TagFile, JSTL, ORM e JSF

Sumário

XML

Extensible Markup Language (XML) é linguagem de marcação de dados (meta-markup language) que provê um formato para descrever dados estruturados. Isso facilita declarações mais precisas do conteúdo e resultados mais significativos de busca através de múltiplas plataformas. O XML também vai permitir o surgimento de uma nova geração de aplicações de manipulação e visualização de dados via internet.

O XML permite a definição de um número infinito de tags. Enquanto no HTML, se as tags podem ser usadas para definir a formatação de caracteres e parágrafos, o XML provê um sistema para criar tags para dados estruturados.

Um elemento XML pode ter dados declarados como sendo preços de venda, taxas de preço, um título de livro, a quantidade de chuva, ou qualquer outro tipo de elemento de dado. Como as tags XML são adotadas por intranets de organizações, e também via Internet, haverá uma correspondente habilidade em manipular e procurar por dados independentemente das aplicações onde os quais são encontrados. Uma vez que o dado foi encontrado, ele pode ser distribuído pela rede e apresentado em um browser como o Internet Explorer 5 de várias formas possíveis, ou então esse dado pode ser transferido para outras aplicações para processamento futuro e visualização.

Comparações entre HTML e XML

HTML e XML são primos. Eles derivam da mesma inspiração, o SGML. Ambos identificam elementos em uma página e ambos utilizam sintaxes similares. Se você é familiar com HTML, também o será com o XML. A grande diferença entre HTML e XML é que o HTML descreve a aparência e a ações em uma página na rede enquanto o XML não descreve nem aparência e ações, mas sim o que cada trecho de dados é ou representa ! Em outras palavras, o XML descreve o conteúdo do documento !

Como o HTML, o XML também faz uso de tags (palavras encapsuladas por sinais '<' e '>') e atributos (definidos com name="value"), mas enquanto o HTML especifica cada sentido para as tags e atributos (e frequentemente a maneira pela qual o texto entre eles será exibido em um navegador), o XML usa as tags somente para delimitar trechos de dados, e deixa a interpretação do dado a ser realizada completamente para a aplicação que o está lendo. Resumindo, enquanto em um documento HTML uma tag <p> indica um parágrafo, no XML essa tag pode indicar um preço, um parâmetro, uma pessoa, ou qualquer outra coisa que se possa imaginar (inclusive algo que não tenha nada a ver com um p como por exemplo autores de livros).

Os arquivos XML são arquivos texto, mas não são tão destinados à leitura por um ser humano como o HTML é. Os documentos XML são arquivos texto porque facilitam que os programadores ou desenvolvedores "debuguem" mais facilmente as aplicações, de forma que um simples editor de textos pode ser usado para corrigir um erro em um arquivo XML. Mas as regras de formatação para documentos XML são muito mais rígidas do que para documentos HTML. Uma tag esquecida ou um atributo sem aspas torna o documento inutilizável, enquanto que no HTML isso é tolerado. As especificações oficiais do XML determinam que as aplicações não podem tentar adivinhar o que está errado em um arquivo (no HTML isso acontece), mas sim devem parar de interpretá-lo e reportar o erro.

Características da linguagem XML

O XML provê uma representação estruturada dos dados que mostrou ser amplamentemente implementável e fácil de ser desenvolvida.

Implementações industriais na linguagem SGML (Standard Generalized Markup Language) mostraram a qualidade intrínseca e a força industrial do formato estruturado em árvore dos documentos XML.

O XML é um subconjunto do SGML, o qual é otimizado para distribuição através da web, e é definido pelo Word Wide Web Consortium(W3C), assegurando que os dados estruturados serão uniformes e independentes de aplicações e fornecedores.

O XML provê um padrão que pode codificar o conteúdo, as semânticas e as esquematizações para uma grande variedade de aplicações desde simples até as mais complexas, dentre elas:

· Um simples documento.

· Um registro estruturado tal como uma ordem de compra de produtos.

· Um objeto com métodos e dados como objetos Java ou controles ActiveX.

· Um registro de dados. Um exemplo seria o resultado de uma consulta a bancos de dados.

· Apresentação gráfica, como interface de aplicações de usuário.

· Entidades e tipos de esquema padrões.

· Todos os links entre informações e pessoas na web.

Uma característica importante é que uma vez tendo sido recebido o dado pelo cliente, tal dado pode ser manipulado, editado e visualizado sem a necessidade de reacionar o servidor. Dessa forma, os servidores tem menor sobrecarga, reduzindo a necessidade de computação e reduzindo também a requisição de banda passante para as comunicações entre cliente e servidor.

O XML é considerado de grande importância na Internet e em grandes intranets porque provê a capacidade de interoperação dos computadores por ter um padrão flexível e aberto e independente de dispositivo. As aplicações podem ser construídas e atualizadas mais rapidamente e também permitem múltiplas formas de visualização dos dados estruturados.

Se você já está acostumado com a HTML ou a SGML, os documentos XML parecer-lhe-ão familiar. Um documento XML simples é apresentado a seguir:

Exemplo 1: um documento XML simples

```
<?xml version="1.0"?>

<piada>
```

<João>Diga <citação>boa noite</citação>, Maria.</João>

<José><citação>Boa noite, Maria.</citação></José>

<aplausos/>

</piada>

Algumas coisas podem sobressair-se para você:

O documento começa com uma instrução de processamento: **<?xml ...?>**. Esta é a *declaração XML*. Embora não seja obrigatória, a sua presença explícita identifica o documento como um documento XML e indica a versão da XML com a qual ele foi escrito.

Não há declaração do tipo do documento. Diferentemente da SGML, a XML não requer uma declaração de tipo de documento. Entretanto, uma declaração de tipo de documento pode ser fornecida; além disso, alguns documentos irão precisar de uma para serem entendidos sem ambigüidade.

Elementos vazios (**<aplausos/>** neste exemplo) tem uma sintaxe modificada. Enquanto que a maioria dos elementos em um documentos envolvem algum conteúdo, elementos vazios são simplesmente marcadores onde alguma coisa ocorre (uma separador horizontal para a marca em <hr> em HTML, por exemplo, ou uma referência cruzada para DocBook's para a marca <xref>). O final **/>** na sintaxe modificada indica a um programa que processa o documento XML que o elemento é vazio e uma marca de fim correspondente não deve ser procurada. Visto que os documentos XML não requerem uma declaração de tipo de documento, sem esta pista seria impossível para um analisador XML determinar quais marcas são intencionalmente vazias e quais teriam sido deixadas vazias por um erro.
A XML suavizou a distinção entre elementos declarados como EMPTY e elementos que meramente não têm conteúdo. Em XML, é válido usar uma marca de elemento vazio para qualquer um destes casos. Também é válido usar um par de marcas início-fim para elementos vazios: **<aplausos></aplausos>**. Se a interoperabilidade interessa, é melhor reservar a sintaxe de marcas de elementos vazios para elementos declarados como EMPTY e usar a marca de elemento vazio somente para estes elementos.

Os documento XML são compostos de marcas e conteúdos. Existem seis tipos de marcações que podem ocorrer em um documento XML: elementos, referências a entidades, comentários, instruções de processamento, seções marcadas e declarações de tipos de documento. As seções seguintes introduzem cada um destes conceitos de marcação.

Elementos

Elementos são a mais comum forma de marcação. Delimitados pelos sinais de menor e maior, a maioria dos elementos identificam a natureza do conteúdo que envolvem. Alguns elementos podem ser vazios, como visto acima; neste caso eles não têm conteúdo. Se um elemento não é vazio, ele inicia com uma marca de início , **<element>**, e termina com uma marca de término, **</element>**.

Atributos

Atributos são pares de valores nomeados que ocorrem dentro dasmarcas de início após o nome do elemento. Por exremplo:

** **

é um elemento **div** cujo atributo **class** possui o valor **prefácio**. Em XML, todos os valores de atributos devem estar entre aspas.

Referências a Entidades

A fim de introduzir a marcação em um documento, alguns documentos foram reservados para identificar o início da marcação. O sinal de menor, < , por exemplo, identifica o início de uma marca de inicio ou término. Para inserir estes caracteres em seu documento como conteúdo, deve haver uma alternativa para representá-los. Em XML, entidades são usadas para representar estes caracteres especiais. As entidades também são usadas para referenciar um texto freqüentemente repetido ou alterado e incluí-lo no conteúdo de arquivos externos.

Cada entidade deve ter um nome único. A definição dos seus próprios nomes de entidades é discutido na seção declarações de entidades. Para usar uma entidade, você simplesmente a referencia pelo nome. As referências às entidades iniciam com o E comercial e terminam com um ponto-e-vírgula.

Por exemplo, a entidade **lt** insere um literal < em um documento. A cadeia de caracteres **<element>** pode ser representada em um documento XML como **<<element>**.

Uma forma especial de referência a entidades, chamada de referência a caracter, pode ser usada para inserir arbitrariamente caracteres Unicode em seu documento. Este é um mecanismo para inserir caracteres que não podem ser diretamente digitados pelo seu teclado.

Referências a caracter podem ter uma das duas formas: referências decimais, ℞, e referências hexadecimais, ℞. Ambas se referem ao caracter Unicode número U+211E.

Comentários

Comentários iniciam com **<!--** e terminam com **-->**. Os comentários podem conter qualquer dado, exceto a literal **"--**". Você pode colocar comentários entre marcas em qualquer lugar em seu documento.

Comentários não fazem parte de um conteúdo textual de um documento XML. Um processador XML não é preciso para reconhecê-los na aplicação.

Instruções de Processamento

Instruções de processamento (PIs) são formas de fornecer informações a uma aplicação. Assim como os comentários, elas não são textualmente parte de um documento XML, mas o processador XML é necessário para reconhecê-las na aplicação.

As instruções de processamento têm a forma: **<?nome dadospi?>**. O nome, chamado de alvo PI, identifica a PI na aplicação. As aplicações processariam somente os alvos que eles reconhecem e ignoram todas as outras PIs. Qualquer dado que segue oalvo PI é opcional; é para a aplicação que reconhece o alvo. Os nomes usados em PIs podem ser declarados como notações a fim de identificá-los formalmente.

Os nomes de PI que iniciam com **xml** são reservados para a padronização da XML.

Seções CDATA

Em um documento, uma seção CDATA instrui o analisador para ignorar a maioria dos caracteres de marcação.

Considere um código-fonte em um documento XML. Ele pode conter caracteres que o analisador XML iria normalmente reconhecer como marcação (< e &, por exemplo). Para prevenir isto, uma seção CDATA pode ser usada.

```
<![CDATA[

*p = &q;
b = (i <= 3);

]]>
```

Entre o início da seção, **<![CDATA[**, e o fim da seção, **]]>**, todos os dados de caracteres são passados diretamente para a aplicação, sem interpretação. Elementos, referências a entidades, comentários e instruções de processamento são todos irreconhecíveis e os caracteres que os compõem são passados literalmente para a aplicação.

A única cadeia de caracteres que não pode ocorrer em uma seção CDATA é **"]]>"**.

Declarações de Tipos de Documentos

Uma grande porcentagem da especificação da XML trata de vários tipos de declarações que são permitidas em XML. Se você tem experiência com SGML, você reconhecerá estas declarações dos SGML DTDs (Definições de Tipos de Documentos). Se você já viu isto antes, o seu significado pode não ser imediatamente óbvio.

Um dos maiores poderes da XML é que ela permite que você crie seus próprios nomes para marcas. Mas, para uma dada aplicação, é provável não ser significativo para marcas que ocorrer em uma ordem completamente arbitrária. Considere o exemplo 1. Isto teria significado?

<João>Diga <citação>boa noite</citação>, Maria.</João>

<José><citação>Boa noite, Maria.</citação></José>

Sai totalmente do que normalmente esperamos que tenha senso. Simplesmente não *significa* nada.

Entretanto, de um ponto de vista estritamente sintático, não há nada de errado com este documento XML. Assim, se o documento deve ter significado, e certamente há se você estiver escrevendo uma folha de estilos ou aplicação para processá-lo, deve haver alguma restrição na seqüência e aninhamento das marcas. Declarações são onde estas restrições podem ser expressadas.

Mais genericamente, as declarações permitem a um documento comunicar meta-informações ao analisados a respeito do seu conteúdo. Meta-informação inclui as seqüências e aninhamentos permitidos para as marcas, valores de atributos e seus tipos e padrões, os nomes de arquivos externos que podem ser refernciados e se eles podem ou não conter XML, o formato de algum dado (não-XML) externo que pode ser referenciado e as entidades que podem ser encontradas.

Há quatro tipos de declarações em XML: declarações de tipos de elementos, declarações de listas de atributos, declarações de entidades e declarações de notações.

Declarações de Tipos de Elementos

Declarações de tipos de elementos identificam os nomes dos elementos e a natureza do seu conteúdo. Uma declaração de tipo de elemento típica se parece com isto:

<!ELEMENT piada (João+, José, aplausos?)>

Esta declaração identifica o elemento nomeado como **piada**. Seu *modelo de conteúdo* segue o nome do elemento. O modelo de conteúdo define o que um elemento pode conter. Neste caso, uma **piada** deve conter **João** e **José** e pode conter **aplausos**. As vírgulas entre os nomes dos elementos indicam que eles devem ocorrer em sucessão. O sinal de adição após **João** indica que ele pode ser repetido mais de uma vez, mas deve ocorrer pelo menos uma vez. O ponto de interrogação após **aplausos** indica que ele é opcional (pode estar ausente ou ocorrer somente uma vez). Um nome sem pontuação, como **José**, deve ocorrer somente uma vez.

As declarações para todos os elementos usados em qualquer modelo de conteúdo deve estar presente para que um processador XML verifique a validade do documento.

Além dos nomes de elementos, o símbolo especial **#PCDATA** é reservado para indicar dados de caracter. A cláusula **PCDATA** significa dado de caracter analisável.

Os elementos que contêm somente outros elementos são ditos que têm *conteúdo de elementos*. Os elementos que contêm outros elementos e **#PCDATA** são ditos que têm *conteúdo misturado*.

Por exemplo, a definição para **José** pode ser

<!ELEMENT José (#PCDATA | citação)*>

A barra vertical indica um relacionamento "ou" e o asterisco indica que o conteúdo é opcional (pode ocorrer zero ou mais vezes); por esta definição, portanto, **José** pode conter zero ou mais caracteres e marcas **citação**, misturadas em qualquer ordem. Todos os modelos de conteúdo misturado devem ter esta forma: **#PCDATA** deve vir primeiro, todos os elementos devem ser separados por barras verticais e o grupo inteiro deve ser opcional.

Outros dois modelos de conteúdo são possíveis: **EMPTY** indica que o elemento não possui conteúdo (e, conseqüentemente, não tem marca de término) e **ANY** indica que *qualquer* conteúdo é permitido. O modelo de conteúdo **ANY** é algumas vezes útil durante a conversão de documentos, mas deveria ser evitado ao máximo em um ambiente de produção, pois desabilita toda a verificação do conteúdo deste elemento.

Aqui está um conjunto completo das declarações de elementos para o

Exemplo 1:

Exemplo 2: declarações de elementos para Exemplo 1

<!ELEMENT piada (João+, José, aplausos?)>

<!ELEMENT João (#PCDATA | citação)*>

<!ELEMENT José (#PCDATA | citação)*>

<!ELEMENT citação (#PCDATA)*>

<!ELEMENT aplausos EMPTY>

Declarações de Listas de Atributos

Declarações de listas de atributos identificam que elementos podem ter atributos, que atributos eles podem ter, que valores os atributos podem suportar e qual valor é o padrão. Uma dclaração de lista de atributos típica se parece com isto:

<!ATTLIST piada

nome
ID
#REQUIRED

rótulo
CDATA
#IMPLIED

estado (engraçada | nãoengraçada) 'engraçada'>

Neste exemplo, o elemento **piada** possui três atributos: **nome**, que é um ID e é obrigatório; **rótulo**, que é uma cadeia de caracteres (dados de caracter) e não é obrigatório; e **estado**, que deve ser ou **engraçada** ou **nãoengraçada** e por padrão é **engraçada**, se nenhum valor é especificado.

Cada atributo em uma declaração tem três partes: um nome, um tipo e um valor padrão.

Você tem liberdade para selecionar qualquer nome desejado, sujeito a algumas pequenas restrições, mas os nomes não podem ser repetidos no mesmo elemento.

Existem seis tipos de atributos possíveis:

1. **CDATA**
   Atributos CDATA são cadeias de caracteres; qualquer texto é permitido. Não confunda atributos CDATA com seções CDATA; eles não têm relação.
2. **ID**
   O valor de um atributo ID deve ser um nome. Todos os valores usados para IDs em um documento devem ser diferentes. Os IDs identificam unicamente elementos individuais em um documento. Os elementos podem ter um único atributo ID.
3. **IDREF**
   ou **IDREFS**
   O valor de um atributo IDREF deve ser o valor de um único atributo ID em algum elemento no documento. O valor de um atributo IDREFS pode conter valores IDREF múltiplos separados por espaços em branco.
4. **ENTITY**
   ou **ENTITIES**
   O valor de um atributo ENTITY deve ser o nome de uma única entidade (veja sobre declarações de entidades abaixo). O valor de um atributo ENTITIES pode conter valores de entidades múltiplos separados por espaços em branco.
5. **NMTOKEN**
   ou **NMTOKENS**

Atributos de símbolos de nome são uma forma restrita do atributo de cadeia de caracteres. Em geral, um atributo NMTOKEN deve consistir de uma única palavra, mas não há restrições adicionais para a palavra; não tem que estar associado com outro atributo ou declaração. O valor de um atributo NMTOKENS pode conter valores NMTOKEN múltiplos separados por espaços em branco.

6. Uma lista de nomes

Você pode especificar que o valor de um atributo deve ser pego de uma lista específica de nomes. Isto é freqüentemente chamado de tipo enumerado, porque cada um dos valores possíveis está explicitamente enumerado na declaração.

Alternativamente, você pode especificar que os nomes devem atender a um nome de notação (veja sobre declarações de notação abaixo).

Há quatro valores padrão possíveis:

#REQUIRED

O atributo deve ter um valor explicitamente especificado em cada ocorrência do elemento no documento.

#IMPLIED

O valor do atributo não é requerido, e nenhum valor padrão é fornecido. Se um valor não é especificado, o processador XML deve proceder sem um.

"valor"

Qualquer valor válido pode ser dado a um atributo como padrão. O valor do atributo não é requerido em cada elemento no documento, e se ele estiver presente, será dado a ele o valor padrão.

#FIXED
"value"

Uma declaração de atributo pode especificar que um atributo tem um valor fixo. Neste caso, o atributo não é requerido, mas se ele ocorrer deve ter o valor especificado. Se não estiver presente, será dado a ele o valor padrão. Um uso de atributos fixos é para associar semântica e um elemento. Uma discussão completa vai além do propósito deste trabalho, mas você pode achar vários exemplos de atributos fixos na especificação de XLink.

O processador XML executa a *normalização dos valores dos atributos* nos valores dos atributos: as referências de caracter são substuituídas por caracteres referenciados, referências a entidades são resolvidas (recursivamente) e espaços em branco são normalizados.

Declarações de Entidades

Declarações de entidades lhe permitem associar um nome com algum outro fragmento de conteúdo. Essa construção pode ser um pedaço de texto normal, um pedaço de uma declaração de tipo de documento ou uma referência a um arquivo externo que contém ou texto ou dados binários.

Declarações de entidades típicas são mostradas no Exemplo 3.

Exemplo 3: declaração de entidades típica

```
<!ENTITY
ATI
"ArborText, Inc.">

<!ENTITY boilerplate    SYSTEM
"/standard/legalnotice.xml">

<!ENTITY ATIlogo
SYSTEM "/standard/logo.gif" NDATA GIF87A>
```

Existem três tipos de entidades:

Entidades Internas

Entidades internas associam um nome com uma cadeia de caracteres ou texto literal. A primeira entidade no Exemplo 3 é uma entidade interna. Usando **&ATI;** em qualquer lugar do documento inserirá "ArborText, Inc" naquele local. Entidades internas permitem a você definir atalhos para textos freqüentemente digitados ou textos que se espera que sejam alterados, como o estado de revisão de um documento.

Entidades internas podem incluir referências para outras entidades internas, mas é errado elas serem recursivas.

A especificação XML pré-define cinco entidades internas:

< produz o sinal de menor, <

> produz o sinal de maior, >

& produz o E comercial, &

' produz um apóstrofo, '

" produz aspas, "

Entidades Externas

Entidades externas associam um nome com o conteúdo de um outro arquivo. Entidades externas permitem a documento XML referenciar o conteúdo de um outro arquivo; elas contém ou texto ou dados binários. Se elas contém texto, o conteúdo do arquivo externo é inserido no ponto de referência e analisado como parte do documento referente. Dados binários não são analisados e podem somente ser referenciados em um atributo; eles são usados para referenciar figuras e outro conteúdo não-XML no documento.

A segunda e a terceira entidades no Exemplo 3 são entidades externas.

O uso de **&boilerplate;** inserirá o *conteúdo* do arquivo **/standard/legalnotice.xml** no local da referência da entidade. O processador XML analisará o conteúdo deste arquivo como se ele ocorresse literalmente no local.

A entidade **ATIlogo** também é uma entidade externa, mas o seu conteúdo é binário. A entidade **ATIlogo** pode ser usada somente como o valor de um atributo ENTITY (ou ENTITIES) (em um elemento **graphic**, talvez). O

processador XML passará esta informação para a aplicação, mas ele não tenta processar o conteúdo de **/standard/logo.gif**.

