>
<form type=get action=controlador.jsp>
  <%@include file="menuPrincipal.html"%><br>
  <input type=submit value="Execute">
</form>
</body>
</html>
```

Figura 7.8: Aplicação copa2002: arquivo index.jsp.

Observe que os campos do formulário são obtidos pela inclusão do arquivo `menuPrincipal.html`. Quando o usuário clica sobre o botão `Execute` os dados do formulário são processados pelo arquivo `controlador.jsp`, seguindo assim o modelo Modelo 2 discutido na seção 3.2 (página 22).

7.4.2 Arquivo `bdInacessivel.jsp`

O arquivo `bdInacessivel.jsp`, mostrado na Figura 7.9, gera a página que indica para o usuário da aplicação `copa2002` que o banco de dados acessado por esta aplicação está inacessível.

7.4.3 Arquivo `chaveDuplicada.jsp`

O arquivo `chaveDuplicada.jsp`, mostrado na Figura 7.10, gera a página que indica para o usuário da aplicação `copa2002` que houve um problema na inclusão de um novo jogo. O problema em questão é que o banco de dados já contém um registro cujo campo `jogo` é igual ao valor fornecido pelo usuário.

Observe que para incluir na página o número do jogo em questão é necessário recuperar esta informação com o objeto implícito `request`.

```
<%@page isErrorPage="true"%>
<html>
<head><title>Aplicação web - Copa 2002</title></head>

<body>
<%@include file="cabecalho.html"%>
<h2> Oops! </h2>
<h3>O banco de dados está inacessível...</h3>
<h3>Aguarde um tempo e
    <a href="index.jsp">tente novamente</a></h3>
</body>
</html>
```

Figura 7.9: Aplicação copa2002: arquivo bdInacessivel.jsp.

7.4.4 Arquivo `inclusaoForm.jsp`

O arquivo `inclusaoForm.jsp`, mostrado na Figura 7.11, gera a página que solicita ao usuário da aplicação `copa2002` os dados do jogo a ser incluído na base de dados. Observe que os campos do formulário estão definidos no arquivo `fichaJogo.html`.

7.4.5 Arquivo `procuraPorPais.jsp`

O arquivo `procuraPorPais.jsp`, mostrado na Figura 7.12, gera a página que solicita ao usuário da aplicação `copa2002` o nome do país do qual se deseja conhecer todos os jogos. Observe que os campos do formulário estão definidos no arquivo `fichaPesqPais.html`.

7.4.6 Arquivo `todos.jsp`

O arquivo `todos.jsp`, mostrado na Figura 7.13, tem como finalidade gerar uma página contendo uma tabela de jogos. Os dados usados para gerar a tabela estão disponíveis no JavaBean identificado por `oBD` (intância da classe `ljk.copa2002.bd.BD`).

Observe que este arquivo é usado tanto para mostrar todos os jogos armazenados no banco de dados como somente os jogos de um determinado país. Não há, neste arquivo, **nenhuma** informação a respeito da lógica utilizada para gerar os dados que formam a tabela.

```
<html>
<head>
<title>Aplicação web - Copa 2002</title>
</head>
<body>
<%@include file="cabecalho.html"%>

<h2> Erro Na Inclusão </h2>

<h3> O jogo de número <%= request.getParameter("jogo") %>
já foi cadastrado.</h3>

</body>
</html>
```

Figura 7.10: Aplicação copa2002: arquivo chaveDuplicada.jsp.

7.4.7 Arquivo `controlador.jsp`

O arquivo `controlador.jsp`, mostrado na Figura 7.14, tem como finalidade **definir a lógica de resolução do problema da aplicação** `copa2002`. Este arquivo, diferentemente de todos os demais, funciona literalmente como um algoritmo e não como uma página a ser exibida no browser do cliente. Ele corresponde, portanto, ao elemento **controlador** do Modelo 2 (descrito na seção 3.2, página 22).

O algoritmo funciona basicamente assim:

1. Com exceção da página inicial (arquivo `index.jsp`), toda solicitação feita pelo usuário é encaminhada à página `controlador.jsp` e a vontade do usuário é representada pelo valor do parâmetro `opcao` armazenado no objeto implícito `request`.

2. O controlador identifica o valor de `opcao` e realiza a operação pertinente. Esta etapa envolve duas atividades:

   (a) É executado um algoritmo que, normalmente, envolve o JavaBean responsável por acessar o banco de dados.

   (b) Em função do resultado da execução do passo anterior, determina-se para qual página a solicitação do usuário deve ser encaminhada.

3. A solicitação é encaminhada para uma outra página (definida na variável `proxPagina`), usando-se para isso a tag `<jsp:forward>`. O