Entidades Parâmetro

A entidade parâmetro somente pode ocorrer na declaração de tipo de documento. Uma declaração de uma entidade parâmetro é identificada por "**%** " (porcento e espaço) defronte ao seu nome na declaração. O sinal de porcento também é usado em referências para entidades parâmetro, ao invés do E comercial. As referências a entidade parâmetro são imediatamente expandidas na declaração de tipo de documento e seu texto de substituição é parte da declaração, onde as referências a entidades normais não são expandidas. Entidades parâmetro não são reconhecidas no corpo de um documento.

Voltando às declarações de elementos no Exemplo 2, você perceberá que dois deles têm o mesmo modelo de conteúdo:

```
<!ELEMENT João (#PCDATA | citação)*>

<!ELEMENT José (#PCDATA | citação)*>
```

Até o momento, estes dois elementos são a mesma coisa somente porque eles têm a mesma definição literal. A fim de tornar mais explícito o fato de que estes dois elementos são semânticamente a mesma coisa, é usada uma entidade parâmetro para definir seus modelos de conteúdo. Há duas vantagens em se usar uma entidade parâmetro. Primeiramente, ela lhe permite dar um nome descritivo ao conteúdo, e segundo que lhe permite alterar o modelo de conteúdo em somente um local, se você desejar atualizar as declarações do elemento, garantindo que elas sempre fiquem as mesmas:

```
<!ENTITY % pessoascontentes "#PCDATA | citação">

<!ELEMENT João (%pessoascontentes;)*>

<!ELEMENT José (%pessoascontentes;)*>
```

Declarações de Notação

Declarações de notação identificam tipos específicos de dados binários externos. Estas informações são passadas para a aplicação de processamento, que pode fazer o uso que quiser ou que desejar. Uma declaração de notação típica é:

```
<!NOTATION GIF87A SYSTEM "GIF">
```

Eu preciso de uma Declaração de Tipo de Documento?

Como foi visto, o conteúdo XML pode ser processado sem uma declaração de tipo de documento. Entretanto, existem alguns casos onde a declaração é necessária:

Ambientes de autoria

A maioria dos ambientes de autoria precisa ler e processar declarações de tipo de documento a fim de entender e reforçar o modelo de conteúdo do documento.

Valores padrões de atributos

Se um documento XML conta com valores padrões de atributos, pelo menos uma parte da declaração deve ser processada a fim de se obter os valores padrões corretos.

Manipulação de espaços em branco

A semântica associada com espaço em branco em conteúdo de elementos diferem da semântics associada com espaço em branco em conteúdo misturado. Sem um DTD, não há maneira para o processador distingüir os casos, e todos os elementos são efetivamente conteúdo misturado. Para mais detalhes, veja a seção chamada Manipulação de Espaços em Branco, neste trabalho.

Em aplicações onde uma pessoa compõe ou edita os dados, um DTD provavelmente vai ser preciso se qualquer estrutura deve ser garantida.

Incluindo uma Declaração de Tipo de Documento

Se presente, a declaração de tipo de documento deve ser a primeira coisa em um documento depois de comentários e instruções de processamento opcionais.

A declaração de tipo de documento identifica o elemento raiz do documento e pode conter declarações adicionais. Todos os documentos XML devem ter um elemento raiz único que contenha todo o conteúdo do documento. Declarações adicionais podem vir de um DTD externo, chamado de subconjunto externo, ou ser incluído diretamente no documento, o subconjunto interno, ou ambos:

```
<?XML version="1.0" standalone="no"?>

<!DOCTYPE chapter SYSTEM "dbook.dtd" [

<!ENTITY %ulink.module "IGNORE">

<!ELEMENT ulink (#PCDATA)*>

<!ATTLIST ulink

   xml:link      CDATA  #FIXED "SIMPLE"

   xml-attributes CDATA  #FIXED "HREF URL"

   URL           CDATA  #REQUIRED>

]>

<chapter>...</chapter>
```

Este exemplo referencia um DTD externo, **dbook.dtd**, e inclui declarações de elementos e atributos para o elemento **ulink** no subconjunto interno. Neste caso, ulink dá a semântica de um link simples da especificação XLink.

Note que as declarações no subconjunto interno não leva em conta as declarações no subconjunto externo. O processador XML lê o subconjunto interno antes do externo e a *primeira* declaração tem precedência.

A fim de determinar se um documento é válido, o processador XML deve ler a declaração de tipo de documento inteira (ambos os subconjuntos). Mas para algumas aplicações, a validação pode não ser precisa, e pode ser suficiente para o processador ler somente o subconjunto interno. No exemplo acima, se a validade não é importante e a única razão para ler a declaração de tipo de documento é identifcar a semântica de **ulink**, a leitura do subconjunto externo não é necessária.

Você pode comunicar estas informações na *declaração de documento standalone*. A declaração de documento standalone, **standalone="yes"** ou **standalone="no"**, ocorre na declaração XML. Um valor **yes** indica que somente declarações internas precisam ser processadas. Um valor **no** indica que *ambas* as declarações interna e externa devem ser processadas.

Outras questões de marcação

Além da marcação, existem algumas outras questões a considerar: manipulação de espaços em branco, normalização de valores dos atributos e a linguagem com a qual o documento foi escrito.

Manipulação de Espaços em Branco

A manipulação de espaços em brancos é uma questão sutil. Considere o seguinte fragmento de conteúdo:

<piada>

<João>Diga <citação>boa noite</citação>, Maria.</João>

O espaço em branco (a nova linha entre **<piada>** e **<João>**) é significante?

Provavelmente não.

Mas como você pode afirmar isto? Você somente pode determinar se um espaço em branco é significante se você conhece o modelo de conteúdo dos elementos em questão. Em resumo, um espaço em branco é significante em conteúdo misturado e insignificante em conteúdo de elemento.

A regra para os processadores XML é que eles devem passar por todos os caracteres que não são marcação na aplicação. Se o processador é um processador de validação, ele também deve informar à aplicação se os caracteres espaços em branco são significantes.

O atributo especial **xml:space** pode ser usado para indicar explicitamente que os espaços em branco são significantes. Em qualquer elemento que inclua a especificação de atributo **xml:space='preserve'**, todos os espaços em branco naquele elemento (e dentro dos subelementos que não alteram explicitamente **xml:space**) serão significantes.

Os únicos valores válidos para **xml:space** são **preserve** e **default**. O valor **default** indica que o processamento padrão é desejado. Em um DTD, o atributo **xml:space** deve ser declarado como um tipo enumerado com somente estes dois valores.

Uma última observação sobre espaços em branco: em texto analisável, os processadores XML são requeridos para normalizar todas as marcas de

final de linha para um um único caracter de alimentação de linha (&#A;). Isto raramente é de interesse dos autores, mas elimina um número de questões de portabilidade de plataformas cruzadas.

Normalização dos valores de atributos

O processador XML executa a normalização dos valores de atributos em valores de atributos: referências a caracteres são substituídas por caracteres referenciados, referências a entidades sã resolvidas (recursivamente) e os espaços em branco são normalizados.

Identificação da linguagem

Muitas aplicações de processamento de documentos podem se beneficiar da informação sobre a linguagem natural com a qual o documento foi escrito. A XML define o atributo **xml:lang** para identificar a linguagem. Visto que o propósito deste atributo é padronizar a informação entre as aplicações, a especificação XML também descreve como as linguagens devem ser identificadas.

Validação

Dada a discussão precedente de declarações de tipos, conclui-se que uns documentos são válidos e outros não. Existem duas categorias de documentos XML: bem formatados e válidos.

Documentos Bem Formatados

Um documento somente pode ser bem formatado se ele obedece a sintaxe da XML. Um documento que inclui seqüências de caracteres de marcação que não podem ser analisadas ou são inválidas não podem ser bem formatados.

Além disso, o documento deve atender a todas as seguintes condições (subentendendo-se que algumas destas condições podem exigir experiência com SGML):

A instância do documento deve estar conforme a gramática dos documentos XML. Em particular, algumas construções de marcações (referências a entidades parâmetro, por exemplo) são somente permitidas em locais específicos. O documento não é bem formatadose tais ocorrerem em outros locais, ainda que o documento esteja bem formatado nos outros casos.

O texto de substituição para todas as entidades parâmetro referenciadas dentro de uma declaração de marcação consiste em zero ou mais declarações de marcações completas. (Nenhuma entidade usada no documento pode consistir de somente uma parte de uma declaração de marcação.)

Nenhum atributo pode aparecer mais do que uma vez na mesma marca de início.

Valores de atributos cadeias de caracteres não podem conter referências a entidades externas.

Marcas não-vazias devem ser apropriadamente aninhadas.

Entidades parâmetro devem ser declaradas antes de serem usadas.

Todas as entidades devem ser declaradas, exceto as seguintes: **amp**, **lt**, **gt**, **apos** e **quot**.

Uma entidade binária não pode ser referenciada no fluxo do conteúdo; ela pode ser usada somente em um atributo declarado como **ENTITY** ou **ENTITIES**.

Nem a texto ou entidades parâmetro são permitidas recursividade, direta ou indiretamente.

Por definição, se um documento não está bem formatado, ele não é XML. Isto significa que não há documento XML que não seja bem formatado e os processadores XML não fazem nada com tais documentos.

Documentos

Um documento bem formatado é válido somente se ele contém uma declaração de tipo de documento e se o documento obedece as restrições da declaração (seqüência e aninhamento de elementos é válido, atributos necessários são fornecidos, valores de atributos são do tipo correto, etc.). A especificação XML identifica todos os creitérios em detalhes.

XML e Java

A WWW (*Word Wide Web*) tem a finalidade de permitir aos usuários o acesso a diversos tipos de documentos eletrônicos e dados. Para a formatação e exibição desses documentos foi definido como um padrão, a linguagem de marcação HTML, que permite que as páginas possam ser reconhecidas por um *browser*. Com o grande crescimento da utilização da Internet e uso dos *Web Services,* a linguagem HTML não estava mais suprindo necessidades, deixando a desejar em diversos aspectos, quanto à troca de informações e dados via Internet, por esse motivo, se viu a necessidade de uma linguagem para WWW, que viesse a suprir esta demanda. Sendo assim um grupo de empresas denominado *Word Wide Web Consortium* (W3C), que é um consorcio de varias empresas que em 1996 deram origem ao projeto de uma nova linguagem mais simples, com um padrão bem definido, para ser usada na WWW, surgindo assim a linguagens de marcação XML (*eXtensible Markup Language*) Linguagem de Marcação Extensível que é um subconjunto do SGML(*Standart Generalized Markup Language*) Linguagem de marcação Generalizada. A XML permite que se troquem informações e dados através da WEB, servindo também para uma grande variedade de aplicações.

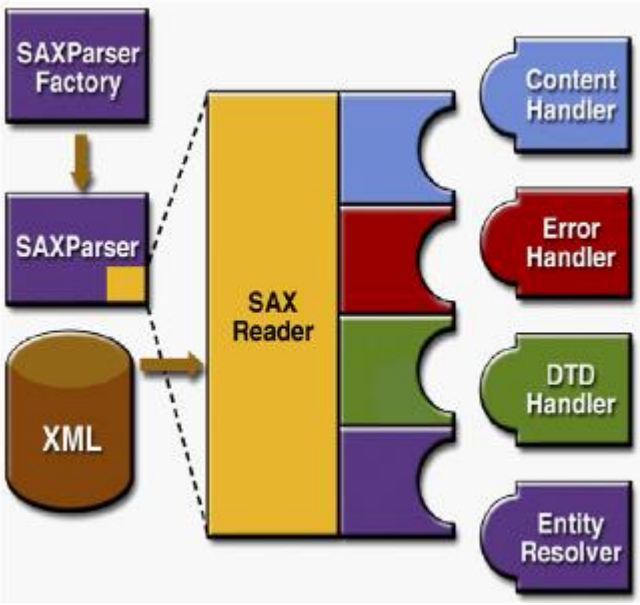
Para se ter o domínio da linguagem XML é necessário estar familiarizado com diversos acrônimos como: XML, XSL, XPath, RDF, XML *Schema*, DTD, PI, XSLT, XSP, JAXP™, SAX, DOM, e mais. E não existe um gerente de desenvolvimento nas grandes empresas, que não queira que sua equipe aprenda XML. A linguagem XML promete trazer a formatação de dados, o que a linguagem Java trouxe para a programação na WEB: portabilidade completa tanto de sistema quanto de plataforma. De fato, somente com XML é que a promessa de Java pode ser plenamente percebida: a portabilidade completa que foi assumida como um compromisso sério, para formatos de dados proprietários, que vinham sendo usados durante anos, permitindo assim que uma aplicação venha ser executada em plataformas múltiplas, mas não de maneira separada mas sim de um modo unificado. XML promete preencher a lacuna que existia, possibilitando interoperabilidade completa para programas de Java, removendo os dados em formato proprietário e permitindo aos sistemas que se comuniquem usando um padrão por meios de representação de dados.

APIS PARA JAVA/XML:

Java é freqüentemente a primeira linguagem para implementar novas tecnologias relacionadas a XML. Isto é em grande parte devido à natureza do funcionamento e da independência de plataforma, código (Java) e dados (XML). Porém, o desenvolvimento de APIs em Java tem sido deslocado, ficando por demais complicado. Vários grupos implementaram funcionalidade de XML em Java de modos diferentes, e em diferente épocas o que conduziu à proliferação se sobrepondo e gerando incompatibilidades entre as APIs.

Em relação a este assunto a *Sun Microsystems* coordena um movimento entre os desenvolvedores Java, com o intuito de unificar e simplificar várias APIs Java para XML. O Java APIs para XML (JAX) é atualmente uma família de especificações de API relacionadas.

Java e SAX



A API SAX cresceu pelo motivo do método DOM ser extremamente complexo, tornando-se inadequado para varias aplicações. SAX faz a analise gramatical de um documento XML em seqüência, gerando e lançando eventos para a capa do processo ao encontrar elementos XML diferentes. O modelo de SAX permite *parsers* simples permitindo *parsers* para ler do princípio ao fim um documento de um modo seqüencial e então chamar um manipulador de evento toda vez um evento de marcacão acontece. Esta aproximação seqüencial habilita rapidamente o *parsing* de dados XML, especialmente no caso de documentos XML longos ou complexos, porem a desvantagem é que o *parser* de SAX não pode ser usado para ter acesso aos nos de um documento XML de uma maneira aleatória ou não seqüencial.

Para começar o processo de analise uma instância do *SaxParserFactory* é usado para gerar uma instância do *parser*.O *parser* encapsula um objeto *SAXReader*. Quando o *parser* faz a analise gramatical o método é invocado, o leitor invoca um dos vários métodos de *callback* implementado na aplicação. Esses métodos são definidos pelas interfaces *ContentHandler*, *ErrorHandler, DTDHandler*, e *EntityResolver*.

Abaixo é um resumo das chaves SAX APIs :

SAXParserFactory

Um objeto de S*AXParserFactory* cria uma instância do *parser* determinada pela propriedade de sistema, j*avax.xml.parsers.SAXParserFactory*.

SAXParser

A interface de *SAXParser* define vários tipos de métodos para analise gramatical.

Geralmente é passado dados fontes XML e um objeto de *DefaultHandler* para o *parser* que processa o XML e invoca os métodos apropriados no objeto de manipulação.

SAXReader

O *SAXParser* encapsula um *SAXReader*. Tipicamente, o programador não se preocupa com isso, mas de vez em quando é precisa adquirir uma conexão, que usa o *getXMLReader* de *SAXParser* (), geralmente isto pode ser configurado. É o *SAXReader* que mantêm a conversação com os eventos SAX manipuladores definidos pelo programador.

DefaultHandler

Um *DefaultHandler* implementa o *ContentHandler, ErrorHandler, DTDHandler*, e *EntityResolver* conecta (com métodos nulos), assim pode-se anular só aquilo que interessar.

ContentHandler

Métodos como *startDocument*, invocam *endDocument, startElement*, e *endElement* quando uma *tag* de XML é reconhecida. Esta interface também define caráter de métodos e *processingInstruction* que são invocados quando o *parser* encontra o texto em um elemento de XML ou um em linha que processa instrução, respectivamente.

ErrorHandler

Erro de métodos, *fatalError*, estes métodos são invocados em resposta a vários erros analisados gramaticalmente. O manipulador de erro, lança uma exceção para erros fatais e ignora outros erros (inclusive erros de validação). Isso é uma razão que você precisa saber algo sobre o *parser* de SAX, até mesmo se você está usando o DOM. Às vezes, a aplicação pode poder recuperar de um erro de validação. Outras vezes, pode precisar gerar uma exceção. Para assegurar a manipulação correta, é necessário prover seu próprio manipulador de erro para o *parser*.

DTDHandler

Define métodos utilizados pelo *parser* para relatar notações e entidades definidas no DTD.

EntityResolver

O método de *resolveEntity* é invocado quando o *parser* têm que identificar dados identificados por um URI. Em a maioria dos casos, um URI é simplesmente um URL que especifica a localização de um documento, mas em alguns casos o documento pode ser identificado por uma URNA um identificador público, ou nomeia, isso é sem igual no espaço de rede. O identificador público pode ser especificado além do URL. O *EntityResolver* pode usar o identificador público, então em vez do URL achar o documento, por exemplo, para ter acesso uma cópia local do documento se a pessoa existe.

Uma aplicação típica implementa a maioria dos métodos de *ContentHandler*. Como a implementação de falta das interfaces ignora tudo que é introduzido com exceção de erros fatais.

Os Pacotes de SAX

O *parser* de SAX é definido nos pacotes seguintes:

Pacote	Descrição
org.xml.sax	Defines as interfaces de SAX. O nome " org.xml " é o prefixo de pacote que foi criado grupo que definiu o SAX API.
org.xml.sax.ext	Defines extensões de SAX que são usadas quando se faz um processamento SAX mais sofisticado, por exemplo, para processar umas definições de tipo de documento (DTD) ou ver a sintaxe detalhada para um arquivo.
org.xml.sax.helpers	Contêm classes de ajuda que tornam mais fácil de usar SAX, por exemplo, definindo um manipulador de falhas que tem métodos nulos tudo das interfaces, assim você só precisa anular aquilo de fato que você quer implementar.
javax.xml.*parsers*	Defines o SAX*Parser*Factory que classificam e devolvem o SAX*Parser*. Também define classificação de exceção para informar erros.

TABELA 1.2 - PACOTES API SAX

Resumo:

Define uma API dirigida por evento para leitura de documentos XML.

Utiliza a noção de desenvolvimento de métodos de call-back.

Oferece boa performance para leitura do dado, porém não temos flexibilidade de navegação e alteração de dados.

Necessita de menos recurso que DOM pois não monta representação do dado em memória.

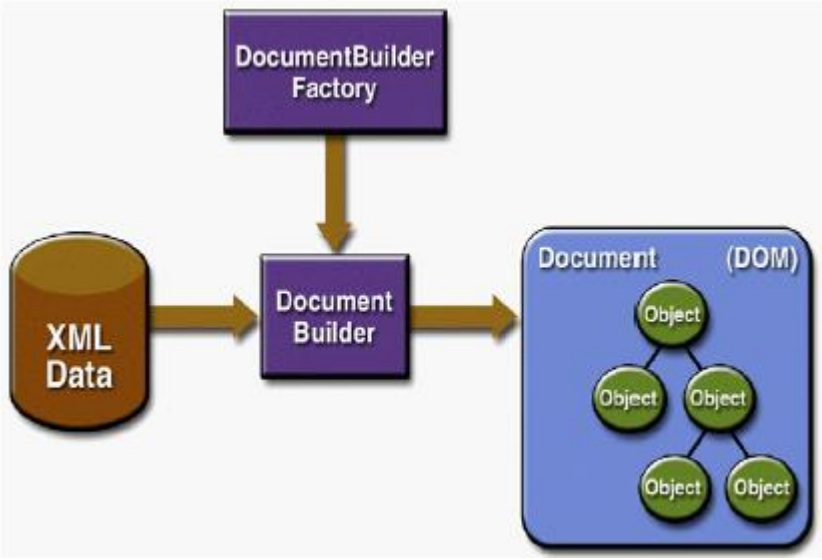
Utilize SAX sempre que necessitar de performance ou então quando necessita somente de operações e documentos simples.

Exemplo de Leitura com SAX: sampleSAX.java

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;

public class sampleSAX extends DefaultHandler {
public void startElement(String uri, String localName, String qName,
Attributes attributes) {
System.out.println("startElement - " + qName);
System.out.println(uri + localName + qName);

}
public void startDocument() {
System.out.println("startDocument");
}

public void endDocument() {
System.out.println("endDocument");
}

public void endElement(String uri, String localName, String qName) {
System.out.println("endElement - " + qName);
}

public static void main(String args[]) throws Exception {
//out = new OutputStreamWriter(System.out, "UTF8");
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
saxParser.parse(args[0], new sampleSAX());
```

```
}
}
```

Java e o XML DOM (Document Object Model)



Uma vês que um programador de Java tem um documento XML, ele precisa ter acesso às informações contidas neles usando provavelmente um código Java. Um DOM *parser* de XML é um programa de Java que converte os documentos XML em um modelo de Java Objeto. Uma vez analisado gramaticalmente um documento de XML, existe na memória uma Máquina Virtual Java como um grupo de objetos. Quando for necessário ter acesso ou modificar informações armazenadas no documento XML, não é necessário manipular o documento XML em arquivos diferentes, ao contrario é preciso ter acesso modificando a informação por estes objetos na memória Assim o *parser* DOM cria a representação de um documento Java do arquivo do documento XML.

O *parser* também executa algum texto simples que processa e cria um modelo de objeto do documento XML, ampliando todas as entidades usadas no arquivo e comparando a estrutura da informação no documento para um DTD. Tendo êxito o processo, o *parser* cria uma representação de objetos do documento XML. Para que o programa Java tenha acesso a informação no documento XML,, o *parser* têm que ler os arquivos de XML do cabeçalho. O *parser* processa o arquivo que foi lido conferindo a informação contida nele para validar e ampliando todas as entidades usadas neste arquivo. Então o documento XML processado é convertido em um objeto na memória pelo *parser* XML. Este objeto de documento contém uma árvores de nós que possuem os dados e estrutura da informação contida neste documento de XML. Esta árvore de nós pode ser acessada e modificada usando o DOM API. Todos os nós começam conseqüentemente de um nó raiz que é chamado um objeto de documento ou DOM.

JDOM é um modelo de documento Java especifico que interage com XML, mais simples e mais rápido que a implementação DOM. JDOM difere de DOM em dois assuntos principais. Primeiro o uso de JDOM só se solidificam classes em lugar de interface. Isto simplifica a API em um tamanho razoável, mas também limita sua flexibilidade.Segundo, a API faz uso extenso das coleções classificando e simplificando para o uso do programador de Java que já estão familiarizados com essas classes.

Conforme Dennis M. Sosnoski na documentação de JDOM é declarado como meta "resolva 80% (ou mais) de problemas de Java/XML com 20% (ou menos) do esforço" (com os 20% se referindo presumivelmente à curva de aprendizagem). JDOM é certamente utilizável para a maioria de aplicações de Java/XML, e a maioria dos programadores tem o API significativamente mais fácil de entender que DOM. JDOM também inclui conferências bastante extensas em comportamento de programa, para impedir que o usuário venha a fazer qualquer coisa que não faz sentido em XML. Porém, ainda é preciso que se tenha um entendimento de XML, para fazer qualquer coisa além dos fundamentos (ou até mesmo entender os erros, em alguns casos). Este provavelmente é um esforço mais significante que aprendendo o DOM ou interface de JDOM.

É usado o javax.xml.*parser*s.*DocumentBuilderFactory* para classificar e adquirir uma instância de *DocumentBuilder*, usando isso para produzir um Documento (um DOM), isso conforme a especificação de DOM. O construtor adquirido é determinado pela propriedade de sistema, javax.xml.*parser*s.*DocumentBuilderFactory* que seleciona a implementação de construção que é usado para produzir o construtor. (O valor de falta da plataforma pode ser anulado da linha de comando).

Pode-se também usar o *newDocumen*t de *DocumentBuilder* (), método para criar um documento vazio que implementa a org.w3c.dom.*Document* interface. Alternativamente, pode-se usar um construtor que analise gramaticalmente métodos para criar um documento de dados de XML existentes.

Os Pacotes de DOM

O Objeto de Documento de implementação Modelo é definido nos pacotes seguintes:

Pacote	Descrição
Org.w3c.dom	Definem o DOM que programa interfaces para documentos XML, como especificado pelos W3C.
javax.xml.*parser*s	Defines o DocumentBuilderFactory e o DocumentBuilder que devolve um objeto que implementa a interface W3C de Documento. O método que é usado para criar o construtor é determinado pelo javax.xml.*parser*s, propriedade do sistema que pode ser fixada da linha de comando ou pode ser anulada quando for invocado o método de newInstance. Este pacote também define o *Parser*ConfigurationException que usado para informar erros.

TABELA 1.1 - PACOTES DA API DOM

DOM disponibiliza um modelo mais flexível montando uma representação do XML em memória.

Exemplo: sampleDOM.java

```
 import javax.xml.parsers.*;
import org.xml.sax.*;
import java.io.*;
import org.w3c.dom.*;

public class sampleDOM {
public static void main(String args[]) throws Exception {
Document document;
DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
factory.setValidating(false);
DocumentBuilder builder = factory.newDocumentBuilder();
document = builder.parse(new File(args[0]));
for(int x=0;x<document.getChildNodes().getLength();x++) {
printNode(document.getChildNodes().item(x));
}
}
static public void printNode(Node node)
{
Node currentNode;
if (node.hasChildNodes())
{
NodeList nlNode = node.getChildNodes();
for (int i=0; i<nlNode.getLength();i++)
{
currentNode = nlNode.item(i);
if (currentNode.getNodeType()==currentNode.ELEMENT_NODE)
{
String strCurrentNode = "<"+currentNode.getNodeName()+">";
System.out.println("START - " + strCurrentNode);
printNode(currentNode);
System.out.println("END - " + strCurrentNode);
}
else if (currentNode.getNodeType()==currentNode.TEXT_NODE) {
System.out.println("-->TEXT NODE = " +
currentNode.getNodeValue());
}
else if
(currentNode.getNodeType()==currentNode.ATTRIBUTE_NODE) {
System.out.println("-->ATTRIBUTE NODE = " +
currentNode.getNodeValue());
}
else if (currentNode.getNodeType()==currentNode.COMMENT_NODE)
{
```

```
System.out.println("-->COMMENT NODE = " +
currentNode.getNodeValue());
}
else if (currentNode.getNodeType()==currentNode.ENTITY_NODE) {
System.out.println("-->ENTITY NODE = " +
currentNode.getNodeValue());
}
else {
System.out.println("-->OUTROS NODE = " +
currentNode.getNodeValue());
}
}
}
}
}
```

```
import java.io.FileOutputStream;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;
public class GeraXml {
private String fileName = "Cliente.xml";
public void geraXml( Cliente cliente ){
// elemento principal do xml
Element Cliente = new Element("Cliente");
//criando os elementos que irão fazer parte do xml
Element logradouro = new Element("logradouro");
Element bairro = new Element("bairro");
Element cidade = new Element("cidade");
Element uf = new Element("uf");
Element cpf = new Element("cpf");
Element rg = new Element("rg");
Element profissao = new Element("profissao");
//atribui os valores para os campos
logradouro.setText( cliente.getLogradouro() );
bairro.setText( cliente.getBairro() );
cidade.setText( cliente.getCidade() );
uf.setText( cliente.getUf() );
cpf.setText( cliente.getCpf() );
rg.setText( cliente.getRg() );
profissao.setText( cliente.getProfissao() );
//atribui os campos para ao elemento principal
Cliente.addContent( logradouro );
Cliente.addContent( bairro );
Cliente.addContent( cidade );
```

```
Cliente.addContent( uf );
Cliente.addContent( cpf );
Cliente.addContent( rg );
Cliente.addContent( profissao );
//criando o documento
Document doc = new Document();
//atribui o elemento principal
doc.setRootElement( Cliente );
//cria o output
XMLOutputter xout = new XMLOutputter( Format.getPrettyFormat() );
try{
xout.output( doc, new FileOutputStream( fileName ) );
}catch (Exception e){
e.printStackTrace();
}
}
}
```

Gerando PDF com Java e XML

Para o processamento é necessária a utilização de uma api chamada FOP (Formatting Objects Processor). Através dela, é possível estar gerando PDF utilizando XML + XSLT.

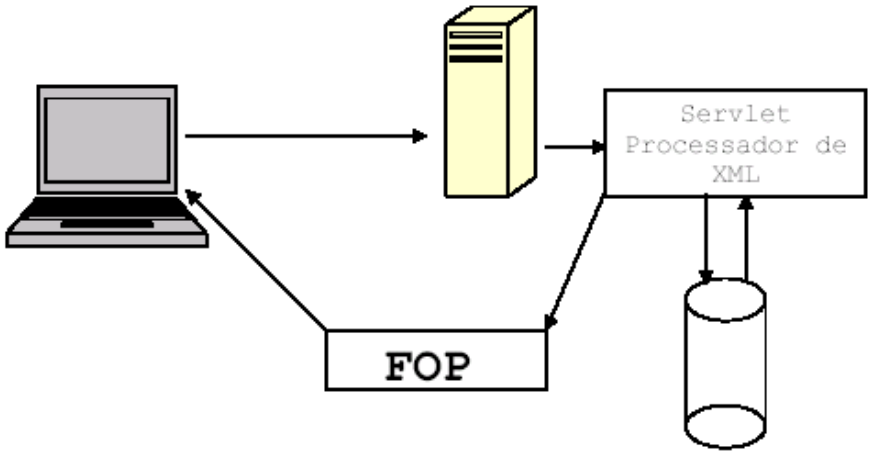
O FOP é um projeto *open source* da Apache e encontra-se no endereço: http://xml.apache.org/fop

Será utilizado também o JDOM: http://www.jdom.org

A especificação XSL-FO pode ser encontrada em: http://www.w3.org/TR/2001/REC-xls-20011015/

Veja abaixo como seria o processamento para a geração de um PDF baseando-se na arquitetura web-centric.

Servidor J2EE



Pré-Requisitos para Arquitetura Web-Centric 8.3:

Servidor WEB (Tomcat, JRun, IPlanet)

JDK 1.2.x ou mais atual.

Download

Para fazer o download acesse o endereço http://xml.apache.org/dist/fop e veja qual é a versão para o seu sistema operacional.

O arquivo fop-0.20.5-bin.zip possui os jar´s necessários, além de muitos exemplos que poderão ser executados.

ExampleObj2PDF.Java

ExampleXML2PDF.Java

ExampleObj2XML.java

ExampleFO2PDF.java

ExampleXML2FO.java

ExampleAWTViewer.java

Configuração

Coloque no /lib os arquivos:

avalon-framework-cvs-20020806.jar fop.jar jdom.jar

Estes arquivos são necessários para a geração do PDF, porém o FOP pode ser utilizado para a criação de outros tipos de arquivos. Os arquivos citados acima são o mínimo necessário para a geração do PDF.

Criando PDF´s

Para a criação do PDF são necessários 2 arquivos, um XML e um XSL. O arquivo XML pode ser dinâmico de acordo com os dados que serão gerados. Porém devemos ter pronto o arquivo XSL, pois é onde temos a formatação do texto a ser gerado no PDF.

Abaixo temos um documento XSLT com o estilo do seu PDF a ser gerado.

Exemplo: Cliente.XSL

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.1" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format" exclude-result-prefixes="fo">
   <xsl:output method="xml" version="1.0" omit-xml-declaration="no"
indent="yes"/>
   <xsl:param name="versionParam" select="'1.0'"/>
   <xsl:template match="Cliente">
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
<fo:layout-master-set>
<fo:simple-page-master master-name="simpleA4" page-height="29.7cm" page-
width="21cm"
margin-top="2cm" margin-bottom="2cm" margin-left="2cm" margin-right="2cm">
<fo:region-before
extent="1.5in"
padding="6pt 1in"
border-bottom="0.5pt"
display-align="after" />
<fo:region-body margin-bottom="20pt" margin-top="80pt"/>
<fo:region-after extent="1in" padding="3pt 1in" border-bottom="0.5pt"
display-align="after" />
</fo:simple-page-master>
</fo:layout-master-set>
<fo:page-sequence master-reference="simpleA4">
<fo:static-content flow-name="xsl-region-before">
<fo:block>
<fo:external-graphic src="pdf.gif" content-height="1em" content-width="1em" />
</fo:block>
</fo:static-content>
<fo:static-content flow-name="xsl-region-after" font-size="8pt" font-
style="normal">
<fo:block line-height="14pt" text-align="start" font-weight="bold">
Locadora DVD Show
</fo:block>
<fo:block line-height="14pt" text-align="start">
```

```
Rua Rocha Pombo, 105 - Vila Independencia
</fo:block>
<fo:block line-height="14pt" text-align="start">
Porto Alegre - RS - CEP: 01234-567
</fo:block>
<fo:block line-height="14pt" text-align="start">
Tel: (11) 8888-8888 . Fax: (11) 8888-8889
</fo:block>
</fo:static-content>
<fo:flow flow-name="xsl-region-body">
<fo:block font-size="14pt" line-height="17pt" text-align="center" font-
weight="bold">
DECLARACAO
</fo:block>
<fo:block font-size="11pt" line-height="17pt" text-align="justify" space-
before="50pt">
Eu,
<xsl:value-of select="nome"/>,
<xsl:value-of select="profissao"/> domiciliado na
<xsl:value-of select="logradouro"/>, no bairro
<xsl:value-of select="bairro"/>, na cidade de
<xsl:value-of select="cidade"/>, estado
<xsl:value-of select="uf"/>, inscrito(a) no CPF / CNPJ
<xsl:value-of select="cpf"/>, portador da cedula de identidade (RG) n.:
<xsl:value-of select="rg"/>, declaro nesta data que:
</fo:block>
<fo:block font-size="11pt" line-height="17pt" text-align="justify" space-
before="20pt">
Recebi da Locadora DVD Show Ltda. todos os valores que me eram devidos,
devidamente corrigidos.
</fo:block>
<fo:block font-size="11pt" line-height="17pt" text-align="justify" space-
before="20pt">
Declaro ainda que, a partir desta data, nada tenho a exigir da empresa
Locadora DVD Show Ltda., civil e ou criminalmente, posto que esta empresa
cumpriu com os termos da legislacao vigente.
</fo:block>
<fo:block font-size="11pt" line-height="17pt" space-before="20pt">
Por ser verdade, firmo a presente em uma unica via.
</fo:block>
<fo:block font-size="11pt" line-height="17pt" space-before="60pt">
_____
</fo:block>
<fo:block font-size="11pt" line-height="17pt">
<xsl:value-of select="nome"/>
</fo:block>
```

```
</fo:flow>
</fo:page-sequence>
</fo:root>
</xsl:template>
</xsl:stylesheet>
```

Depois precisamos criar o documento XML com os dados a serem inseridos no PDF.

Cliente.XML gerada por exemplo anterior

```
<?xml version="1.0" encoding="UTF-8"?>
<Cliente>
  <logradouro>Rua Mauricio de Mendonca, 344</logradouro>
  <bairro>Cohab II</bairro>
  <cidade>Mococa</cidade>
  <uf>SP</uf>
  <cpf>286.694.458-59</cpf>
  <rg>29.890.025-7</rg>
  <profissao>Jogador</profissao>
</Cliente>
```

Uma simples classe para fazer o processamento do XML + XSLT gerando o PDF.

Exemplo: GeraPdf.java

```
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.sax.SAXResult;
import javax.xml.transform.stream.StreamSource;
import org.apache.avalon.framework.logger.ConsoleLogger;
import org.apache.avalon.framework.logger.Logger;
import org.apache.fop.apps.Driver;
public class GeraPdf {
private TransformerFactory transformerFactory;
private String pdfFile = "Cliente.pdf";
public void GerarPdf(){
this.transformerFactory = TransformerFactory.newInstance();
//cria o driver
Driver driver = new Driver();
```

```
Logger logger = new ConsoleLogger(ConsoleLogger.LEVEL_INFO);
driver.setLogger(logger);
//seta o tipo de renderização
driver.setRenderer( Driver.RENDER_PDF );
try{
//Seta o buffer para o output
ByteArrayOutputStream out = new ByteArrayOutputStream();
driver.setOutputStream(out);
//Setup Transformer
String xslFile="Cliente.xsl";
Source xsltSrc = new StreamSource( new File( xslFile ) );
Transformer transformer = this.transformerFactory.newTransformer( xsltSrc ) ;
Result res = new SAXResult( driver.getContentHandler() );
//Setup o source
String xmlFile="Cliente.xml";
Source src = new StreamSource( new File( xmlFile ) );
//Inicia o processo de transformação
transformer.transform(src, res);
FileOutputStream fos = new FileOutputStream( this.pdfFile );
fos.write( out.toByteArray() );
}catch( Exception e) {
e.printStackTrace();
}
}
}
```

Exemplo: Cliente.java

```
public class Cliente {
    private String nome;
    private String logradouro;
    private String bairro;
    private String cidade;
    private String uf;
    private String cpf;
    private String rg;
    private String profissao;
public Cliente(){
    nome="nome";
    logradouro=" logradouro ";
    bairro="bairro";
    cidade="cidade";
    uf="uf";
    cpf="cpf";
    rg="rg";
```

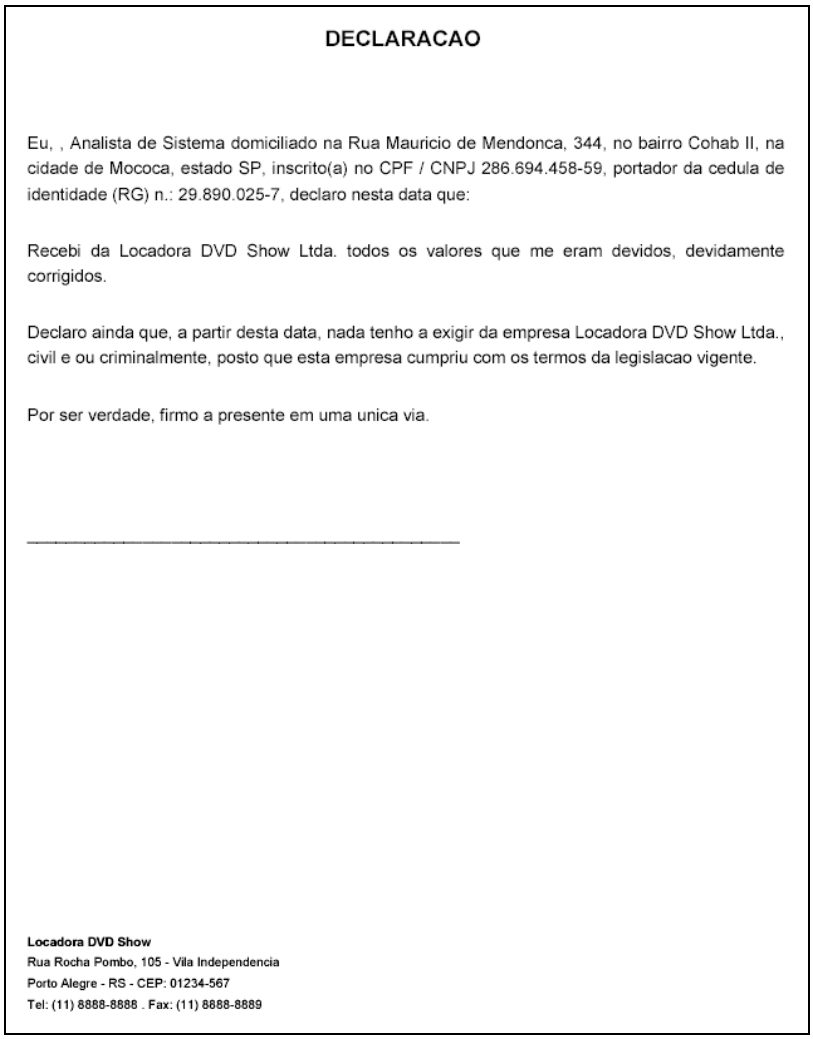
```
    profissao="profissao";
}
    //métodos getters e setters
    public String getNome (){
    return nome;
    }
    public String getLogradouro (){
    return logradouro;
    }
    public String getBairro (){
    return bairro;
    }
    public String getCidade (){
    return cidade;
    }
    public String getUf (){
    return uf;
    }
    public String getCpf (){
    return cpf;
    }
    public String getRg (){
    return rg;
    }
    public String getProfissao (){
    return profissao;
    }
    public void setNome (String local){
     nome=local;
    }
    public void setLogradouro (String local){
     logradouro=local;
    }
    public void setBairro (String local){
     bairro=local;
    }
    public void setCidade (String local){
     cidade=local;
    }
    public void setUf (String local){
     uf=local;
    }
    public void setCpf (String local){
     cpf=local;
    }
    public void setRg (String local){
```

```
    rg=local;
   }
   public void setProfissao (String local){
    profissao=local;
   }
}
```

Exemplo: Application.java

```
public class Application {
public static void main(String[] args) {
Cliente cliente = geraCliente();
GeraXml geraXml = new GeraXml();
//Chama a classe que gera o arquivo xml
geraXml.geraXml( cliente );
GeraPdf geraPdf = new GeraPdf();
//Chama a classe que gera o arquivo pdf
geraPdf.GerarPdf();
System.out.println("Processamento efetuado com sucesso!");
}
/**
* Esta Classe popula um objeto do tipo Cliente para
* ser utilizado como teste.
*/
private static Cliente geraCliente(){
Cliente cliente = new Cliente();
cliente.setNome("Anderson Rodrigues Araujo");
cliente.setProfissao("Jogador");
cliente.setLogradouro("Rua Mauricio de Mendonca, 344");
cliente.setBairro("Cohab II");
cliente.setCidade("Mococa");
cliente.setUf("SP");
cliente.setCpf("286.694.458-59");
cliente.setRg("29.890.025-7");
return cliente;
}
```

Documento PDF gerado 8.7:



DECLARACAO

Eu, , Analista de Sistema domiciliado na Rua Mauricio de Mendonca, 344, no bairro Cohab II, na cidade de Mococa, estado SP, inscrito(a) no CPF / CNPJ 286.694.458-59, portador da cedula de identidade (RG) n.: 29.890.025-7, declaro nesta data que:

Recebi da Locadora DVD Show Ltda. todos os valores que me eram devidos, devidamente corrigidos.

Declaro ainda que, a partir desta data, nada tenho a exigir da empresa Locadora DVD Show Ltda., civil e ou criminalmente, posto que esta empresa cumpriu com os termos da legislacao vigente.

Por ser verdade, firmo a presente em uma unica via.

_____

Locadora DVD Show
Rua Rocha Pombo, 105 - Vila Independencia
Porto Alegre - RS - CEP: 01234-567
Tel: (11) 8888-8888 . Fax: (11) 8888-8889

O Uso de JavaBeans

A medida que o código Java dentro do HTML torna-se cada vez mais complexo o desenvolvedor pode-se perguntar: Java em HTML não é o problema invertido do HTML em Servlet? O resultado não será tão complexo quanto

produzir uma página usando println()? Em outras palavras, estou novamente misturando conteúdo com forma?