```
<html>
<head>
<title>Aplicação web - Copa 2002</title>
</head>
<body>
<%@include file="cabecalho.html"%>

<h2> Incusão de Nova Partida</h2>
<form type=get action=controlador.jsp>
  <%@include file="fichaJogo.html"%><br>
  <input type="hidden" name="opcao" value="Inclua">
  <input type=submit value="Inclua">
</form>
</body>
</html>
```

Figura 7.11: Aplicação copa2002: arquivo inclusaoForm.jsp.

parâmetro extra `titulo` é usado para caracterizar a finalidade da página `todos.jsp`. Quando esta página não está envolvida o parâmetro é ignorado.

7.4.8 Arquivo `menuPrincipal.html`

O arquivo `menuPrincipal.html`, mostrado na Figura 7.15, contém as opções disponibilizadas para o usuário na página inicial da aplicação. Observe que todas as tags `<input>` têm valor `opcao` para o atributo `name`. É esta combinação de atributo-valor que sinaliza para o controlador (`controlador.jsp`) a intenção do usuário. A intenção, por sua vez, é explicitada no valor associado ao atributo `value` de cada tag.

Observe que, em função desta forma de organizar o menu principal, a mudança na forma do menu principal pode ser facilmente implementada, desde que se mantenha o padrão de nomes descrito no parágrafo anterior.

7.4.9 Arquivo `fichaJogo.html`

O arquivo `fichaJogo.html`, mostrado na Figura 7.16, contém os campos do formulário que deve ser preenchido quando o usuário deseja incluir um novo jogo na base de dados.

Observe que a disposição (*layout*) dos campos pode ser modificada livremente. A única restrição é que devem ser mantidos os valores dos atributos

```
<html>
<head>
<title>Aplicação web - Copa 2002</title>
</head>
<body>
<%@include file="cabecalho.html"%>

<h2> Pesquisa por País</h2>
<form type=get action=controlador.jsp>
  <%@include file="fichaPesqPais.html"%><br>
  <input type="hidden" name="opcao" value="pesq_por_pais">
  <input type=submit value="Pesquise por País">
</form>
</body>
</html>
```

Figura 7.12: Aplicação copa2002: arquivo procuraPorPais.jsp.

`name` das tags.

7.4.10 Arquivo `fichaPesqPais.html`

O arquivo `fichaPesqPais.html`, mostrado na Figura 7.17, contém os campos do formulário que deve ser preenchido quando o usuário deseja ver todos os jogos de um determinado país. No caso em questão existe um único campo referente ao nome do país.

Observe que a disposição (*layout*) do campo pode ser modificada livremente. A única restrição é que deve ser mantido o valor do atributo `name` da tag `<input>`.

```
<%@page import = "java.sql.*"%>
<jsp:useBean id="oBD" class="ljk.copa2002.bd.BD" scope="session"/>

<%! ResultSet rs; %>
<% rs = oBD.getResultSet(); %>

<%! String pais1, pais2;%>
<%! int jogo, gols1, gols2;%>
<%!Date data;%>

<html>
<head>
<title>Aplicação web - Copa 2002</title>
</head>
<body>
<%@include file="cabecalho.html"%>

<h2> <%= request.getParameter("titulo")%></h2>

<table border=1>
  <tr bgcolor="yellow"><th>Jogo</th><th>Data</th> <th>País</th>
      <th>Placar</th><th>País</th>
  </tr>
<%
  while (rs.next()) {
    jogo = rs.getInt("jogo");
    data = rs.getDate("data");
    pais1 = rs.getString("pais1");
    pais2 = rs.getString("pais2");
    gols1 = rs.getInt("gols1");
    gols2 = rs.getInt("gols2");
    %><tr><td><%=jogo%></td>
          <td><%=data%></td>
          <td><%=pais1%></td>
          <td><%=gols1 + " x " + gols2%></td>
          <td><%=pais2%></td>
       </tr><%
  }
%>
</table>
</body>
</html>
```

Figura 7.13: Aplicação copa2002: arquivo todos.jsp.