Para solucionar esse problema a especificação de JSP permite o uso de JavaBeans para manipular a parte dinâmica em Java. JavaBeans já foram descritos detalhadamente em um capítulo anterior, mas podemos encarar um JavaBean como sendo apenas uma classe Java que obedece a uma certa padronização de nomeação de métodos, formando o que é denominado de *propriedade*. As propriedades de um bean são acessadas por meio de métodos que obedecem a convenção getXxxx e setXxxx. , onde Xxxx é o nome da propriedade. Por exemplo, getItem() é o método usado para retornar o valor da propriedade item. A sintaxe para o uso de um bean em uma página JSP é:

```
<jsp:useBean id="nome" class="package.class" />
```

Onde nome é o identificador da variável que conterá uma referência para uma instância do JavaBean. Você também pode modificar o atributo scope para estabelecer o escopo do bean além da página corrente.

```
<jsp:useBean id="nome" scope="session" class="package.class" />
```

Para modificar as propriedades de um JavaBean você pode usar o jsp:setProperty ou chamar um método explicitamente em um scriptlet. Para recuperar o valor de uma propriedade de um JavaBean você pode usar o jsp:getProperty ou chamar um método explicitamente em um scriptlet. Quando é dito que um bean tem uma propriedade prop do tipo T significa que o bean deve prover um método getProp() e um método do tipo setProp(T). O exemplo abaixo mostra uma página JSP e um JavaBean. A página instancia o JavaBean, altera a propriedade mensagem e recupera o valor da propriedade, colocando-o na página.

Página bean.jsp

```
<HTML> <HEAD>
<TITLE>Uso de beans</TITLE>


</HEAD> <BODY> <CENTER>
<TABLE BORDER=5> <TR><TH CLASS="TITLE"> Uso de  JavaBeans </TABLE>
</CENTER> <P>

<jsp:useBean id="teste" class=”curso.BeanSimples" />
<jsp:setProperty name="teste" property="mensagem" value=”Ola mundo!" />

<H1> Mensagem: <I>
<jsp:getProperty name="teste" property="mensagem" /> </I></H1>
</BODY> </HTML>
```

Arquivo Curso/BeanSimples.java

```
package curso;

public class BeanSimples {
private String men = "Nenhuma mensagem";

public String getMensagem() {
return(men);
 }

public void setMensagem(String men) {
this.men = men;
 }

}
```

Exemplo – *Exemplo do uso de JavaBean**.***

A figura 16 mostra o resultado da requisição dirigida à página bean.jsp.

Figura 16- *Resultado da requisição à página bean.jsp.*

Se no *tag* setProperty usarmos o valor "*" para o atributo property então todos os valores de elementos de formulários que possuírem nomes iguais à propriedades serão transferidos para as respectivas propriedades no momento do processamento da requisição. Por exemplo, seja uma página jsp contendo um formulário com uma caixa de texto com nome mensagem, como mostrado no exemplo abaixo. Note que, neste caso, a propriedade mensagem do JavaBean tem seu valor atualizado para o valor digitado na caixa de texto, sem a necessidade de uma chamada explícita no *tag* setProperty. Os valores são automaticamente convertidos para o tipo correto no bean.

```
<HTML> <HEAD><TITLE>Uso de beans</TITLE> </HEAD>
<BODY> <CENTER>
<TABLE BORDER=5> <TR><TH CLASS="TITLE"> Uso de  JavaBeans </TABLE>
</CENTER> <P>

<jsp:useBean id="teste" class="curso.BeanSimples" /> <jsp:setProperty name="teste"
property="*" />
<H1> Mensagem: <I>
<jsp:getProperty name="teste" property="mensagem" />
</I></H1>
<form method="POST" action="bean2.jsp">     Texto: <input type="text" size="20"
name="mensagem" ><br>     <INPUT TYPE=submit name=submit value="envie">
</form>

</BODY> </HTML>
```

Exemplo – *Exemplo de atualização automática da propriedade.*

A figura 17 mostra o resultado da requisição dirigida à página bean2.jsp após a digitação do texto Olá!

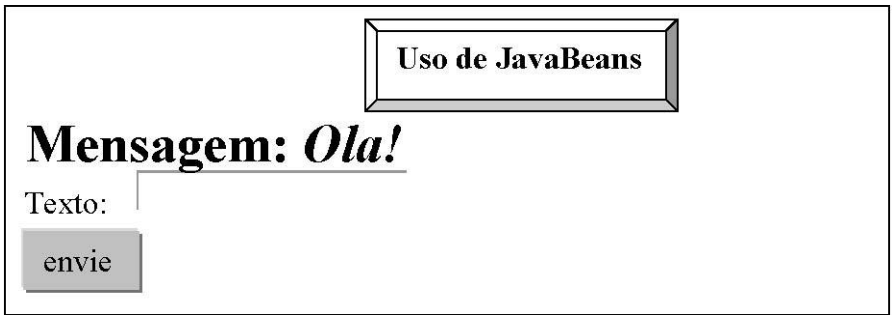


Figura 17- Resultado da requisição à página bean2.jsp.

Escopo

Existem quatro valores possíveis para o escopo de um objeto: page, request, session e application. O default é page. A tabela 3 descreve cada tipo de escopo.

Escopo	***Descrição***
page	Objetos declarados com nesse escopo são válidos até a resposta ser enviada ou a requisição ser encaminhada para outro programa no mesmo ambiente, ou seja, só podem ser referenciados nas páginas onde forem declarados. Objetos declarados com escopo page são referenciados pelo objeto pagecontext.
request	Objetos declarados com nesse escopo são válidos durante a requisição e são acessíveis mesmo quando a requisição é encaminhada para outro programa no mesmo ambiente. Objetos declarados com escopo request são referenciados pelo objeto request.
session	Objetos declarados com nesse escopo são válidos durante a sessão desde que a página seja definida para funcionar em uma sessão. Objetos declarados com escopo session são referenciados pelo objeto session.
application	Objetos declarados com nesse escopo são acessíveis por páginas no mesmo servidor de aplicação. Objetos declarados com escopo application são referenciados pelo objeto application.

Tabela 3 –*Escopo dos objetos nas páginas JSP****.***

Implementação de um Carrinho de compras

O exemplo abaixo ilustra o uso de JSP para implementar um carrinho de compras virtual. O carrinho de compras virtual simula um carrinho de compras de supermercado, onde o cliente vai colocando os produtos selecionados para compra até se dirigir para o caixa para fazer o pagamento. No carrinho de compras virtual os itens selecionados pelo usuário são armazenados em uma estrutura de dados até que o usuário efetue o pagamento. Esse tipo de exemplo exige que a página JSP funcione com o escopo session para manter o carrinho de compras durante a sessão. O exemplo compras.jsp mostra um exemplo simples de implementação de carrinho de compras. O exemplo é composto por dois arquivos: um para a página JSP e um para o JavaBean que armazena os itens selecionados.

Página compras.jsp

```
<html>
<jsp:useBean id="carrinho" scope="session" class="compra.Carrinho" />
<jsp:setProperty name="carrinho" property="*" />
<body bgcolor="#FFFFFF">
<%
carrinho.processRequest(request);
String[] items = carrinho.getItems();
if (items.length>0) {
%>
<font size=+2 color="#3333FF">Voc&ecirc; comprou os seguintes itens:</font>
<ol>
<%
for (int i=0; i<items.length; i++) {
out.println("<li>"+items[i]);
}
}
%>
</ol>
```

```
<hr>
<form type=POST action= compras.jsp>
<br><font color="#3333FF" size=+2>Entre um item para adicionar ou remover:
</font><br>
<select NAME="item">
<option>Televis&atilde;o
<option>R&aacute;dio
<option>Computador
<option>V&iacute;deo Cassete
</select>
<p><input TYPE=submit name="submit" value="adicione">
<input TYPE=submit name="submit" value="remova"></form>
</body>
</html>
```

JavaBean compra/Carrinho.java

```
Package compra;

Import javax.servlet.http.*;
Import java.util.Vector;
Import java.util.Enumeration;

Public class Carrinho {
Vector v = new Vector();
String submit = null;
String item = null;
Private void addItem(String name) {
v.addElement(name); }

Private void removeItem(String name) {
v.removeElement(name); }

Public void setItem(String name) {
item = name; }

Public void setSubmit(String s) {
submit = s; }

Public String[] getItems() {
String[] s = new String[v.size()];
v.copyInto(s);
return s;
}

private void reset() {
submit = null;
item = null;
}
```

```
public void processRequest(HttpServletRequest request){
if (submit == null) return;
if (submit.equals("adicione")) addItem(item);
else if (submit.equals("remova")) removeItem(item);
reset();
}
}
```

Exemplo – *Implementação de um carrinho de compras Virtual.*

O exemplo anterior implementa apenas o carrinho de compras, deixando de fora o pagamento dos itens, uma vez que esta etapa depende de cada sistema. Geralmente o que é feito é direcionar o usuário para outra página onde ele digitará o número do cartão de crédito que será transmitido por meio de uma conexão segura para o servidor. Existem outras formas de pagamento, como boleto bancário e dinheiro virtual. O próprio carrinho de compras geralmente é mais complexo, uma vez que os para compra devem ser obtidos dinamicamente de um banco de dados. A figura 18 mostra a tela resultante de algumas interações com o carrinho de compras.

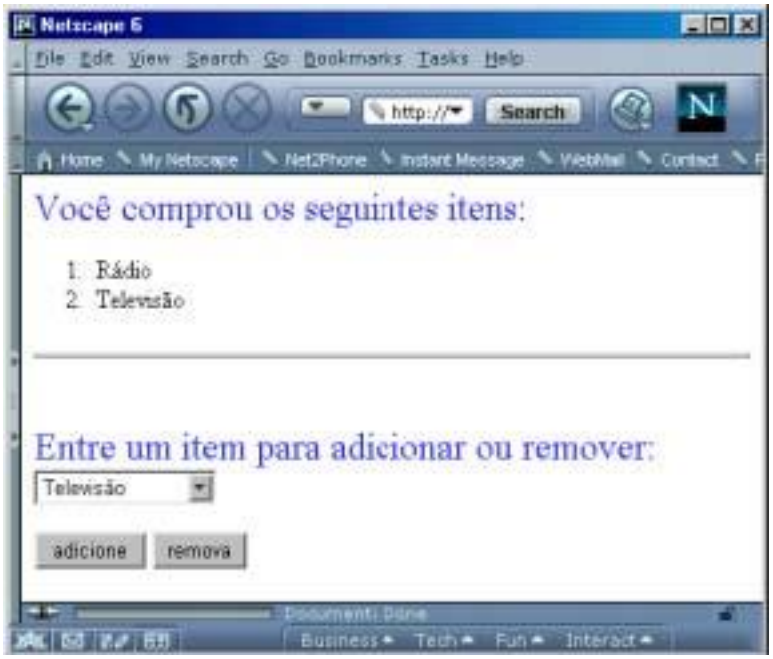


Figura 18- Carrinho de compras virtual.

Arquitetura MVC para a Web

A figura 19 contém um diagrama de blocos que mostra a participação de Servlets, JSP e JavaBeans na arquitetura proposta. A idéia é isolar cada aspecto do modelo MVC com a tecnologia mais adequada.

A página JSP é ótima para fazer o papel da visão, uma vez que possui facilidades para a inserção de componentes visuais e para a apresentação de informação.

No entanto, é um pouco estranho usar uma página JSP para receber e tratar uma requisição. Esta tarefa, que se enquadra no aspecto de controle do modelo MVC é mais adequada a um Servlet, uma vez que neste momento compontes de apresentação são indesejáveis.

Finalmente, é desejável que a modelagem do negócio fique isolada dos aspectos de interação. A proposta é que a modelagem do negócio fique contida em classes de JavaBeans.

Em aplicações mais sofisticadas a modelagem do negócio deve ser implementada por classes de Enterprise JavaBeans (EJB), no entanto esta forma de implementação foge ao escopos da apostila. Cada componente participa da seguinte forma:

•Servlets – Atuam como controladores, recebendo as requisições dos usuários. Após a realização das análises necessária sobre a requisição, instancia o JavaBean e o armazena no escopo adequado (ou não caso o bean já tenha sido criado no escopo) e encaminha a requisição para a página JSP.

•JavaBeans – Atuam como o modelo da solução, independente da requisição e da forma de apresentação. Comunicam-se com a camada intermediária que encapsula a lógica do problema.

•JSP – Atuam na camada de apresentação utilizando os JavaBeans para obtenção dos dados a serem exibidos, isolando-se assim de como os dados são obtidos. O objetivo é minimizar a quantidade de código colocado na página.

•Camada Intermediária (Middleware) – Incorporam a lógica de acesso aos dados. Permitem isolar os outros módulos de problemas como estratégias de acesso aos dados e desempenho. O uso de EJB (Enterprise JavaBeans) é recomendado para a implementação do Middleware, uma vez que os EJBs possuem capacidades para gerência de transações e persistência. Isto implica na adoção de um servidor de aplicação habilitado para EJB.

A figura 19 mostra a interação entre os componenetes.

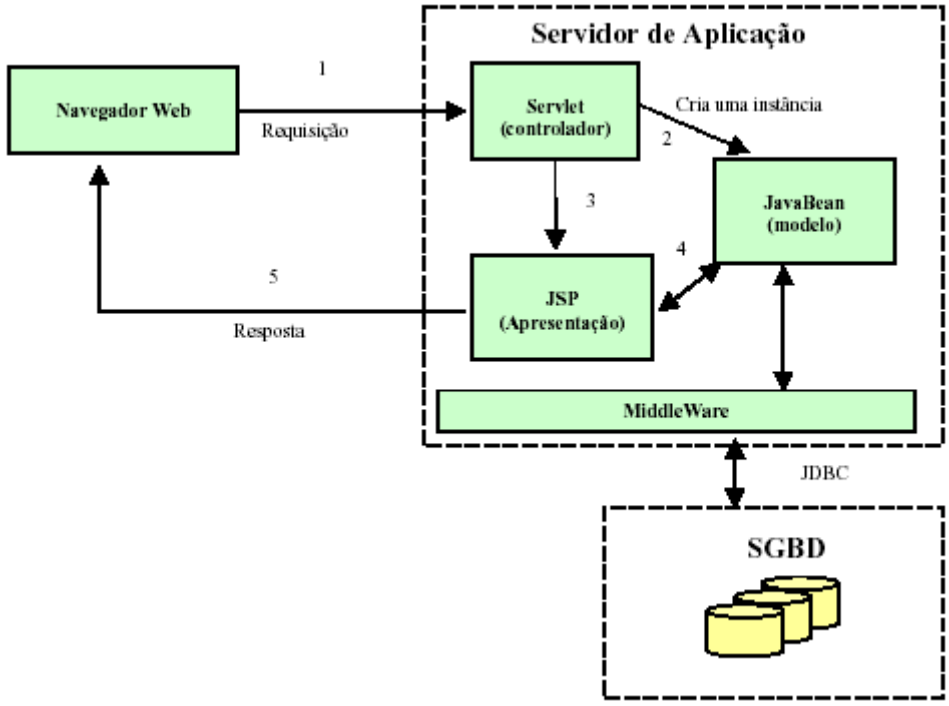


Figura 19. *Arquitetura de uma aplicação para Comércio Eletrônico.*

Essa arquitetura possui as seguintes vantagens:

1. Facilidade de manutenção: a distribuição lógica das funções entre os módulos do sistema isola o impacto das modificações.

2. Escalabilidade: Modificações necessária para acompanhar o aumento da demanda de serviços (database pooling, clustering, etc) ficam concentradas na camada intermediária.

A figura 20 mostra a arquitetura física de uma aplicação de comércio eletrônico.

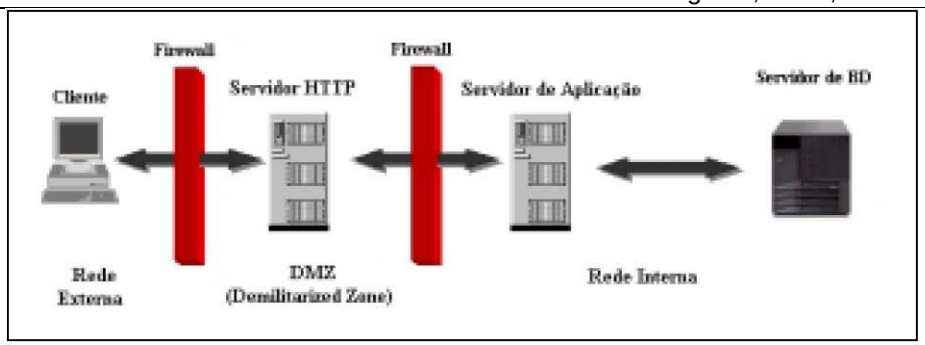


Figura 20. *Arquitetura física de uma aplicação para Comércio Eletrônico.*

Demilitarized Zone (DMZ) é onde os servidores HTTP são instalados. A DMZ é protegida da rede púbica por um firewall, também chamado de firewall de protocolo. O firewall de protocolo deve ser configurado para permitir tráfego apenas através da porta 80. Um segundo firewall, também chamado de firewall de domínio separa a DMZ da rede interna. O firewall de domínio deve ser configurado para permitir comunicação apenas por meio das portas do servidor de aplicação

Agenda Web:

Um Exemplo de uma aplicação Web usando a arquitetura MVC

O exemplo a seguir mostra o desenvolvimento da agenda eletrônica para o funcionamento na Web. A arquitetura adotada é uma implementação do modelo MVC. Apenas, para simplificar a solução, a camada intermediária foi simplificada e é implementada por um JavaBean que tem a função de gerenciar a conexão com o banco de dados. O banco de dados será composto por duas tabelas, uma para armazenar os usuários autorizados a usar a tabela e outra para armazenar os itens da agenda. A figura 21 mostra o esquema conceitual do banco de dados e a figura 22 mostra o comando para a criação das tabelas. Note que existe um relacionamento entre a tabela USUARIO e a tabela PESSOA, mostrando que os dados pessoais sobre o usuário ficam armazenados na agenda.

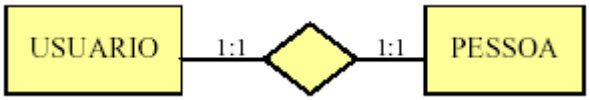


Figura 21. *Esquema conceitual do banco de dados para a agenda.*

As tabelas do BD devem ser criadas de acordo com o seguinte script:

```
CREATE TABLE PESSOA (ID INT PRIMARY KEY,
 NOME VARCHAR(50) NOT NULL,
 TELEFONE VARCHAR(50),
 ENDERECO VARCHAR(80),
 EMAIL VARCHAR(50),
 HP VARCHAR(50),
 CELULAR VARCHAR(20),
 DESCRICAO VARCHAR(80));


CREATE TABLE USUARIO (ID INT PRIMARY KEY,
 LOGIN VARCHAR(20) NOT NULL,
 SENHA VARCHAR(20) NOT NULL,
 CONSTRAINT FK_USU FOREIGN KEY (ID)
 REFERENCES PESSOA(ID));
```

Figura 22. *Script para criação das tabelas.*

Para se usar a agenda é necessário que exista pelo menos um usuário cadastrado. Como no exemplo não vamos apresentar uma tela para cadastro de usuários será preciso cadastrá-los por meio comandos SQL. Os comandos da figura 23 mostram como cadastrar um usuário.

```
INSERT INTO PESSOA(ID,NOME,TELEFONE,ENDERECO,EMAIL)
VALUES(0,'Amariles Lopes','3315-4535,
            'Andradas,105','amarileslopes@ig.com.br');
INSERT INTO USUARIO(ID,LOGIN,SENHA) VALUES(0,'Amariles','senha');
```

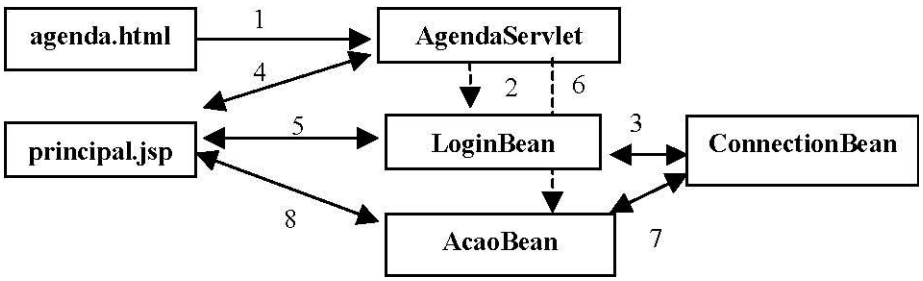
Figura 23. *Script para cadastra um usuário.*

O sistema **e-agenda** é composta pelos seguintes arquivos:

Arquivo	***Descrição***
agenda.html	Página inicial do site, contendo o formulário para a entrada do login e senha para entrar no restante do site.
principal.jsp	Página JSP contendo o formulário para entrada de dados para inserção, remoção ou consulta de itens da agenda.
LoginBean.java	JavaBean responsável por verificar se o usuário está autorizado a acessar a agenda.
AgendaServlet.java	Servlet responsável pelo tratamento de requisições sobre alguma função da agenda (consulta, inserção e remoção)
AcaoBean.java	JavaBean responsável pela execução da ação solicitada pelo usuário.
ConnectionBean.java	JavaBean responsável pelo acesso ao DB e controle das conexões.

Tabela 4. *Arquivos do sistema e-agenda.*

O diagrama de colaboração abaixo mostra as interação entre os componentes do sistema.



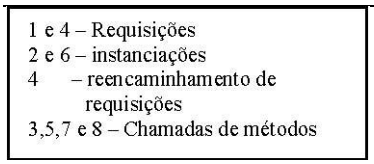


Figura 24. *Interação entre os componentes do sistema.*

Descreveremos agora cada componente da aplicação. O exemplo agenda.html mostra código HTML da página agenda.html. Esta é a página inicial da aplicação. Ela contém o formulário para a entrada do login e senha para entrar no restante do site.

```
1       <HTML>
2       <HEAD>
4       <TITLE>Agenda</TITLE>
5       </HEAD>
6       <BODY BGCOLOR="#FFFFFF">
7       <P align="center"><IMG src="tit.gif" width="350" height="100" border="0"></P>
8       <BR>
9
10      <CENTER>
11      <FORM method="POST" name="TesteSub" onsubmit="return TestaVal()"
12      action="/agenda/agenda"><BR>
13      Login:<INPUT size="20" type="text" name="login"><BR><BR>
14      Senha:<INPUT size="20" type="password" name="senha"><BR><BR><BR>
15      <INPUT type="submit" name="envia" value="Enviar">
16      <INPUT size="3" type="Hidden" name="corrente" value="0"><BR>
17      </FORM>
18      </CENTER>
19      </BODY>
20      </HTML>
```

Exemplo – agenda.html.

O formulário está definido nas linha 11 a 17. Na linha 12 o parâmetro action indica a URL que dever receber a requisição. A URL é virtual e sua associação com o Servlet AgendaServlet será definida no arquivo web.xml. Na linha 16 é definido um campo oculto (Hidden) como o nome de corrente e valor 0. Ele será usado pelo AgendaServlet reconhecer a página de onde saiu a requisição. As

linha 19 a 31 definem uma função em JavaScript que será usada para verificar se o usuário digitou o nome e a senha antes de enviar a requisição ao usuário. O uso de JavaScript no lado cliente para criticar a entrada do usuário é muito comum pois diminui a sobrecarga do servidor.

O exemplo a seguir mostra código da página principal.jsp. Esta página contém o formulário para entrada de dados para inserção, remoção ou consulta de itens da agenda.

```
1   <HTML><HEAD>
2   <TITLE>Tela da Agenda </TITLE>
3   </HEAD><BODY bgcolor="#FFFFFF">
4   <%@ page session="true" import="agenda.*" %>
5
6   <%
7   agenda.LoginBean lb = (agenda.LoginBean) session.getAttribute("loginbean");
8   agenda.AcaoBean ab = (agenda.AcaoBean) request.getAttribute("acaobean");
9   if (lb != null && lb.getStatus())
10  { %>
11  <H2>Sess&atilde;o do <%= lb.getNome() %></H2>
12  <%
13  if (ab!=null) out.println(ab.toString());
14  %>
15
16  <P><BR></P>
17  <FORM method="POST" name="formprin" onsubmit="return TestaVal()"
18  action="/agenda/agenda">
19  Nome: <INPUT size="50" type="text" name="nome"><BR>
20  Telefone: <INPUT size="20" type="text" name="telefone"><BR>
21  Endere&ccedil;o: <INPUT size="50" type="text" name="endereco"><BR>
22  Email: <INPUT size="50" type="text" name="email"><BR><BR>
23  P&aacute;gina: <INPUT size="50" type="text" name="pagina"><BR>
24  Celular: <INPUT size="20" type="text" name="celular"><BR>
25  Descri&ccedil;&atilde;o: <INPUT size="20" type="text" name="descricao">
26  <BR><CENTER>
27  <INPUT type="submit" name="acao" value="Consulta">
28  <INPUT type="submit" name="acao" value="Insere">
```

```
29      <INPUT type="submit" name="acao" value="Apaga"></CENTER>
30      <INPUT size="3" type="Hidden" name="corrente" value="1">
31      </FORM>
32
33      <SCRIPT language="JavaScript"> <!--
34      function TestaVal()
35      {
36      if (document.formprin.nome.value == "" &&
37      document.formprin.descricao.value== "")
38      {
39      alert ("Campo Nome ou Descricao devem ser Preenchidos!")
40      return false
41      }
42      else
43      {
44      return true
45      }
46      }
47      //--></SCRIPT>
48
49      <%
50      }
51      else
52      {
53      session.invalidate();
54      %>
55      <H1>Usu&aacute;rio n&atilde;o autorizado</H1>
56      <%
57      }
58      %>
59      </BODY></HTML>
```

Exemplo – principal.jsp.

Na linha 4 a diretiva page define que o servidor deve acompanhar a sessão do usuário e importa o pacote agenda.

Na linha 7 um objeto da classe agenda.LoginBean é recuperado da sessão por meio do método getAttribute(). Para recuperar o objeto é preciso passar para o método o nome que está associado ao objeto na sessão.

De forma semelhante, na linha 8 um objeto da classe agenda.AcaoBean é recuperado da requisição por meio do método getAttribute(). Este objeto é recuperado da requisição porque cada requisição possui uma ação diferente associada.

Na linha 9 é verificado se objeto agenda.LoginBean foi recuperado e se o retorno do método getStatus() é true. Se o objeto agenda.LoginBean não foi recuperado significa que existe uma tentativa de acesso direto à página principal.jsp sem passar primeiro pela página agenda.html ou que a sessão se esgotou. Se o método getStatus() retornar false significa que o usuário não está autorizado a acessar essa página. Nestes casos é processado o código associado ao comando else da linha 51 que apaga a sessão por meio do método invalidate() do objeto HttpSession (linha 53) e mostra a mensagem “Usuário não autorizado” (linha 55). Caso o objeto indique que o usuário está autorizado os comandos internos ao if são executados.

Na linha 11 é mostrada uma mensagem com o nome do usuário obtido por meio do método getNome() do objeto agenda.LoginBean . Na linha 13 é mostrado o resultado da ação anterior por meio do método toString() do objeto agenda.AcaoBean. A ação pode ter sido de consulta, inserção de um novo item na agenda e remoção de um item na agenda.

No primeiro caso é mostrado uma lista dos itens que satisfizeram a consulta.

No segundo e terceiro casos é exibida uma mensagem indicado se a operação foi bem sucedida.

As linhas 17 a 31 definem o código do formulário de entrada.

Nas linhas 17 e 18 são definidos os atributos do formulário. O atributo method indica a requisição será enviada por meio do método POST. O atributo name define o nome do formulário como sendo formprin. O atributo onsubmit define que a função javaSript TestaVal() deve ser executada quando o formulário for submetido.

Finalmente, o atributo action define a URL para onde a requisição deve ser enviada. Neste caso a URL é agenda/agenda que está mapeada para o Servlet AgendaServlet. O mapeamento é feito no arquivo web.xml do diretório web-inf do contexto agenda, como mostrado na figura 24.

As linhas 19 a 25 definem os campos de texto para entrada dos valores.

As linhas 27 a 29 definem os botões de submit. Todos possuem o mesmo nome, de forma que o Servlet precisa apenas examinar o valor do parâmetro acao para determinar qual ação foi solicitada

Na linha 30 é definido um campo oculto (Hidden) como o nome de corrente e valor 0. Ele será usado pelo AgendaServlet reconhecer a página de onde saiu a requisição.

As linha 33 a 47 definem uma função em JavaScript que será usada para verificar se o usuário entrou com valores nos campos de texto nome ou decricao. No mínimo um desses campos deve ser preenchido para que uma consulta possa ser realizada.

O exemplo ConnectionBean.java mostra código do JavaBean usado para intermediar a conexão com o banco de dados. O JavaBean ConnectionBean tem a responsabilidade de abrir uma conexão com o banco de dados, retornar uma referência desta conexão quando solicitado e registrar se a conexão esta livre ou ocupada. Neste exemplo estamos trabalhando com apenas uma conexão com o banco de dados porque a versão gerenciador de banco de dados utilizado (PointBase[tm]), por ser um versão limitada, permite apenas uma conexão aberta.

Se o SGBD permitir varias conexões simultâneas pode ser necessário um maior controle sobre as conexões, mantendo-as em uma estrutura de dados denominada de *pool de conexões*.

Na linha 12 podemos observar que o construtor da classe foi declarado com o modificador de acesso private. Isto significa que não é possível invocar o construtor por meio de um objeto de outra classe. Isto é feito para que se possa ter um controle sobre a criação de instâncias da classe.

No nosso caso permitiremos apenas que uma instância da classe seja criada, de modo que todas as referências apontem para esse objeto. Esta técnica de programação, onde se permite uma única instância de uma classe é denominada de padrão de projeto *Singleton*. O objetivo de utilizarmos este padrão é porque desejamos que apenas um objeto controle a conexão com o banco de dados.

Se o construtor não pode ser chamado internamente como uma instância da classe é criada e sua referência é passada para outros objetos? Esta é a tarefa do método estático getInstance() (linhas 14 a 19). Este método verifica se já existe a instância e retorna a referência. Caso a instância não exista ela é criada antes de se retornar a referência.

O método init() (linhas 21 a 27) é chamado pelo construtor para estabelecer a conexão com o SGBD. Ele carrega o driver JDBC do tipo 4 para PointBase e obtém uma conexão com o SGBD.

O método devolveConnection() (linhas 29 a 34) é chamado quando se deseja devolver a conexão. Finalmente, o método getConnection() (linhas 36 a 46) é chamado quando se deseja obter a conexão.

```
1       package agenda;
2
3       import java.sql.*;
4       import java.lang.*;
5       import java.util.*;
6
7       public class ConnectionBean {
8       private Connection con=null;
9       private static int clients=0;
10      static private ConnectionBean instance=null;
11
12      private ConnectionBean() { init(); }
13
14      static synchronized public ConnectionBean getInstance() {
15      if (instance == null) {
16      instance = new ConnectionBean();
17      }
18      return instance;
19      }
20
21      private void init() {
22      try {
23      Class.forName("com.pointbase.jdbc.jdbcUniversalDriver");
24      con=
25      DriverManager.getConnection("jdbc:pointbase:agenda","PUBLIC","public");
26      } catch(Exception e){System.out.println(e.getMessage());};
27      }
28
29      public synchronized void devolveConnection(Connection con) {
30      if (this.con==con) {
31      clients--;
32      notify();
33      }
34      }
```

```
35
36      public synchronized Connection getConnection() {
37      if(clients>0) {
38      try { wait(5000); }
41      catch (InterruptedException e) {};
42      if(clients>0) return null;
43      }
44      clients ++;
45      return con;
46      }
47
48      } //fim da classe
```

Exemplo – *ConnectionBean.java.*

O exemplo abaixo mostra código do JavaBean usado para verificar se o usuário está autorizado a usar a agenda. O JavaBean LoginBean recebe o nome e a senha do usuário, obtém a conexão com o SGBD e verifica se o usuário está autorizado, registrando o resultado da consulta na variável status (linha 10). Tudo isso é feito no construtor da classe (linhas 12 a 35).

Note que na construção do comando SQL (linhas 17 a 20) é inserido uma junção entre as tabelas PESSOA e USUARIO de modo a ser possível recuperar os dados relacionados armazenados em ambas as tabelas. Os métodos getLogin(), getNome() e getStatus() (linhas 36 a 38) são responsáveis pelo retorno do login, nome e status da consulta respectivamente.

```
1       package agenda;
2
3       import java.sql.*;
4       import java.lang.*;
5       import java.util.*;
6
7       public class LoginBean {
8       protected String nome = null;
9       protected String login= null;
10      protected boolean status= false;
11      public LoginBean(String login, String senha)
12      {
13      this.login = login;
```

```
14      Connection con=null;
15      Statement stmt =null;
16      String consulta = "SELECT NOME FROM PESSOA, USUARIO "+
17      "WHERE USUARIO.ID = PESSOA.ID AND "+
18      "USUARIO.SENHA ='"+senha+"' AND "+
19      "USUARIO.LOGIN ='"+login+"'";
20      try {
21      con=ConnectionBean.getInstance().getConnection();
22      stmt = con.createStatement();
23      ResultSet rs =stmt.executeQuery(consulta);
24      if(rs.next()) {
25      status = true;
26      nome = rs.getString("NOME");
27
68      }
28      } catch(Exception e){System.out.println(e.getMessage());}
29      finally {
30      ConnectionBean.getInstance().devolveConnection(con);
31      try{stmt.close();}catch(Exception ee){};
32      }
33      }
34      public String getLogin(){return login;}
35      public String getNome(){return nome;}
36      public boolean getStatus(){return status;}
37      }
38      }
39      } catch(Exception e){System.out.println(e.getMessage());}
```

Exemplo – *LoginBean.java.*

O exemplo AgendaServlet.java mostra código do Servlet que implementa a camada de controle do modelo MVC. O Servlet AgendaServlet recebe as requisições e, de acordo com os parâmetros, instância os JavaBeans apropriados e reencaminha as requisições para as páginas corretas.

Tanto o método doGet() (linhas 9 a 12) quanto o método doPost()(linhas 13 a 17) invocam o método performTask()(linhas 19 a 61) que realiza o tratamento da

requisição. Na linhas 24 a 26 do método performTask() é obtido o valor do parâmetro corrente que determina a página que originou a requisição. Se o valor for nulo é assumido o valor default zero.

Na linha 30 é executado um comando switch sobre esse valor, de modo a desviar para bloco de comandos adequado. O bloco que vai da linha 32 até a linha 43 trata a requisição originada em uma página com a identificação 0 (página agenda.html).

Nas linhas 32 e 33 são recuperados o valor de login e senha digitados pelo usuário. Se algum desses valores for nulo então a requisição deve ser reencaminhada para a página de login (agenda.html) novamente (linha 35).

Caso contrário é instanciado um objeto LoginBean, inserido na sessão corrente e definida a página principal.jsp como a página para o reencaminhamento da requisição (linhas 38 a 41).

Já o bloco que vai da linha 44 até a linha 54 trata a requisição originada em uma página com a identificação 1 (página principal.jsp).

Na linha 44 é recuperado o objeto HttpSession corrente. O argumento false é utilizado para impedir a criação de um novo objeto HttpSession caso não exista um corrente. Se o valor do objeto for null, então a requisição deve ser reencaminhada para a página de login (linha 47). Caso contrário é instanciado um objeto AcaoBean, inserido na requisição corrente e definida a página principal.jsp como a página para o reencaminhamento da requisição (linhas 50 a 52).

Na linha 56 a requisição é reencaminhada para a página definida (página agenda.html ou principal.jsp).

```
1   package agenda;
2
3   import javax.servlet.*;
4   import javax.servlet.http.*;
5   import agenda.*;
6
7   public class AgendaServlet extends HttpServlet
8   {
9   public void doGet(HttpServletRequest request, HttpServletResponse response)
10  {
11  performTask(request,response);
```

```
12      }
13      public void doPost(HttpServletRequest request,
14      HttpServletResponse response)
15      {
16      performTask(request,response);
17      }
18
19      public void performTask(HttpServletRequest request,
20      HttpServletResponse response)
21      {
22      String url;
23      HttpSession sessao;
24      String corrente = request.getParameter("corrente");
25      int icorr=0;
26      if (corrente != null) icorr = Integer.parseInt(corrente);
27
28      try
29      {
30      switch(icorr)
31      {
32      case 0: String login = request.getParameter("login");
33      String senha = request.getParameter("senha");
34      if (login == null||senha == null)
35      url= "/agenda.html";
36      else
37      {
38      sessao = request.getSession(true);
39      sessao.setAttribute("loginbean",
40      new agenda.LoginBean(login,senha));
41      url= "/principal.jsp";
42      };
43      break;
44      case 1:
45      sessao = request.getSession(false);
```

```
46      if (sessao == null)
47      url= "/agenda.html";
48      else
49      {
50      request.setAttribute("acaobean",
51      new agenda.AcaoBean(request));
52      url= "/principal.jsp";
53      };
54      break;
55      }
56      getServletContext().getRequestDispatcher(url).forward(request,response);
57      }catch (Exception e) {
58      System.out.println("AgendaServlet falhou: ");
59      e.printStackTrace();
60      }
61      }
62      }
```

Exemplo – *AgendaServlet.java.*

O exemplo AcaoBean mostra código do JavaBean usado para realizar a manutenção da agenda. O JavaBean AcaoBean é responsável pela consulta, remoção e inserção de novos itens na agenda. Um objeto StringBuffer referenciado pela variável retorno é utilizado pelo JavaBean para montar o resultado da execução. O construtor (linhas 16 a 27) verifica o tipo de requisição e invoca o método apropriado.

O método consulta() (linhas 29 a 77) é responsável pela realização de consultas. As consultas podem ser realizadas sobre o campo nome ou descrição e os casamentos podem ser parciais, uma vez que é usado o operador LIKE. A consulta SQL é montada nas linhas 40 a 47. Na linha 50 é obtida uma conexão com SGBD por meio do objeto ConnectionBean. Na linha 57 o comando SQL é executado e as linhas 59 a 72 montam o resultado da consulta.

O método insere() (linhas 79 a 148) é responsável por inserir um item na agenda. Na linha 95 é obtida uma conexão com SGBD por meio do objeto ConnectionBean. Para inserir um novo item é preciso obter o número do último identificador usado, incrementar o identificador e inserir na base o item com o identificador incrementado. Esta operação requer que não seja acrescentado nenhum identificador entre a operação de leitura do último identificador e a inserção de um novo item. Ou seja, é necessário que essas operações sejam

tratadas como uma única transação e o isolamento entre as transações sejam do nível *Repeatable Read*. A definição do inicio da transação é feita no comando da linha 102. A mudança do nível de isolamento é feita pelos comandos codificados nas linha 103 a 109.

Na linha 112 é invocado o método obtemUltimo() (linhas 150 a 171) para obter o último identificador utilizado. As linhas 114 a 128 montam o comando SQL para a execução. O comando SQL é executado na linha 131. O fim da transação é definido pelo comando da linha 132. Ao fim da transação, de forma a não prejudicar a concorrência, o nível de isolamento deve retornar para um valor mais baixo. Isto é feito pelos comandos das linhas 133 a 137.

O método apaga() (linhas 173 a 201) é responsável por remover um item da agenda. As linhas 175 a 180 contém o código para verificar se o usuário digitou o nome associado ao item que deve ser removido.