```
<%@page import = "java.sql.*" errorPage="bdInacessivel.jsp"%>
<jsp:useBean id="oBD" class="ljk.copa2002.bd.BD" scope="session"/>


<%! String opcao, proxPagina, titulo; %>
<%! boolean processou; %>
<%
  proxPagina = "bdInacessivel.jsp";
  oBD.crieConexao();
  opcao = request.getParameter("opcao");
  if (opcao.equals("incluir")) {
    proxPagina = "inclusaoForm.jsp";
  }
  else if (opcao.equals("todos")) {
    processou = oBD.procureTodos();
    if (processou) {
        proxPagina = "todos.jsp";
        titulo = "Todos os Jogos";
    }
    }
  else if (opcao.equals("Inclua")) {
    try {
    oBD.incluaJogo(Integer.parseInt(request.getParameter("jogo")),
                   Date.valueOf(request.getParameter("data")),
                   request.getParameter("pais1"),
                   request.getParameter("pais2"),
                   Integer.parseInt(request.getParameter("gols1")),
                   Integer.parseInt(request.getParameter("gols2")));
    proxPagina = "index.jsp";
    }
    catch(Exception e) {
      proxPagina = "chaveDuplicada.jsp";
    }
  }
  else if (opcao.equals("procpais")) {
    proxPagina = "procurarPorPais.jsp";
  }
  else if (opcao.equals("pesq_por_pais")) {
    oBD.procurePais(request.getParameter("pais"));
    proxPagina = "todos.jsp";
    titulo = "Todos os Jogos do País " + request.getParameter("pais");
  }

%>
<jsp:forward page = '<%= proxPagina %>'>
  <jsp:param name="titulo" value='<%= titulo%>'/>
</jsp:forward>
```

Figura 7.14: Aplicação copa2002: arquivo controlador.jsp.

```
<input type="radio" name="opcao"
       value="incluir">  Incluir jogo.<br>
<input type="radio" name="opcao"
       value="todos" checked="true"> Mostrar todos os jogos.<br>
<input type="radio" name="opcao"
       value="procpais"> Mostrar jogos de um país.<br>
```

Figura 7.15: Aplicação copa2002: arquivo menuPrincipal.html.

```
Jogo <input type="text" name="jogo" size="2">
Data <input type="text" name="data" size="10">
<br>
País <input type="text" name="pais1" size="15">
<input type="text" name="gols1" size="2"> X
<input type="text" name="gols2" size="2">
<input type="text" name="pais2" size="15">País
```

Figura 7.16: Aplicação copa2002: arquivo fichaJogo.html.