A linha 181 montam o comando SQL para a execução. Na linha 184 é obtida uma conexão com SGBD por meio do objeto ConnectionBean. O comando SQL é executado na linha 191.

```
1       package agenda;
2
3       import java.lang.*;
4       import java.util.*;
5       import java.sql.*;
6
7       public class AcaoBean
8       {
9       private Connection con=null;
10      private StringBuffer retorno = null;
11      private Statement stmt=null;
12      private String [] legenda= {"C&oacute;digo","Nome","Telefone",
13              "Endere&ccedil;o", "email","hp",
14              "celular","Descri&ccedil;&atilde;o"};
15
16      public AcaoBean(javax.servlet.http.HttpServletRequest request)
17      {
18      String acao = request.getParameter("acao");
19      if (acao.equals("Consulta"))
```

```
20      {
21      String nome = request.getParameter("nome");
22      String descri = request.getParameter("descricao");
23      consulta(nome,descri);
24      }
25      else if (acao.equals("Insere")) insere(request);
26      else if (acao.equals("Apaga")) apaga(request);
27      }
28
29      private void consulta(String nome,String descri)
30      {
31      String consulta = null;
32
33      if ((nome == null||nome.length()<1) &&
34      (descri == null|| descri.length()<1))
35      {
36      retorno = new StringBuffer("Digite o nome ou descricao!");
37      return;
38      }
39
40      if (descri == null|| descri.length()<1)
41        consulta = "SELECT * FROM PESSOA WHERE NOME LIKE '%"+
42              nome+"%'"+" ORDER BY NOME";
43      else if (nome == null|| nome.length()<1)
44        consulta = "SELECT * FROM PESSOA WHERE DESCRICAO LIKE '%"+
45              descri+"%'"+" ORDER BY NOME";
46      else consulta="SELECT * FROM PESSOA WHERE DESCRICAO LIKE '%"+
47              descri+"%' AND NOME LIKE '%"+nome+"%' ORDER BY NOME";
48      try
49      {
50      con=ConnectionBean.getInstance().getConnection();
51      if (con == null)
52      {
53      retorno = new StringBuffer("Servidor ocupado. Tente mais tarde.!");
```

```
54      return;
55      }
56      stmt = con.createStatement();
57      ResultSet rs = stmt.executeQuery(consulta);
58
59      retorno = new StringBuffer();
60      retorno.append("<br><h3>Resultado</h3><br>");
61      while(rs.next())
62      {
63      retorno.append("ID:").append(rs.getString("id"));
64      retorno.append("<br>Nome:").append(rs.getString("Nome"));
65      retorno.append("<br>Telefone:").append(rs.getString("Telefone"));
66      retorno.append("<br>Endereco:").append(rs.getString("Endereco"));
67      retorno.append("<br>email:").append(rs.getString("email"));
68      retorno.append("<br>hp:").append(rs.getString("hp"));
69      retorno.append("<br>celular:").append(rs.getString("celular"));
70      retorno.append("<br>descricao:").append(rs.getString("descricao"));
71      retorno.append("<br><br>");
72      }
73      } catch(Exception e){System.out.println(e.getMessage());}
74      finally {ConnectionBean.getInstance().devolveConnection(con);
75      try{stmt.close();}catch(Exception ee){};
76      }
77      }
78
79      private void insere(javax.servlet.http.HttpServletRequest request)
80      {
81      String[] par = {"telefone","endereco","email","hp","celular","descricao"};
82
83      StringBuffer comando = new StringBuffer("INSERT INTO PESSOA(");
84      StringBuffer values = new StringBuffer(" VALUES(");
85
86      String aux = request.getParameter("nome");
87      if (aux == null || aux.length()<1)
```

```
88      {
89      retorno = new StringBuffer("<br><h3>Digite o nome!</h3><br>");
90      return;
91      }
92
93      try
94      {
95      con=ConnectionBean.getInstance().getConnection();
96      if (con == null)
97      {
98      retorno = new StringBuffer("Servidor ocupado. Tente mais tarde!");
99      return;
100     }
101
102     con.setAutoCommit(false);
103     DatabaseMetaData meta=con.getMetaData();
104
105     if(meta.supportsTransactionIsolationLevel(
106     con.TRANSACTION_REPEATABLE_READ)) {
107     con.setTransactionIsolation(
108     con.TRANSACTION_REPEATABLE_READ);
109     }
110
111
112     int ultimo = obtemUltimo(con);
113     if (ultimo==-1) return;
114     ultimo++;
115     comando.append("id,nome");
116     values.append(ultimo+",'").append(aux).append("'");
117
118     for(int i=0;i<par.length;i++)
119     {
120     aux = request.getParameter(par[i]);
121     if (aux != null && aux.length()>0)
```

```
122     {
123     comando.append(",").append(par[i]);
124     values.append(",'").append(aux).append("'");
125     }
126     }
127     comando.append(")");
128     values.append(")");
129     aux = comando.toString()+values.toString();
130     stmt = con.createStatement();
131     stmt.executeUpdate(aux);
132     con.setAutoCommit(true);
133     if(meta.supportsTransactionIsolationLevel(
134     con.TRANSACTION_READ_COMMITTED)) {
135     con.setTransactionIsolation(
136     con.TRANSACTION_READ_COMMITTED);
137     }
138     retorno = new StringBuffer("<br><h3>Inserido!</h3><br>");
139     return;
140     } catch(Exception e)
141     {retorno =
142     new StringBuffer("<br><h3>Erro:"+e.getMessage()+"!</h3><br>"); }
143     finally
144     {
145     ConnectionBean.getInstance().devolveConnection(con);
146     try{stmt.close();}catch(Exception ee){};
147     }
148     }
149
150     private int obtemUltimo(Connection con)
151     {
152     String consulta = "SELECT MAX(ID) AS MAX FROM PESSOA";
153     try
154     {
155     if (con == null)
```

```
156     {
157     retorno = new StringBuffer("Servidor ocupado. Tente mais tarde.!");
158     return -1;
159     }
160     stmt = con.createStatement();
161     ResultSet rs = stmt.executeQuery(consulta);
162     if(rs.next())
163     return Integer.parseInt(rs.getString("max"));
164     else return 0;
165     } catch(Exception e) {
166     retorno =
167     new StringBuffer("<br><h3>Erro:"+e.getMessage()+"!</h3><br>");
168     return -1;
169     }
170     finally {try{stmt.close();}catch(Exception ee){};}
171     }
172
173     private void apaga(javax.servlet.http.HttpServletRequest request)
174     {
175     String aux = request.getParameter("nome");
176     if (aux == null || aux.length()<1)
177     {
178     retorno = new StringBuffer("<br><h3>Digite o nome!</h3><br>");
179     return;
180     }
181     String consulta = "DELETE FROM PESSOA WHERE NOME ='"+aux+"'";
182     try
183     {
184     con=ConnectionBean.getInstance().getConnection();
185     if (con == null)
186     {
187     retorno = new StringBuffer("Servidor ocupado. Tente mais tarde.!");
188     return;
189     }
```

```
190     stmt = con.createStatement();
191     stmt.executeUpdate(consulta);
192
193     retorno = new StringBuffer("<br><h3>Removido!</h3><br>");
194     return;
195     } catch(Exception e){
196     retorno = new StringBuffer("<br><h3>Erro:"+e.getMessage()+"!</h3><br>");
197     }
198     finally {
199     ConnectionBean.getInstance().devolveConnection(con);
200     try{stmt.close();}catch(Exception ee){};}
201     }
202
203     public String[] getLeg(){return legenda;}
204     public String toString(){return retorno.toString();}
205     }
```

Exemplo – AcaoBean.java.

Instalação

Para instalar crie a seguinte estrutura de diretório abaixo do diretório webapps do Tomcat:

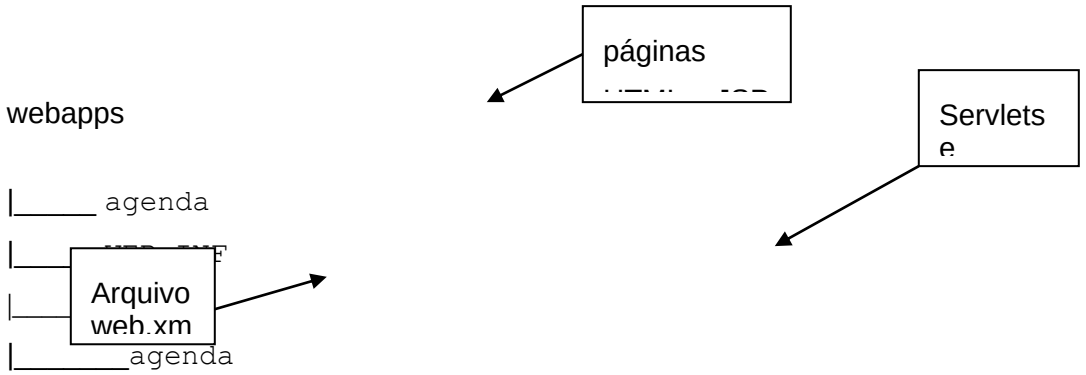


Figura 25. Estrutura de diretórios para a aplicação agenda.

O arquivo web.xml deve ser alterado para conter mapeamento entre a URL agenda e o Servlet AgendaServlet.

```
...
<web-app>
        <servlet>
                <servlet-name>
                  agenda
                </servlet-name>
                <servlet-class>
                  agenda.AgendaServlet
                </servlet-class>
        </servlet>
        <servlet-mapping>
                <servlet-name>
                  agenda
                </servlet-name>
                <url-pattern>
                  /agenda
                </url-pattern>
```

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

Figura . *Arquivo web.xml para a agenda.*

Considerações sobre a solução

A aplicação acima implementa uma agenda que pode ser acessada por meio da Internet, no entanto, devido à falta de espaço e à necessidade de destacarmos os pontos principais, alguns detalhes foram deixados de lado, como por exemplo uma melhor interface com o usuário. Abaixo seguem alguns comentários sobre algumas particularidades da aplicação:

1. O JavaBean da classe LoginBean é armazenado na sessão para permitir a verificação se o acesso ao site é autorizado. Isto impede que os usuários tentem acessar diretamente a página principal.jsp da agenda. Caso tentem fazer isso, a sessão não conterá um objeto LoginBean associado e, portanto, o acesso será recusado.

2. O JavaBean da classe AcaoBean é armazenado no objeto request uma vez que sua informações são alteradas a cada requisição. Uma forma mais eficiente seria manter o objeto AcaoBean na sessão e cada novo requisição invocar um método do AcaoBean para gerar os resultados. No entanto, o objetivo da nossa implementação não é fazer a aplicação mais eficiente possível, e sim mostrar para o leitor uma aplicação com variadas técnicas.

3. Apesar de termos adotado o padrão MVC de desenvolvimento a aplicação não exibe uma separação total da camada de apresentação (Visão) em relação à camada do modelo. Parte do código HTML da visão é inserido pelo AcaoBean no momento da construção da String contendo o resultado da ação. Isto foi feito para minimizar a quantidade de código Java na página JSP. Pode-se argumentar que neste caso a promessa da separação entre as camadas não é cumprida totalmente. Uma solução para o problema seria gerar o conteúdo em XML e utilizar um analisador de XML para gerar a página de apresentação. No entanto, o uso da tecnologia XML foge ao escopo da apostila.

4. A solução apresenta código redundante para criticar as entradas do usuário. Existe código JavaScript nas páginas, e código Java no Servlet e JavaBeans. O uso de código JavaScript nas páginas para críticas de entrada é indispensável para aliviarmos a carga sobre o servidor. Já o código para crítica no servidor não causa impacto perceptível e útil para evitar tentativas de violação.

5. O código exibe uma preocupação com a com concorrência de acessos ao banco de dados que aparentemente não é necessária, uma vez que apenas uma conexão por vez é permitida. No entanto, procuramos fazer o código o mais genérico possível no pouco espaço disponível.

Tag Files

Um dos recursos mais interessantes do JSP é, sem dúvida, a possibilidade de se criar tags customizadas (Custom Tags). Com elas, podemos obter muitas melhorias no desenvolvimento, tais como evitar código repetitivo e criar uma interface web componentizada.

E as taglibs estão soltas por aí, Internet a fora, sempre presentes no mundo Java. A maioria dos frameworks web possui taglibs, para funções como iterações, condições, i18n. Há por aí tags para paginação, exibição de gráficos, e outros recursos visuais. Você já deve ter ouvido falar da JSTL (Java Standart Tag Libraries). O emergente JSF (Java Server Faces) também faz extensivo uso de tags.

Enfim, se você ficou com água na boca, e quer criar suas próprias taglibs, há um recurso muito simples e poderoso para fazer isso: os **Tag Files**.

O que são Tag Files?

Tag Files são arquivos com extensão **.tag**, na qual a custom tag é programada. Não é necessário criar nenhuma classe Java.

Preparação

Geralmente os tag files são empacotados em um arquivo JAR, que é colocado na pasta **WEB-INF/lib** da sua aplicação. Entretanto, também podemos colocar os tag files na pasta **WEB-INF/tags**.

O segundo método é mais adequado para o desenvolvimento, enquanto o primeiro para a distribuição das tags.

Assim, crie a pasta WEB-INF/tags. Podemos colocar os nossos tag files diretamente nessa pasta, ou criar subpastas, uma para cada conjunto de tags. Dessa forma, crie a subpasta **mytags**.

Pronto, agora podemos começar a desenvolver nossas tags.

Minha primeira Tag! (não é HelloWorld)

Para começar, vamos criar uma tag que informa a data atual do sistema operacional, de acordo com o Locale default. Dentro de WEB-INF/tags/mytags, crie o arquivo **dataDeHoje.tag**.

```
<%@tag description="descricao" pageEncoding="UTF-8" import="java.util.*,
java.text.*"%>
<%
DateFormat formatador = DateFormat.getDateInstance(DateFormat.FULL,
Locale.getDefault());
out.print(formatador.format(new Date()));
%>
```

Todo tag file deve começar com a diretiva @**tag**. Nela, colocamos informações gerais sobre a tag, sendo que as principais são essas que estão no exemplo. Depois, programamos a tag. Obtemos o formatador e escrevemos a data formatada.

Pois bem, agora criemos a página **exibeDataDeHoje.jsp**.

```
<%@taglib prefix="mytags" tagdir="/WEB-INF/tags/mytags/"%>
<html>
<head> <title>Tutorial - TagFiles</title> </head>
<body>
<h1><mytags:dataDeHoje/></h1>
</body>
</html>
```

Observe a diretiva @**taglib**. Ela é responsável por definir uma taglib que será usada na página. Podemos ter várias diretivas @taglib, que definirão várias taglibs a serem usadas. Informamos o prefixo **(prefix),** que pode ser qualquer um, e depois o diretório das tags **(tagDir)**. Quando empacotarmos as tags em um JAR, ao invés de tagDir, usaremos **uri**.

Depois, podemos usar a nossas tags livremente, como demonstrado na parte em negrito do exemplo. Só não devemos nos esquecer de fechar as tags, pois elas seguem o padrão xml.

O resultado disso será:

Tags com atributos

Quando fazemos uma tag, geralmente vamos querer passar atributos, para informar melhor o que queremos. Então, vamos incrementar a nossa dataDeHoje. Vamos informar para a tag qual o idioma no qual queremos ver a data escrita.

```
<%@tag description="descricao" pageEncoding="UTF-8" import="java.util.*,
java.text.*"%>
<%@attribute name="language" required="false"%>
<%
Locale locale = Locale.getDefault();
if (language != null) { //Testa se o language foi passado
locale = new Locale(language);
}
DateFormat formatador = DateFormat.getDateInstance(DateFormat.FULL, locale);
out.print(formatador.format(new Date()));
%>
```

Veja a diretiva **@attribute**. Ela é usada para definir um atributo da tag, sendo que cada tag pode conter vários atributos.

Um atributo precisa, necessariamente, ter um **name**. No nosso caso, o name é “language”. Podemos informar também o **required**. Se for true, a passagem deste atributo é obrigatória, e se for false, não é obrigatória (o default é false). Também pode ser informado o tipo, através de **type** (o default é String).

Há outras propriedades de @attribute, mas essas são as mais usadas.

Para acessar o atributos de dentro do tag file, como você deve ter percebido, usamos como identificador o name definido. Podemos também acessar os atributos por meio de EL (Expression Language). No nosso caso, seria **${language}**.

Agora, modificando a exibeDataDeHoje.jsp, teríamos:

```
<%@taglib prefix="mytags" tagdir="/WEB-INF/tags/mytags/"%>
<html>
<head> <title>Tutorial - TagFiles</title> </head>
<body>
<h1><mytags:dataDeHoje language="fr"/></h1>
```

```
</body>
</html>
```

Queremos a data em francês. Assim, o resultado seria esse:

Uma tag pode ter um corpo: parte compreendida entre a abertura e o fechamento da tag. O acesso ao corpo se dá pela tag **<jsp:doBody/>**. Como exemplo prático, faremos uma tag que, recebendo um título como atributo e um texto qualquer no corpo, gera um quadro. Chamaremos de **quadro.tag**.

```
<%@tag description="descricao" pageEncoding="UTF-8"%>
<%@attribute name="title" required="true"%>
<table>
<tr><td>
<fieldset>
<legend>${title}</legend>
<jsp:doBody/>
</fieldset>
</td></tr>
</table>
```

O <jsp:doBody/> acessa e escreve o conteúdo do corpo da tag. Agora, para testarmos, criaremos a pagina **testaQuadro.jsp**:

```
<%@taglib prefix="mytags" tagdir="/WEB-INF/tags/mytags/"%>
```

```
<html>
<head>
<title>Quadro</title>
</head>
<body>
<mytags:quadro title="Salgados">
<li> Pao de Queijo
<li> Coxinha
<li> Bolinha de queijo
<li> Enroladinho de presunto e queijo
<li> Kibe
<li> Croquete
<li> X Guloso
</mytags:quadro>
</body>
</html>
```

O resultado deverá ser algo assim:

Se quisermos manipular o conteúdo do corpo podemos, ao invés de escrevê-lo na

tela, armazená-lo em alguma variável, através dos atributos **var** e **scope** de <jsp:doBody/>. Dentro do tag file, faríamos:

```
<jsp:doBody var="texto" scope="request"/>
```

Nesse caso, gravaríamos o corpo na variável de requisição **texto**. Depois, se quiséssemos acessá-lo, usaríamos ${texto} ou request.getAttribute(“texto”).

Dependendo da sua tag, você pode querer que ela não aceite corpo. Para fazer isso, é simples:

```
<%@tag description="descricao" pageEncoding="UTF-8" body-content="empty"%>
```

Empacotando suas tags para distribuição

Se você quiser distribuir suas tags, para que outros usem, você deve empacotá-las em um arquivo JAR, formando assim uma biblioteca de tags. Aí, a pessoa que for usar coloca o jar em WEB-INF/lib e pronto.

Assim, vamos empacotar as nossas duas tags. Para fazer isso, devemos criar um arquivo **TLD** (Tag Library Descriptor), que ficará numa pasta chamada **META-INF**, e daremos o nome de **taglibs.tld** (qualquer outro

nome poderia ser dado).

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd">
<tlib-version>1.0</tlib-version>
<uri>taglibs</uri>
<tag-file>
<name>today</name>
<path>/META-INF/tags/mytags/dataDeHoje.tag</path>
</tag-file>
<tag-file>
<name>quadro</name>
<path>/META-INF/tags/mytags/quadro.tag</path>
</tag-file>
</taglib>
```

Dentro de taglibs.tld, podemos destacar:

<uri>taglibs</uri>: Especifica a uri da taglib, de forma que quando ela for usada em uma página, faremos <%@taglib prefix="mytags" uri="taglibs"%>.

<tag-file> ... </tagfile> : Aqui, declaramos as nossas tag files, informando o nome e o arquivo com caminho completo. O name definido será utilizado na hora de fazer uso da tag, como por exemplo:

```
<mytags:today/>
```

Agora, como você de ter concluído, colocamos os nossos tagfiles em **/META-INF/tags/mytags**.

Depois disso, é só criar o JAR, na linha de comando ou no seu IDE favorito. Criado o JAR, você já pode distribuir suas tags sem problemas.

JSTL

No ano 2000, o Java Comunity Process (JCP), selecionou um expert group para JSTL. Desde de então o expert group definiu a especificação da JSTL e produziu uma referência de implementação.

A JSTL é projetada para trabalhar com servlets containers com suporte as APIs Servlet 2.3 e JSP 1.2 ou acima.

A JSTL é uma coleção de custom tags que executam funcionalidades comuns em aplicações WEB, incluindo iteração e seleção, formatação de dados, manipulação de XML e acesso a banco de dados. A JSTL permite que os desenvolvedores JSP focalizem em necessidades específicas do desenvolvimento, ao invés de reinventar a roda.

A JSTL é composta de:

• Uma linguagem de Expressão

• Bibliotecas de Ações padrão (42 ações em quatro bibliotecas)

• Validators (2 validators)

Instalação da JSTL

Para que possamos utilizar os exemplos citados nesse tutorial, utilizaremos o pacote fornecido pelo projeto Apache Jakarta, o Taglib Standard que é a implementação de referência da JSTL.

Você pode baixar o pacote em http://jakarta.apache.org/taglibs/index.html

Nota:

JSTL 1.1 requer um container que suporte as especificações Servlet 2.4 e JSP 2.0

JSTL 1.0 requer um container que suporte as especificações Servlet 2.3 e JSP 1.2

Para utilizar a JSTL em suas aplicações, copie os arquivos *.jar* que estão dentro da pasta lib do pacote da JSTL para a pasta WEB-INF do seu projeto.

TLD

Há muitas recomendações para copiar os arquivos TLD para a pasta WEB-INF, para que possa se utilizar as tags. Mas uma boa prática é utilizar os TLDs que estão embutidos nos JARs do Taglibs Standard, dentro da pasta META-INF.

A Expression Language (EL)

A EL é uma simples linguagem baseada em ECMAScript (também conhecida como JavaScript) e XPath. Ela provem expressões e identificadores; aritiméticos, lógicos, operadores relacionais; e conversão de tipos.

A EL torna simples o acesso a objetos implícitos tal como o servlet request/response, variáveis de escopo e objetos armazenados no escopo JSP (page, request, session e application). A EL reduz drasticamente a necessidade de utilizar expressões JSP e scriplets, aumentando a manutenabilidade e extensibilidade de aplicações WEB.

Expressões EL são invocadas com essa sintaxe: *${expressão}*. As expressões podem consistir em:

Identificadores

Identificadores representam o nome dos objetos armazenados em um escopo JSP: *page*, *request*, *session* ou *application.*

Quando a EL encontra um identificador, ela procura por uma variável de escopo com o mesmo nome no escopo JSP, na ordem citada acima.

Por exemplo, o seguinte trecho de código armazena uma String em um escopo *page* e é acessada através de uma expressão EL:

```
<%
// cria a String
String s = “Portal Java”;
// armazena a String no escopo page
pageContext.setAttribute(“name”, s);
%>
```

E para a acessar a String com a expressão EL:

```
<c:out value='${name}'/>
```

Operadores

São estes os operadores da EL:

Tipo	Operador
Aritimético	+ - * / (div) % (mod)
Grupo	()
Identificador de Acesso	. []
Lógico	&& (and) || (or) ! (not) empty
Relacional	== (eq) != (ne) < (lt) > (gt) <= (le) >= (ge)
Unário	-

Precedência de operadores:

Os operadores são listados da esquerda para a direita. Por exemplo, o operador **[]** tem a precedência sobre o operador **.**

- [] .
- ()
- - (unary) not ! empty
- * / div % mod
- + - (binary)
- < > <= >= lt gt le ge
- == != eq ne
- && and
- || or =

Os operadores [] e .

A EL prove dois operadores que permitem você acessar variáveis de escopo e suas propriedades. O operador **.** é similar ao operador **.** do Java, mas em vez de acessar métodos, você acessará propriedades de Beans. Por exemplo:

Você tem um Bean chamado Pessoa armazenado em uma variável de escopo chamada pessoa e o Bean contém as propriedades nome e idade, você pode acessar essas propriedades assim:

```
Nome: <c:out value='${pessoa.nome}'/>
Idade: <c:out value='${pessoa.idade}'/>
```

O operador **[]** é utilizado para acessar objetos em maps, lists e arrays.

O seguinte código acessa o primeiro objeto em um Array:

```
<%
String[] num = { "1", "2", "3" };
pageContext.setAttribute("array", num);
%>

<c:out value='${array[0]}'/>
```

Valores Literais

São estes os valores literais da EL:

Tipo	Exemplo
Boolean	true ou false
Integer	143 +3 -4
Double	1.43 -2.35
String	strings com “” e ‘’
Null	null

Objetos implícitos

A característica mais útil da EL é o acesso a todos os objetos implícitos da aplicação:

Objeto	Tipo	Descrição	Valor
Cookie	Map	Cookie name	Cookie
Header	Map	Request header name	Request header value
headerValues	Map	Request header name	String[] of request header values
initParam	Map	Request header name	Initialization parameter value
param	Map	Request parameter name	Request parameter value
paramValues	Map	Request parameter name	String[] of request parameter values
pageContext	PageContext	--	--
pageScope	Map	Page-scoped attribute name	Page-scoped attribute value
requestScope	Map	Request-scoped attribute name	Request-scoped attribute value
sessionScope	Map	Session-scoped attribute name	Session-scoped attribute value
applicationScope	Map	Application-scoped attribute name	Application-scoped attribute value

Um dos objetos implícitos mais utilizado é o param, que acessa parâmetros do request.

Por exemplo:

Esse formulário apenas exibe dois campos a serem preenchidos e sua action aponta para a página *parametros.jsp* .

form.jsp

```
<html>
<head>
<title>Objetos Implícitos - EL</title>
</head>
<body>
<form action='parametros.jsp'>
<table>
<tr>
<td>Nome:</td>
<td><input type=”text” name=”nome”></td>
</tr>
<tr>
<td>Idade:</td>
<td><input type=”text” name=”idade”></td>
</tr>
</table>
<input type=”submit” value=”Enviar”>
</form>
</body>
</html>
```

parametros.jsp

```
<%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>
<html>
<head>
<title>Acessando Request Parameters com EL</title>
</head>
<body>
<font size='5'>Parâmetros</font>
<c:out value='${param.nome}'/>,
<c:out value='${param.idade}'/>
</body>
</html>
```

Utilizando uma biblioteca de tags

Há dois passos simples que devem ser seguidos para utilizar uma biblioteca de tags existente em um JSP:

• Disponibilize a biblioteca de tags para o JSP

• Utilizar a tag requerida no momento em que precisar

Importando a bibliotecas de tags

O processo é realmente muito simples:

Para importar a biblioteca de tags em um JSP, basta utilizar a diretiva taglib. .

```
<%@ taglib uri="" prefix="" %>
```

Detalhando os atributos:

• **uri** – permite especificar a localização do TLD (*tag lib descriptor*)

• **prefix** – é uma string que é utilizada unicamente para identificar as tags personalizadas que vc deseja utilizar.

TLD

É um arquivo XML que descreve o mapeamento entre as tags e seu tratador de tags.

Há duas maneiras básicas de especificar esse mapeamento. Uma opção é especificar um URL (relativo ou absoluto) para o arquivo TLD. A outra opção é fornecer um mapeamento de nome/URL lógico dentro do arquivo web.xml da aplicação.

Caso queira utilizar a primeira opção:

```
<%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
```

Uma desvantagem de utilizar uma URL para localizar um arquivo TLD torna-se aparente se o nome ou a localização do arquivo mudar. É por essa razão que a outra forma é preferida.

As URLs são muito comuns e, ao utilizar JSTL, você verá diretivas como essa:

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
```

Ações de Finalidades Gerais

A JSTL fornece quatro ações de finalidades gerais:

• <c:out>

• <c:set

• <c:remove>

• <c:catch>

As ações listadas acima representam ações fundamentais. Vamos detalhar cada uma das ações:

<c:out>

A sintaxe da ação <*c:out*> é:

```
<c:out value [default] [escapeXml]/>
```

Essa ação avalia uma expressão EL, converte o resultado para uma String e envia essa String para o JspWriter corrente.

Considere o seguinte trecho de código:

```
<%= request.getParameter("nome") %>
```

A ação <*c:out*> é o equivalente a expressão JSP citada acima:

```
<c:out value='${param.nome}'/>
```

O atributo *default* (opcional) especifica um valor padrão que deve ser utilizado quando a expressão é avaliada como *null* ou se a expressão não conseguir ser avaliada e lançar uma exceção.

Exemplo:

```
<c:out value='${param.nome}' default=”João Java da Silva” />
```

Caso o parâmetro nome não for passado, o valor que será exibido será:

João Java da Silva!!!

O atributo *escapeXml* (opcional) controla certos caracteres de XML que são convertidos em referências de entidade ou em referências de caractere. Por padrão, essa conversão acontece para os seguintes caracteres:

• Um caractere < é substituído por <
• Um caractere > é substituído por >
• Um caractere & é substituído por &
• Um caractere ‘ é substituído por '
• Um caractere “ é substituído por ".

O atributo *escapeXml* recebe um valor *boolean* (*true* ou *false*), por padrão o valor é *true*.

<c:set>

A ação <*c:set*> é completamente versátil, permitindo :

• Armazenar um valor em uma variável de escopo
• Apagar uma variável de escopo
• Ajustar a propriedade de um Bean a um valor especificado
• Ajustar a propriedade de um Bean para null
• Armazenar uma entrada (chave/par de valor) em um MAP Java
• Modificar uma entrada em um MAP Java
• Remover uma entrada em um MAP Java

A ação <*c:set*> suporta quatro sintaxes, duas dessas sintaxes permitem você manipular variáveis de escopo e as outras duas, permitem você manipular Beans e MAPs.

A primeira sintaxe da ação <*c:set*> é:

```
<c:set value var [scope]/>
```

Essa forma de ação <*c:set*> especifica o nome e o valor da variável exportada junto com o escopo opcional (por padrão assume o valor *page*).

A segunda sintaxe:

```
<c:set value var [scope]/>
conteúdo do corpo
</c:set>
```

Essa forma permite especificar o valor utilizando qualquer código válido JSP como conteúdo do corpo da ação *<c:set>*

Nas duas formas, o atributo *scope* é opcional e por padrão assume o valor *page*.

As outras duas formas da ação *<c:set>* são utilizadas para configurar as propriedades de objetos. Essa ação tem a mesma funcionalidade da tag *<jsp:setProperty>* que permite configurar os valores das propriedades de um Bean. Você também pode utilizar a ação *<c:set>* para esse propósito.

A sintaxe é:

```
<c:set target property value/>
```

e

```
<c:set target property>
conteúdo do corpo
</c:set>
```

Vamos a um exemplo de uso da utilização da tags *<c:set>* para configurar propriedades de um Bean.

Este trecho de código apenas ajusta a propriedade nome do Bean Usuário para o parâmetro passado:

```
<jsp:useBean id='usuario' class='beans.Usuario'/>
<c:set target='${usuario}' property='nome'
value='${param.nome}'/>
```

<c:remove>

Como podemos observar, a ação <*c:set*> permite a criação de variáveis de escopo. Algumas vezes é necessário remover essas variáveis.

Para essa funcionalidade a JSTL disponibiliza a ação <*c:remove*>

A sintaxe da ação <*c:remove*> é:

```
<c:remove var [scope]/>
```

Você precisa especificar o nome da variável de escopo que você quer remover com o atributo *var*. Opcionalmente você pode especificar a variável de escopo com o atributo *scope*.

Se você não especificar o atributo escopo, a ação <*c:remove*> irá buscar nos escopos por uma variável de escopo com o nome que você especificou no atributo *var*.

A ordem da busca nos escopos é:

- page,
- request
- session,
- application

<c:catch>

A ação <*c:catch*> disponibiliza um mecanismo de tratamento de erros.

A sintaxe da ação <*c:catch*> é:

```
<c:catch [var]>
conteúdo do corpo, com ações que podem lançar exceções
</c:catch>
```

Ações Condicionais

Ações condicionais são essenciais em qualquer linguagem de programação.

Elas foram projetadas para facilitar a realização de programação condicional no JSP e são também uma alternativa uma à utilização de scriplets.

A JSTL dispõe as seguintes tags condicionais:

- <c:if>
- <c:choose>
- <c:when>
- <c:otherwise>

Elas permitem realizar dois tipos diferentes de processamento: Processamento condicional simples e processamento mutuamente exclusivo.

<c:if>

A ação <*c:if*> realiza o processamento condicional simples. Ou seja, ela fornece uma condição de teste que é avaliada com um valor *boolean.* Se a ação for avaliada como um valor *true*, o conteúdo do corpo da ação <*c:if*> é processado. Caso contrário o conteúdo do corpo é ignorado.

O conteúdo do corpo das tags pode ser qualquer código JSP válido!

Vamos a sintaxe da tag:

```
<c:if test='condição' var='nome da variável' [scope='{page || request || session || application}'] />
```

O atributo *test,* que é requerido, é uma expressão *boolean* que determina quando o conteúdo do corpo da tag será processado.

Os atributos opcionais *var* e *scope* especificam a variável de escopo que armazenará o valor *boolean* da expressão especificada com o atributo *test*.

O seguinte trecho de código testa se um parâmetro é menor que 30, se o teste for true o parâmetro tamanho será exibido.

```
<c:if test='${param.tamanho < 30 }'>
<c:out value='${param.tamanho}'/>
</c:if>
```

Nota:

Não há nenhuma funcionalidade do tipo *if-then-else* fornecida pela ação <*c:if*>*.* Nos casos mais simples,você pode utilizar um número pequeno de ações <*c:if*> mutuamente para abranger os diferentes casos nos quais está interessado. Uma idéia melhor é utilizar a ação <*c:choose*>*.*

Condições Mutuamente Exclusivas

<c:choose>

Algumas vezes você precisa executar uma ação se uma de diversas circunstâncias for verdadeira.

Quando você especifica uma condição mutuamente exclusiva com JSTL, você utilizará a ação <*c:choose*>

A sintaxe da ação <*c:choose*> é:

```
<c:choose>
conteúdo do corpo
</c:choose>
```

O conteúdo do corpo da ação <*c:choose*> pode ser:

• Uma ou muitas ações <*c:when*>

• Nenhuma ação <*c:otherwise*>. Essa ação deve aparecer depois as ações <*c:when*>

• Caracteres de espaço em branco entre as ações <*c:when*> e <*c:otherwise*>

Você pode pensar sobre a ação <*c:choose*> como uma *if/else if/else if/else* em Java:

Código Java	Ações JSTL
if(condicao1)	<c:choose> <c:when test=”condicao1”> </c:when>
else if(condicao2)	<c:when test=”condicao2”> </c:when>
else	<c:otherwise> <c:otherwise> <c:choose>

A ação <*c:choose*> processa no máximo uma de suas ações aninhadas. A regra é que a ação <*c:choose*> processe a primeira ação <*c:when*> cuja condição de teste é avaliada como *true*. Entretanto, se nenhuma das ações <*c:when*> for aplicável, a ação <*c:otherwise*> é processada, se estiver presente. Se fornecer uma ação <*c:otherwise*>, você deverá certificar-se de que ela é a última ação dentro da ação <*c:choose*> pai.

<c:when>

Se você necessitar escolher entre mais de duas condições, você pode simular um *switch* simplesmente adicionando mais ações <*c:when*> no corpo da ação <*c:choose*>.

A sintaxe da ação <*c:when*> é:

```
<c:when test=''>
conteúdo do corpo
</c:when>
```

A ação <*c:when*> é similar a ação <*c:if*>, ambas as ações tem um atributo *test* que determina se o conteúdo do corpo da ação é avaliado.

Exemplo:

```
<c:when test='${sexo == 1}'>
<img src='<c:out value="masculino.jpg"/>'>
</c:when>
```

<c:otherwise>

A ação <*c:otherwise*> deve avaliar se nenhuma das suas ações <*c:when*> irmãs for avaliada como *true*. Ela deve ser a última ação dentro de uma ação <*c:choose*>.

A sintaxe da ação é <*c:otherwise*> é:

```
<c:otherwise>
conteúdo do corpo
</c:otherwise>
```

Exemplo:

```
<c:otherwise>
Olá usuário !!!
</c:otherwise>
```

O conteúdo do corpo pode ser qualquer código JSP válido.

A seguir, um exemplo de uso de algumas tags citadas.

Esse trecho de código apenas cria um formulário com os campos nome , sexo e um campo oculto chamado submitted que é utilizado para identificar se o form já foi submetido alguma vez.

```
<form name="form1" method="post" action="exemplo.jsp">
<table width="37%" border="1" cellpadding="0" cellspacing="0" bordercolor="WhiteSmoke">
<tr>
<td width="15%"><b>Nome:</b></td>
<td width="85%">
<input name="nome" type="text" value="<c:out value='${param.nome}'/>">
</td>
</tr>
<tr>
<td><b>Sexo:</b></td>
<td>
<select name="sexo">
<option></option>
<option value="1" <c:if test='${param.sexo == 1}'>selected</c:if>>Masculino</option>
<option value="2" <c:if test='${param.sexo == 2}'>selected</c:if>>Feminnino</option>
</select>
</td>
</tr>
<tr>
<td colspan="2">
<div align="center">
<input name="bt" type="submit" value="Enviar">
```

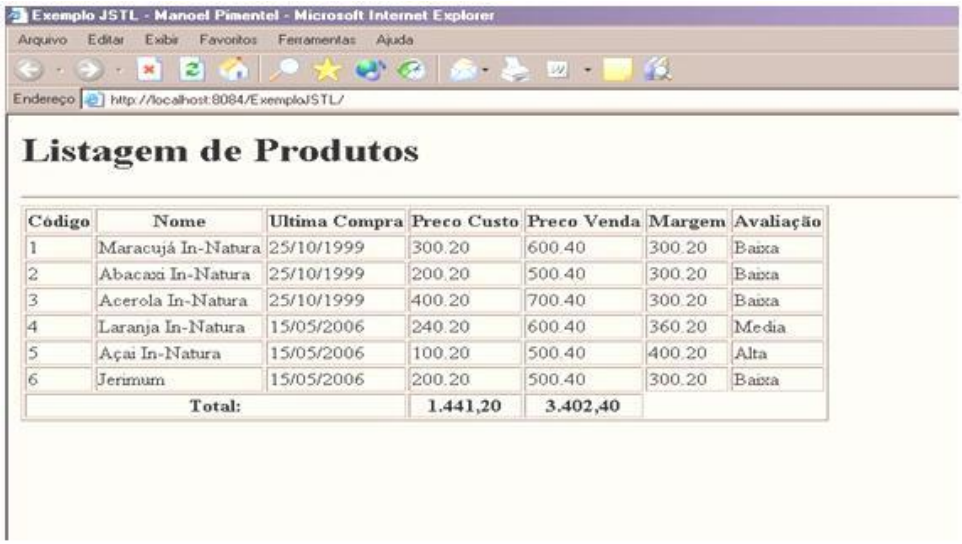
```
<input type="hidden" value="true" name="submitted">
</div>
</td>
</tr>
<tr>
<td colspan="2"><b>Nome: </b><i><c:out value='${param.nome}'/></i></td>
</tr>
<tr>
<td colspan="2"><b>Sexo:</b>
<c:if test='${param.sexo == 1}'>
<i>Masculino</i>
</c:if>
<c:if test='${param.sexo == 2}'>
<i>Feminino</i>
</c:if>
</td>
</tr>
</table>
</form>
```

Exemplo do uso de JSTL

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<!--taglib é uma tag JSP para declaração dos pacotes específicos JSTL
através da URI apropriada e da sugestão de prefixo padrão, lembrando que
essa etapa é importante, pois, definirá o prefixo de chamada das tags de
cada pacote.-->

<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>
    <head>
      <!-- setDataSource é uma tag JSTL para configuração de acesso ao
      servidor de banco de dados criando um objeto dataSource que será
      consumido pelas outras tags SQL,     onde devem ser definidos os
      atributos necessários para especificar: o nome da variável
      dataSource, driver, url(caminho do servidor), usuário, senha e
      principalmente o "escopo" de visibilidade do objeto na aplicação,
      em nosso caso, definimos que  nosso objeto será compartilhado por
      todo o tempo de    vida da sessão.-->

      <sql:setDataSource var="ds"
            driver="org.firebirdsql.jdbc.FBDriver"
            url="jdbc:firebirdsql:localhost:c:\ACADEMICO.FDB"
            user="sysdba"
            password="masterkey"
            scope="session" />

        <title>Exemplo JSTL - Manoel Pimentel </title>
    </head>

    <body>

        <h1>Listagem de Produtos</h1>
        <hr>
      <!--A tag query é usada para processar uma setença SQL de seleção
      de registros e gerar um objeto ResultSet internamente na memória,
      conforme especificado no atributo   var, usando a conexão aberta
      chamada "ds", que neste caso está definido no atributo dataDource
      através do uso de EL(Expression Language). -->

        <sql:query var="ResultadoProdutos" dataSource="${ds}">
            select * from PRODUTOS
        </sql:query>



            <table border="1">
```

```
<thead>
    <th>Código</th>
    <th>Nome</th>
    <th>Ultima Compra</th>
    <th>Preco Custo</th>
    <th>Preco Venda</th>
    <th>Margem</th>
    <th>Avaliação</th>
</thead>

<!--forEach, implementa um laço  para fazer a interação no
ResultSet  gerado pela tag query  conforme o atributo items.
-->

<c:forEach var="listaProdutos"
items="${ResultadoProdutos.rows}">
    <tr>
<!—
A tag out é responsável por gerar uma String de saída na tela
-->
        <td><c:out value="${listaProdutos.CODPRD}"/></td>
        <td><c:out value="${listaProdutos.NOME}"/></td>

        <td>
        <fmt:formatDate pattern="dd/MM/yyyy"
      Value="${listaProdutos.DATA_ULTIMA_COMPRA}"/>
        </td>
       <td><c:out value="${listaProdutos.PRECO_CUSTO}"/></td>
       <td><c:out value="${listaProdutos.PRECO_VENDA}"/></td>

      <c:set var="totalPrecoCusto"
      value="${totalPrecoCusto+listaProdutos.PRECO_CUSTO}"/>
      <c:set var="totalPrecoVenda"
      value="${totalPrecoVenda+listaProdutos.PRECO_VENDA}"/>
      <c:set var="valorMargem"
      value="${listaProdutos.PRECO_VENDA-
      listaProdutos.PRECO_CUSTO}"/>

        <td><c:out value="${valorMargem}"/></td>

      <!--As tags choose, when e otherwise,  aplicam um
      conjunto de  estrutura de decisão. -->




        <c:choose>
            <c:when test="${valorMargem<=350}">
                <td>Baixa</td>
            </c:when>
```

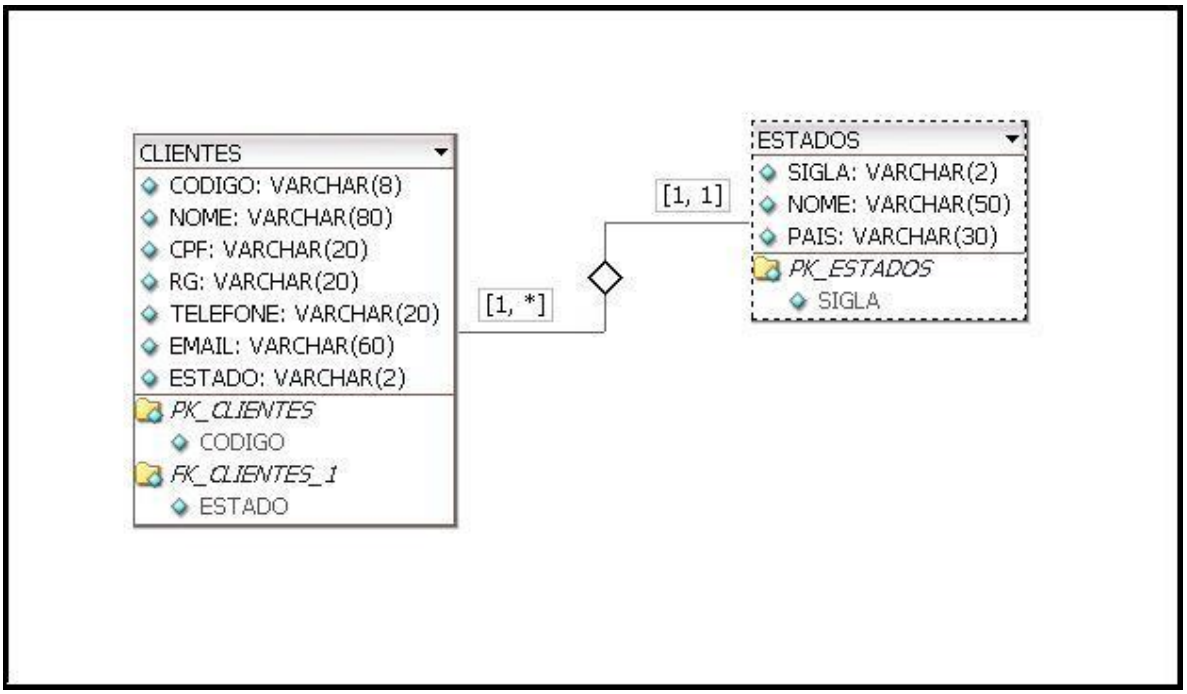
```
                        <c:when test="${valorMargem<=400}">
                            <td>Media</td>
                        </c:when>
                        <c:otherwise>
                            <td>Alta</td>
                        </c:otherwise>
                    </c:choose>
                </tr>
            </c:forEach>

            <tfoot>
                <th colspan="3">
                Total:
                </th>
                <th>
                  <!--formatNumber aplica uma formatação de decimais no
                  atributo value conforme o atributo pattern. -->
                  <fmt:formatNumber value="${totalPrecoCusto}"
                  pattern="#,#00.00#"/>
                </th>

                <th>
                  <fmt:formatNumber value="${totalPrecoVenda}"
                  pattern="#,#00.00#"/>
                </th>
            </tfoot>
        </table>
    </body>