```
Nome do país desejado <input type="text" name="pais" size="15">
```

Figura 7.17: Aplicação copa2002: arquivo fichaPesqPais.html.

7.4.11 Arquivo `BD.java`

O arquivo `BD.java`, mostrado na Figura 7.18, contém o código fonte da classe `ljk.copa2002.bd.BD`. Esta classe representa um JavaBean cuja finalidade é acessar o banco de dados utilizado na aplicação `copa2002`. A atuação deste JavaBean sempre está a serviço de alguma página que usa a tecnolgia JSP.

O atributo **con** representa uma conexão com o banco. O atributos `sttTodos`, `sttPais` e `sttInclua` representam comandos SQL usado na consulta à base de dados. O primeiro é usado para obter todos os jogos armazenados. O segundo é usado para obter todos os jogos de umpaís. E o terceiro é usado para incluir um novo jogo no banco.

O atributo `url` identifica onde está localizado (em que computador) o banco de dados e qual base será usada. O banco de dados pode estar localizado em qualquer computador conectado à Internet. Finalmente o atributo `rs` representa as linhas da tabela resultantes das consultas.

O conteúdo dos métodos representados por "..." na Figura 7.18 são visualizados e comentados nas subseções que seguem.

7.4.11.1 Classe BD: método construtor

A Figura 7.19 mostra o método construtor da classe BD. Observe o valor do atributo `url`. Ele indica que está sendo usado o banco de dados MySQL, que está no mesmo computador onde a aplicação `copa2002` está sendo executada e que será usada a base de dados `xxe`.

7.4.11.2 Classe BD: método `crieConexao()`

A Figura 7.20 mostra o método que faz a conexão como banco de dados. A string `"org.gjt.mm.mysql.Driver"` representa o nome do driver JDBC para o banco MySQL. Este driver está definido em um arquivo `.jar` cujo nome depende do fabricante (não há uma norma). Este arquivo deve estar dentro do diretório `lib` da aplicação.

7.4.11.3 Classe BD: método `incluaJogo()`

A Figura 7.21 mostra o método que inclui na base de dados os dados relativos ao novo jogo.

7.4.11.4 Classe BD: método `processeStatement()`

A Figura 7.22 mostra o método que executa o código SQL correspondente ao tipo de acesso (consulta ou inclusão) ao banco de dados. Observe que o resultado da consulta é armazenado no atributo `rs`.

```
package ljk.copa2002.bd;
import java.sql.*;

public class BD {
  Connection con;
  PreparedStatement sttTodos, sttPais, sttInclua;
  String url;
   ResultSet rs;

  public BD() {
    ...
  }

  public void crieConexao() throws Exception {
    ...
  }

  public boolean procureTodos() throws Exception {
        return processeStatement(sttTodos);
  }

  public boolean procureJogo(int jogo) throws Exception {
    sttJogo.setInt(1,jogo);
    return processeStatement(sttJogo);
  }

  public boolean procurePais(String pais) throws Exception {
    sttPais.setString(1,pais);
    sttPais.setString(2,pais);
    return processeStatement(sttPais);
  }

  public boolean incluaJogo(int jogo, Date data,
                            String pais1, String pais2,
                            int gols1,int gols2) throws Exception {
     ...
  }

  protected boolean processeStatement(PreparedStatement s)
                                    throws Exception  {
     ...
  }

    public ResultSet getResultSet()  {
        return(rs);
    }
}
```

Figura 7.18: Aplicação copa2002: arquivo BD.java.

```
...
public BD() {
 con = null;
 url = "jdbc:mysql://localhost/xxe";
 rs = null;
}
...
```

Figura 7.19: Aplicação copa2002: método construtor da classe BD.

```
...
public void crieConexao() throws Exception {
    if (con == null) {
      Class.forName("org.gjt.mm.mysql.Driver");
      con = DriverManager.getConnection(url);
      sttTodos = con.prepareStatement("select * from copa2002
                                        order by jogo");
      sttPais = con.prepareStatement("select * from copa2002
                                      where pais1 = ? or pais2 = ?
                                      order by jogo");
      sttInclua = con.prepareStatement("insert into copa2002
                                         values (?, ?, ?, ?, ?, ?)");
    }
}
...
```

Figura 7.20: Aplicação copa2002: método crieConexao da classe BD.

```
...
public boolean incluaJogo(int jogo, Date data,
                          String pais1, String pais2,
                          int gols1,int gols2) throws Exception {
  sttInclua.setInt(1,jogo);
  sttInclua.setDate(2,data);
  sttInclua.setString(3,pais1);
  sttInclua.setString(4,pais2);
  sttInclua.setInt(5,gols1);
  sttInclua.setInt(6,gols2);
  return processeStatement(sttInclua);
}
...
```

Figura 7.21: Aplicação copa2002: método incluaJogo da classe BD.

```
...
protected boolean processeStatement(PreparedStatement s)
                                    throws Exception  {
      boolean resultado;
      try {
        rs = null;
        resultado = s.execute();
        if (resultado)
           rs = s.getResultSet();
        return(resultado);
      }
      catch (Exception e) {
        con = null;
        throw e;
      }
  }
...
```

Figura 7.22: Aplicação copa2002: método processeStatement da classe BD.

Apêndice A

Instalando Aplicações Web com Ant

Ant é um software[1] escrito em Java que permite a criação de scripts para a realização automática de uma série de tarefas como, por exemplo, a compilação de programas Java. Os scripts são definidos em um arquivo no formato XML.

Durante o processo de desenvolvimento de aplicações para web, muitas tarefas precisam ser realizadas continuamente. O ciclo de atividades envolve normalmente os seguintes passos:

1. Edição do conteúdo estático da aplicação (páginas HTML, imagens, etc.).
2. Edição do conetúdo dinâmico da aplicação (páginas JSP).
3. Edição dos JavaBeans usados na aplicação (programas Java).
4. Compilação dos JavaBeans.
5. Geração do arquivo que contém a aplicação (arquivo `.war`). Este arquivo contém todos os arquivos definidos nas etapas anteriores, organizados na forma indicada na seção 3.3 (página 23).
6. Instalação[2] do arquivo `.war` no servidor JSP.
7. Reinicialização do servidor JSP para que a nova versão da aplicação seja carregada.

Usando a ferramenta **ant** podemos realizar automaticamente (digitando apenas um único comando) todas as atividades a partir da compilação dos JavaBeans [Goo01a]. O **ant** possui uma grande quantidade de tarefas que já

[1] Disponível para *download* no endereço `http://jakarta.apache.org/ant`.
[2] No original, *deployment*.

estão disponíveis no software e, normalmente, são suficientes para a maioria dos casos. No presente contexto, isto é, instalação de aplicações web em servidor JSP, as tarefas existentes são suficientes.

A.1 Definindo Arquivo `build.xml`

A definição das tarefas implica na criação de um arquivo no formato XML normalmente chamado `build.xml`. Este arquivo é formado por **propriedades** e **targets**. As propriedades definem nomes, diretórios, arquivos, etc. As targets definem conjuntos de tarefas.

Uma vez criado o arquivo **build.xml**, sua utilização pelo ant é realizada digitando-se, em algum console, o comando mostrado a seguir.

```
ant target
```

Várias IDEs modernas já apresentam suporte direto ao ant e sua forma de ativação, naturalmente, pode ser diferente.

A.2 Tarefas no Desenvolvimento de Aplicações para Web

Nesta seção são apresentadas as várias tarefas usadas no desenvolvimento de aplicações para web.

```
<?xml version="1.0" encoding = "ISO-8859-1"?>
<project name="exemplo-sessao"
         default="ajuda" basedir=".">

<!-- Criado por : Leandro J. Komosinski (leandro@inf.ufsc.br) -->


<!-- Propriedades Relativas ao Tomcat -->
<property environment="e"/>
<property name="catalina.home"
          value="${e.CATALINA_HOME}"/>
<property name="linux.tomcat.liga"
          value="${catalina.home}/bin/startup.sh"/>
<property name="linux.tomcat.desliga"
          value="${catalina.home}/bin/shutdown.sh"/>
<property name="win.tomcat.desliga"
          value="${catalina.home}/bin/shutdown.bat"/>
<property name="webapps.home"
          value="${catalina.home}/webapps"/>

...
</project>
```

Figura A.1: Propriedades do servidor JSP Tomcat.

```
...
<!-- Propriedades Relativas a Aplicacao Web -->
<property name="webxml.home"  value="./meta"/>
<property name="dist.home"    value="./dist"/>
<property name="html.home"    value="./src/html"/>
<property name="jars.home"    value="./jars"/>
<property name="fontes.home"  value="./src/java"/>
<property name="classes.home" value="./classes"/>

<path id="classpath">
  <fileset dir="${jars.home}">
    <include name="**/*.jar"/>
  </fileset>
  <fileset dir="${classes.home}">
    <include name="**/*.class"/>
  </fileset>
</path>

...
```

Figura A.2: Propriedades comuns a todas as aplicações.

```
...
<property name="app.name"    value="flags"/>
<property name="webxml.name" value="flags_web.xml"/>

<property name="app.home"  value="${webapps.home}/${app.name}"/>
<property name="web.xml"   value="${webxml.home}/${webxml.name}"/>
<property name="app.war"   value="${dist.home}/${app.name}.war"/>
...
```

Figura A.3: Propriedades específicas de cada aplicação.

```
...
<!-- Target     : init

     Finalidade : preparar diretorios e propriedades usados em
                  outros targets.
     Como faz   : - cria diretorios para arquivos .class e .war
                  - verifica se o OS eh da familia Windows.
     Observacao : O Ant nao executa tarefas assincronamente no Windows.
                  Isso impede, por exemplo, que o Tomcat seja carregado
                  via Ant.
-->
<target name="init">
  <mkdir dir="${classes.home}"/>
  <mkdir dir="${dist.home}"/>
  <condition property="windows.presente">
    <os family="windows"/>
  </condition>
</target>
```

Figura A.4: Target init.

```
...
<!-- Target     : ajuda
     Finalidade : mostrar opcoes de uso ao usuario.
     Como faz   : mostra cada opcao na tela.
     Observacao : target default.
-->
<target name="ajuda">
  <echo message = "Uso:"/>
  <echo message = "ant liga      - carrega o Tomcat"/>
  <echo message = "ant desliga   - encerra o Tomcat"/>
  <echo message = "ant compila   - compila java beans"/>
  <echo message = "ant gera_war  - gera .war em ./dist"/>
  <echo message = "ant copia_war - copia .war para CATALINA_HOME/webapps"/>
  <echo message = "ant deploy    - faz o deployment da aplicacao web"/>
  <echo message = "ant refaz     - apaga versao atual e gera novo .war"/>
  <echo message = "ant tudo      - refaz e deploy"/>
</target>
...
```

Figura A.5: Target ajuda.

```
...
<!-- Target     : refaz
     Finalidade : Recompilar toda a aplicacao web.
     Como faz   : - apaga diretorios que contem as classes java compiladas e
                    e arquivo .war .
-->
<target name="refaz" depends="apaga,gera_war"/>
```

Figura A.6: Target refaz.

```
...
<!-- Target     : apaga
     Finalidade : Apagar os diretorios gerados pela compilacao da aplicacao.
     Como faz   : - apaga o diretorio de classes java (arquivos .class)
                  - apaga o diretorio que contem o arquivo .war .
-->
<target name="apaga">
  <delete dir="${classes.home}"/>
  <delete dir="${dist.home}"/>
</target>
...
```

Figura A.7: Target apaga.

```
...
<!-- Target     : tudo
     Finalidade : Realiza o ciclo completo de geracao da aplicacao e seu
                  deployment no Tomcat.
     Como faz   : - recompila a aplicacao web (gerando arquivo .war).
                  - instala a aplicacao no Tomcat.
     Observacao : o Tomcat ja deve ter sido ligado antes.
-->
<target name="tudo" depends="refaz,deploy"/>
...
```

Figura A.8: Target tudo.

```
...
<!-- Target     : deploy
     Finalidade : Faz o deployment da aplicacao web no Tomcat.
     Como faz   : - desliga (shutdown) o Tomcat.
                  - copia o arquio .war para "dentro" do Tomcat.
                  - liga (startup) o Tomcat.
     Observacao : o Tomcat ja deve ter sido ligado antes.
-->
<target name="deploy" depends="desliga,copia_war,liga"/>
...
```

Figura A.9: Target deploy.

```
...
<!-- Target     : copia_war
     Finalidade : Copiar o arquivo .war para "dentro" do Tomcat.
     Como faz   : - apaga o diretorio que contem a aplicacao dentro
                    do Tomcat.
                  - copia a nova versao do arquivo .war para o Tomcat.
-->
<target name="copia_war" depends="gera_war">
  <delete dir="${app.home}"/>
  <copy file="${app.war}" todir="${webapps.home}"/>
</target>
...
```

Figura A.10: Target copia_war.

```
...
<!-- Target     : gera_war
     Finalidade : Gerar o arquivo .war.
     Como faz   : - usa a tarefa "war".
-->
<target name="gera_war" depends="compila">
  <war warfile="${app.war}" webxml="${web.xml}">
     <lib dir="${jars.home}"/>
     <classes dir = "${classes.home}"/>
     <fileset dir = "${html.home}"/>
  </war>
</target>
...
```

Figura A.11: Target gera_war.

```
...
<!-- Target     : compila
     Finalidade : Compilar as classes Java que fazem parte da aplicacao.
     Como faz   : - usa a tarefa javac.
     Observacao : so recompila os arquivos fontes que foram modificados.
-->
<target name="compila" depends="init">
  <javac srcdir="${fontes.home}"
         destdir="${classes.home}" classpathref="classpath"/>
</target>
...
```

Figura A.12: Target compila.

```
...
<!-- Target     : liga
     Finalidade : Ativar o Tomcat.
     Como faz   : - executa o comando $CATALINA_HOME/bin/startup.sh
     Observacao : So carrega automaticamente no Linux. No Windows, o
                  carregamento tem que ser manual (via s.bat).
-->
<target name="liga" depends="init">
  <ant target="win_liga"/>
  <ant target="linux_liga"/>
</target>
<target name="win_liga" if="windows.presente">
  <echo message = "** Carregando o Tomcat **"/>
  <echo message = "Para carregar o Tomcat no Windows digite: s.bat"/>
</target>

<target name="linux_liga" unless="windows.presente">
  <echo message = "** Carregando o Tomcat **"/>
  <exec executable="${linux.tomcat.liga}"/>
  <sleep seconds="10"/>
  <echo message = "Feito!"/>
</target>
...
```

Figura A.13: Target liga.

```
...
<!-- Target     : desliga
     Finalidade : Encerrar o Tomcat.
     Como faz   : - executa o comando $CATALINA_HOME/bin/shutdown.sh
-->
<target name="desliga" depends="init">
  <ant target="linux_desliga"/>
  <ant target="win_desliga"/>
  <sleep seconds="10"/>
  <echo message = "Feito!"/>
</target>

<target name="win_desliga" if="windows.presente">
  <exec executable="${win.tomcat.desliga}"/>
</target>

<target name="linux_desliga" unless="windows.presente">
  <exec executable="${linux.tomcat.desliga}"/>
</target>
...
```

Figura A.14: Target desliga.

Apêndice B

Tomcat: Um Servidor JSP

Tomcat é um servidor JSP implementado em Java [Goo01a]. Desenvolvido pela Fundação Apache[1], o Tomcat é a implementação referência das especificações de Servlet 2.3 e JSP 1.2 (que são as mais recentes).

Informações e download do software podem ser obtidas em `http://jakarta.apache.org/tomcat`. Sua instalação é bastante simples e pode ser usado diretamente sem nenhum ajuste inicial [Goo01b].

Na configuração default, o servidor Tomcat define a porta 8080 para aguardar requisições dos clientes. Neste caso o endereço que deve ser fornecido no programa cliente (normalmente um browser) é o seguinte:

http://*host*:8080/*aplicação*/

Por exemplo, para acessar a aplicação `copa2002` instalada no seu próprio computador deve-se digitar:

http://localhost:8080/copa2002/

O Tomcat pode ser usado como servidor HTTP também. Isto implica que ele deve ser configurado (modificando o arquivo `server.xml`) para aguardar requisições de clientes na porta 80.

B.1 Principais Arquivos

Considere que o Tomcat foi instalado no diretório indicado pela variável de ambiente `%CATALINA_HOME%`. Seus principais arquivos são descritos na Tabela B.1.

[1] Veja o *site* em `http://www.apache.org`

Arquivo	Finalidade
\bin\startup.bat	Inicia a execução do Tomcat.
\bin\shutdown.bat	Encerra a execução do Tomcat.
\conf\server.xml	Arquivo de configuração do Tomcat. É neste arquivo que estão todas as opções de configuração do servidor, inclusive a definição de qual porta o programa usa (o default é 8080). A definição de comunicação segura (via protocolo `https`) também é feita neste arquivo.
\conf\tomcat-users.xml	Indica relação de usuários com privilégios especiais.
\webapps\	Diretório que contém as aplicações gerenciadas pelo Tomcat.

Tabela B.1: Principais arquivos do Tomcat.

Bibliografia

[Ada01] Milan Adamovic. Process jsps effectively with javabeans. transport jsp processing logic into a javabean with the template method design pattern. Java World Magazine, http://www.javaworld.com/javaworld/jw-01-2001/jw-0119-jspframe.html, january 2001.

[Cha00] Alex Chaffee. One, two, three, or n tiers? should you hold back the tiers of your application? Java World Magazine, http://www.javaworld.com/javaworld/jw-01-2000/jw-01-ssj-tiers.html, january 2000.

[Fie99] Duane K. Fields. The nuts and bolts of relational databases. a primer for web developers. http://developer.iplanet.com/viewsource/fields_db/fields_db.html, december 1999.

[Fie00] Duane K. Fields. Adding database support with jdbc. a primer for web developers. http://developer.iplanet.com/viewsource/fields_jdbc2/fields_jdbc2.html, february 2000.

[FK00] Duane K. Fields and Mark Kolb. Building your own jsp components. http://developer.iplanet.com/viewsource/fields_jspcomp/fields_jspcomp_p%.html, may 2000.

[Gea00] David Geary. Jsp templates. Java World Magazine, http://www.javaworld.com/javaworld/jw-09-2000/jw-0915-jspweb.html, september 2000.

[Goo01a] James Goodwill. Deploying web applications to tomcat. The O'Reilly Network, http://www.onjava.com/pub/a/onjava/2001/04/19/tomcat.html, april 2001.

[Goo01b] James Goodwill. Installing and configuring tomcat. The O'Reilly Network, http://www.onjava.com/lpt/a//onjava/2001/03/29/tomcat.html, march 2001.

[Gou00] Steven Gould. Develop n-tier applications using j2ee. Java World Magazine, http://www.javaworld.com/javaworld/jw-12-2000/jw-1201-weblogic.html, december 2000.

[jGu00] jGuru.com, Java Developer Connection, http://developer.java.sun.com/developer/onlineTraining/JSPIntro/. *JavaServer Pages Fundamentals - Short Course*, september 2000.

[Jún02] Francisco Tarcizo Bomfim Júnior. *JSP - A Tecnologia Java na Internet*. Editora Érica, 2002.

[Mah01a] Qusay H. Mahmoud. Web application development with jsp and xml. part i: Fast track jsp. Java Developer Connection, http://developer.java.sun.com/developer/technicalArticles/xml/WebAppDev%/, june 2001.

[Mah01b] Qusay H. Mahmoud. Web application development with jsp and xml. part ii: Jsp with xml in mind. Java Developer Connection, http://developer.java.sun.com/developer/technicalArticles/xml/WebAppDev%2/, july 2001.

[Mah01c] Qusay H. Mahmoud. Web application development with jsp and xml. part iii: Developing jsp custom tags. Java Developer Connection, http://developer.java.sun.com/developer/technicalArticles/xml/WebAppDev%3/, august 2001.

[Mah01d] Qusay H. Mahmoud. Web application development with jsp and xml. part iv: Using j2ee services from jsp. Java Developer Connection, http://developer.java.sun.com/developer/technicalArticles/xml/WebAppDev%4/, october 2001.

[Mar01] Dustin Marx. Jsp best practices. follow these tips for reusable and easily maintainable javaserver pages. Java World Magazine, http://www.javaworld.com/javaworld/jw-11-2001/jw-1130-jsp.html, november 2001.

[Paw01] Monica Pawlan. Introduction to the j2ee platform. Java Developer Connection, http://developer.java.sun.com/developer/technicalArticles/J2EE/Intro/in%dex.html, march 2001.

[RAJ02] Ed Roman, Scott Ambler, and Tyler Jewell. *Mastering Enterprise JavaBeans*. John Wiley & Sons, Inc., 2 edition, 2002.

[Rom99] Ed Roman. *Mastering Enterprise JavaBeans*. John Wiley & Sons, Inc., 1999.

[Sec99] Govind Sechadri. Understanding javaserver pages model 2 architecture. exploring the mvc design pattern. Java World Magazine, http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html, december 1999.