</html>
```



Exemplo de Cadastro Web com JSTL

Nossa pequena aplicação terá 2(duas) páginas **JSP**, uma contendo um formulário com alguns campos para cadastro de cliente, e outra com o código **JSTL** para processamento dos dados enviados via requisição **POST** e gravação dos mesmos no banco de dados.

Para melhor aproveitamento do exemplo que será exposto neste artigo, observe na **Figura 01**, as tabelas CLIENTES e ESTADOS que servirão de base para nossa pequena aplicação.



Agora usando a IDE de sua familiaridade, crie um novo projeto web, de preferência usando o container TomCat(estou usando no exemplo a versão 5.5.17) e crie dois arquivos JSP, um chamado **index.jsp** contendo um formulário com os campos referentes ao cadastro simples de clientes(**ver listagem 02**) e outro chamado **gravaCliente.jsp**, que executar o inserção na tabela do banco de dados, usando os valores enviados pela página index.jsp por método POST (**ver listagem 03**).

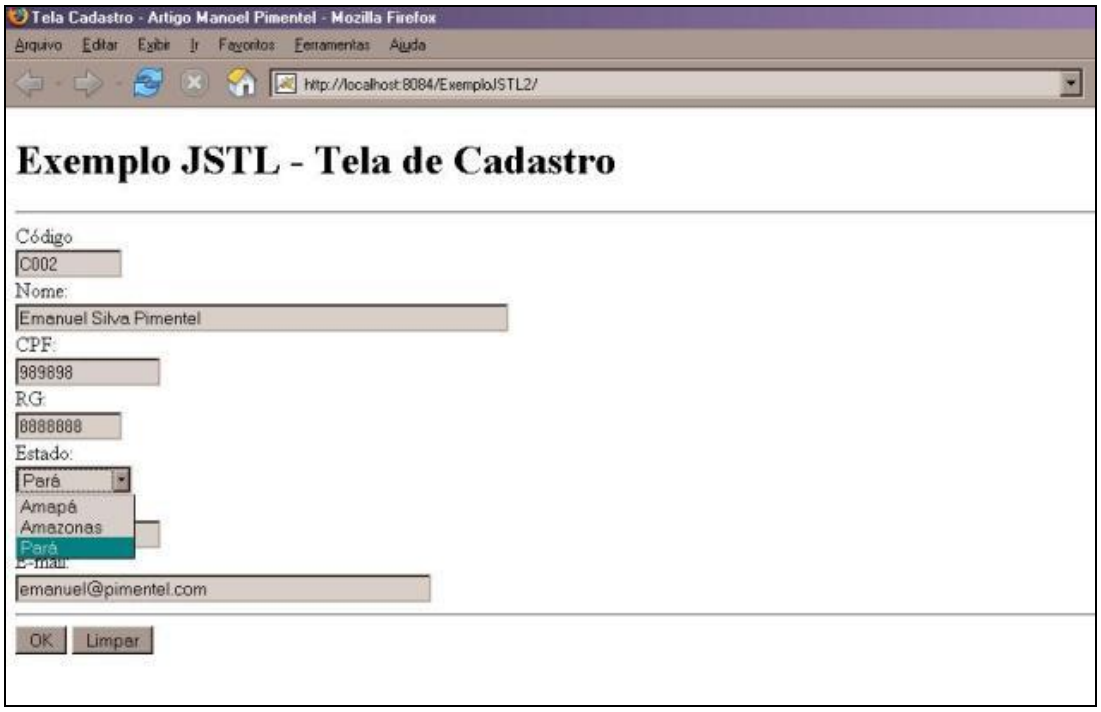
É importante lembrar que a tecnologia JSTL, é uma biblioteca de extensão da tecnologia JSP, por isso, criamos arquivos ".jsp" para que através de chamadas específicas das taglibs, passemos a usar os recursos oferecidos pela JSTL.

Note que os exemplos abaixo, estão comentados em formato web com os marcadores **<!--** e **-->**.

Index.jsp – Tela de Cadastro

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<!--Chamada aos TLD's de cada pacote JSTL -->
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>


<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
   <head>

      <!--Criação de um dataSource, que proverá uma conexão ao servidor de
banco dados, note que o escopo de conexão está definido como sendo de
sessão, dessa forma poderemos usar o mesmo objeto chamado "ds" de
qualquer outra página JSP participante do mesmo gerenciamento de sessão.-
->

      <sql:setDataSource var="ds"
                  driver="org.firebirdsql.jdbc.FBDriver"
                  url="jdbc:firebirdsql:localhost:c:\ACADEMICO.FDB"
                  user="sysdba"
                  password="masterkey"
                  scope="session"/>

      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Tela Cadastro - Artigo Manoel Pimentel</title>
   </head>
   <body>

      <h1>Exemplo JSTL - Tela de Cadastro</h1>
      <hr>
      <form action="gravaCliente.jsp" method="post">
         <label>Código </label><br>
         <input type="text" name="edtCodigo" size="10"/><br>
         <label>Nome: </label><br>
         <input type="text" name="edtNome" size="60"/><br>
         <label>CPF: </label><br>
         <input type="text" name="edtRG" size="15"/><br>
         <label>RG: </label><br>
         <input type="text" name="edtCPF" size="10"/><br>
         <label>Estado: </label><br>

               <!--Executa um comando SQL de seleção, gerando um objeto do
tipo Result, que  é semelhante a classe ResultSet da API  JDBC -->

         <sql:query var="qryEstados" dataSource="${ds}">
            select * from ESTADOS
            order by
```

```
            NOME
        </sql:query>

            <!--Cria um objeto select (estilho comboBox), preenchendo
suas opções com um laço forEach na coleção contida em "qryEstados.rows",
armazenando cada registro, na variável estado, e acessando a valor de um
determinado campo usando a EL(Expression Language)   ${estado.nome}  -->

        <select name="cmbEstados">
           <c:forEach var="estado" items="${qryEstados.rows}">
              <option value="PA">${estado.nome}</option>
           </c:forEach>
        </select><br>

        <label>Telefone: </label><br>
        <input type="text" id="edtTelefone" size="15"/><br>
        <label>E-mail: </label><br>
        <input type="text" name="edtEmail" size="50"/><br>
        <hr>
        <input  accesskey="o" type="submit" name="btnOK" value="OK">
        <input  accesskey="c" type="reset" name="btnCancelar" value="Limpar">
     </form>
  </body>
</html>
```

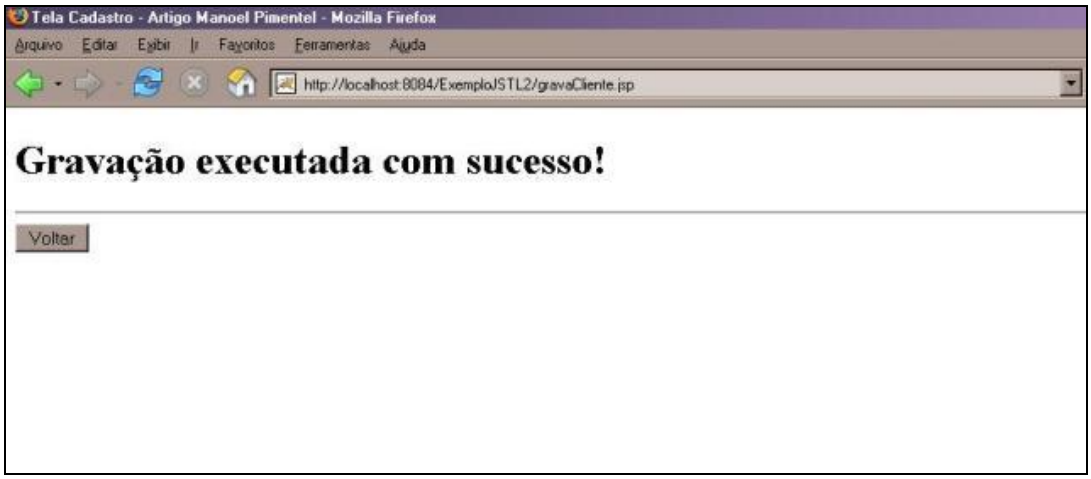


gravaCliente.jsp – Para gravação na tabela CLIENTES

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
```

```
<!--Chamada aos TLD's de cada pacote JSTL -->
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">


<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Tela Cadastro - Artigo Manoel Pimentel</title>
   </head>
   <body>

      <!--Equivalente ao try/catch, a tag catch tenta executar o que
estiver dentro de seu corpo, e caso ocorra alguma exceção, será capturada
e armazenada na variável "ex". -->

      <c:catch var="ex">

            <!--Cria uma transação com o banco dados, onde podemos
executar de forma mais protegida e seqüencial, várias atualizações ou
inserções. -->

         <sql:transaction dataSource="${ds}">

           <!--Executa algum comando como insert, update ou delete  e
armazena o resultado na varável "gravaCli".  Note que estamos usando a
tag sql:param para passar dinamicamente os valores em cada sinal de
interrogação da cláusula values, vale lembrar que dessa forma, estaremos
gerando uma espécie de sentença preparada, e dessa forma ganharemos
performance na execução do comando SQL.  -->

            <sql:update var="gravaCli">
               insert into CLIENTES (CODIGO,NOME,CPF,RG,TELEFONE,EMAIL,ESTADO)
               values(?,?,?,?,?,?,?)
               <sql:param value="${param['edtCodigo']}"/>
               <sql:param value="${param['edtNome']}"/>
               <sql:param value="${param['edtCPF']}"/>
               <sql:param value="${param['edtRG']}"/>
               <sql:param value="${param['edtTelefone']}"/>
               <sql:param value="${param['edtEmail']}"/>
               <sql:param value="${param['cmbEstados']}"/>

            </sql:update>
         </sql:transaction>
      </c:catch>


      <!--Essa é uma sacada legal, pois na tag “out” abaixo, caso o objeto “ex” esteja
nulo(ou seja, sem exceção), será exibido o valor contido no atributo default, dessa forma a
mensagem de sucesso só será exibida caso não tenha ocorrido  nenhuma exceção. -->
      <h1>
```

```
        <c:out value="${ex}" default="Gravação executada com sucesso!"/>
      </h1>
      <hr>
      <input type="button" value="Voltar" name="btnVoltar" onclick="history.back();"
    </body>
</html>
```



JavaServer Faces – JSF

JSF é uma tecnologia que incorpora características de um framework MVC para WEB e de um modelo de interfaces gráficas baseado em eventos. Por basear-se no padrão de projeto MVC, uma de suas melhores vantagens é a clara separação entre a visualização e regras de negócio (modelo).

A idéia do padrão MVC é dividir uma aplicação em três camadas: modelo, visualização e controle.

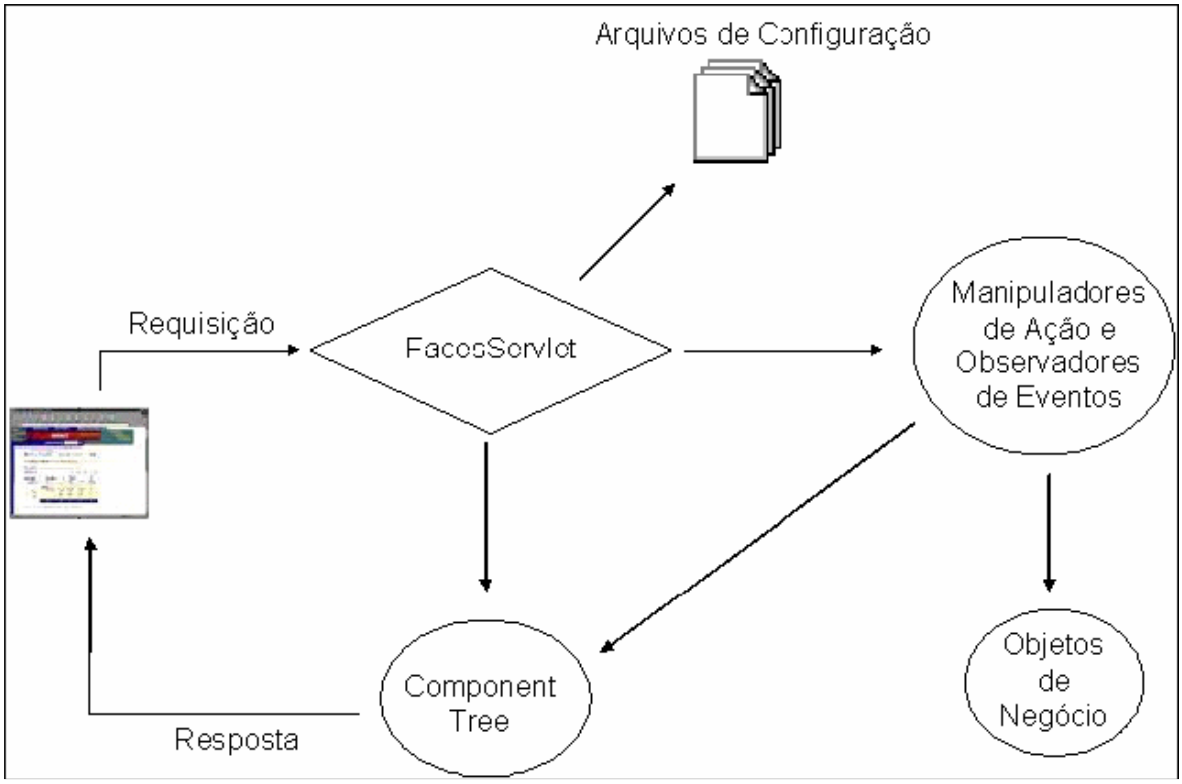
O modelo é responsável por representar os objetos de negócio, manter o estado da aplicação e fornecer ao controlador o acesso aos dados. A visualização representa a interface com o usuário, sendo responsável por definir a forma como os dados serão apresentados e encaminhar as ações dos usuários para o controlador. Já a camada de controle é responsável por fazer a ligação entre o modelo e a visualização, além de interpretar as ações do usuário e as traduzir para uma operação sobre o modelo, onde são realizadas mudanças e, então, gerar uma visualização apropriada.

No JSF, o controle é composto por um servlet denominado *FacesServlet*, por arquivos de configuração e por um conjunto de manipuladores de ações e observadores de eventos. O *FacesServlet* é responsável por receber requisições da WEB, redirecioná-las para o modelo e então remeter uma resposta.

Os arquivos de configuração são responsáveis por realizar associações e mapeamentos de ações e pela definição de regras de navegação. Os manipuladores de eventos são responsáveis por receber os dados vindos da camada de visualização, acessar o modelo, e então devolver o resultado para o *FacesServlet*.

O modelo representa os objetos de negócio e executa uma lógica de negócio ao receber os dados vindos da camada de visualização. Finalmente, a visualização é composta por *component trees* (hierarquia de componentes UI), tornando possível unir um componente ao outro para formar interfaces mais complexas.

A Figura 1 mostra a arquitetura do JavaServer Faces baseada no modelo MVC.



JavaServer Faces oferece ganhos no desenvolvimento de aplicações WEB por diversos motivos:

• Permite que o desenvolvedor crie UIs através de um conjunto de componentes UIs prédefinidos;

• Fornece um conjunto de tags JSP para acessar os componentes;

• Reusa componentes da página;

• Associa os eventos do lado cliente com os manipuladores dos eventos do lado servidor (os componentes de entrada possuem um valor local representando o estado no lado servidor);

• Fornece separação de funções que envolvem a construção de aplicações WEB.

Embora JavaServer Faces forneça tags JSP para representar os componentes em uma página, ele foi projetado para ser flexível, sem limitar-se a nenhuma linguagem markup em particular, nem a protocolos ou tipo de clientes. Ele também permite a criação de componentes próprios a partir de classes de componentes, conforme mencionado anteriormente.

JSF possui dois principais componentes: Java APIs para a representação de componentes UI e o gerenciamento de seus estados, manipulação/observação de eventos, validação de entrada, conversão de dados, internacionalização e acessibilidade; e taglibs JSP que expressam a interface JSF em uma página JSP e que realizam a conexão dos objetos no lado servidor.

É claro que existe muito mais a ser dito sobre JavaServer Faces. Esse artigo apenas fornece uma visão geral, mas espero ter criado uma certa curiosidade a respeito dessa nova tecnologia.

Implementando um exemplo com JSF

Será uma aplicação bem simples para demonstrar o uso dessa nova tecnologia. O nosso exemplo consiste de uma página inicial contendo 2 links: um para a inserção de dados e outro para a busca.

A página de inserção consiste de um formulário onde o usuário entrará com o nome, endereço, cidade e telefone. Os dados serão guardados em um banco de dados (no meu caso, eu uso o PostgreSQL) para uma posterior consulta. Se o nome a ser inserido já existir no banco de dados, uma mensagem será exibida informando ao usuário que o nome já está cadastrado (no nosso exemplo, o nome é a chave primária da tabela). Caso contrário, uma mensagem de sucesso será exibida ao usuário.

A busca se dará pelo nome da pessoa. Se o nome a ser buscado estiver cadastrado no banco, então uma página com os dados relativos ao nome buscado serão exibidos. Caso contrário, será informado ao usuário que o nome buscado não existe no banco.
Para a criação da tabela da base de dados foi utilizado o script apresentado abaixo.

```
CREATE TABLE pessoa
(
nome varchar(30) NOT NULL,
endereco varchar(50),
cidade varchar(20),
telefone varchar(10),
PRIMARY KEY (nome)
);
```

Index.jsf

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<head>
<title>Exemplo JSF</title>
</head>
<body>
<f:view>
<h:form>
<center>
<h1>Agenda</h1>
<br>
<h3>
<h:outputLink value="inserir.jsf">
<f:verbatim>Inserir</f:verbatim>
</h:outputLink>
<br><br>
<h:outputLink value="buscar.jsf">
```

```
<f:verbatim>Buscar</f:verbatim>
</h:outputLink>
</h3>
</center>
</h:form>
</f:view>
</body>
</html>
```

Algumas tags aqui merecem ser comentadas:

• <h:form> gera um formulário.

• <h:outputLink> cria um link para a página definida pelo campo *value*. O texto que compõe o link é colocado utilizando-se a tag <f:verbatim>.

O usuário terá a opção de buscar ou inserir dados. Os códigos das páginas de busca e inserção são mostrados a seguir, respectivamente.

Buscar.jsf

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<body>
<f:view>
<h:form>
<center><h2> Busca </h2></center>
<br>
Digite o nome:
<h:inputText id="nome" value="#{agenda.nome}"/>
<h:commandButton value="OK" action="#{agenda.buscar}"/>
</h:form>
<br>
<h:outputLink value="index.jsf">
<f:verbatim>voltar</f:verbatim>
</h:outputLink>
</f:view>
</body>
</html>
```

Inserir.jsf

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<body>
<f:view>
<h:form>
<center><h2> Inserção </h2></center>
<br>
<h3>Entre com os dados abaixo</h3>
<table>
<tr>
<td>Nome:</td>
<td>
<h:inputText value="#{agenda.nome}"/>
</td>
</tr>
<tr>
<td>Endereço:</td>
<td>
<h:inputText value="#{agenda.endereco}"/>
</td>
</tr>
<tr>
<td>Cidade:</td>
<td>
<h:inputText value="#{agenda.cidade}"/>
</td>
</tr>
<tr>
<td>Telefone:</td>
<td>
<h:inputText value="#{agenda.telefone}"/>
</td>
</tr>
</table>
<p>
<h:commandButton value="Inserir" action="#{agenda.inserir}"/>
</p>
</h:form>
<br>
<h:outputLink value="index.jsf">
<f:verbatim>voltar</f:verbatim>
</h:outputLink>
</f:view>
</body></html>
```

A tag <h:inputText> cria uma caixa de texto onde o valor digitado é guardado em *value*. Para criar um botão, é utilizada a tag <h:commandButton>. O label do botão é colocado em *value*. *Action* determina qual a ação que o botão deve tomar. Na página buscar.jsp, ao clicar no botão OK, o método buscar() da classe AgendaDB é chamado.

O código da classe bean **AgendaDB** com os métodos getters e setters das variáveis nome, endereço, cidade e telefone e com os métodos inserir e buscar ficará conforme mostrado abaixo. É nesse arquivo onde a conexão com o banco é feita. Nas aplicações JSF, os beans são usados para que dados possam ser acessados através de uma página. O código java referente a essa classe deve estar localizado no diretório JavaSource. Já que estamos utilizando um Java bean, um *managed-bean* deverá ser criado no arquivo *faces-config.xml* que está presente no diretório *WEB-INF*.

```
import java.sql.*;

public class AgendaDB
{
private String nome = blank;
private String endereco = blank;
private String cidade = blank;
private String telefone = blank;
private String result_busca = blank;
private String result_inserir = blank;
public static final String BUSCA_INVALIDA = "failure";
public static final String BUSCA_VALIDA = "success";
public static final String SUCESSO_INSERCAO = "success";
public static final String FALHA_INSERCAO = "failure";
static Connection con = null;
static Statement stm = null;
static ResultSet rs;
static private String blank = "";

public AgendaDB()
{
if (con==null) {
try {
Class.forName("org.postgresql.Driver");
con =
DriverManager.getConnection("jdbc:postgresql://localhost:5432/talita","ta
lita","tata");
} catch (SQLException e) {
System.err.println ("Erro: "+e);
con = null;
} catch (ClassNotFoundException e) {
System.out.println("ClassNotFound...");
e.printStackTrace();
}
}
}
public String getNome() {
return nome;
}
public void setNome(String nome) {
this.nome = nome;
}
public String getCidade() {
return cidade;
}
public void setCidade(String cidade) {
this.cidade = cidade;
```

```
}
public String getEndereco() {
return endereco;
}
public void setEndereco(String endereco) {
this.endereco = endereco;
}
public String getTelefone() {
return telefone;
}
public void setTelefone(String telefone) {
this.telefone = telefone;
}

public String inserir() {
String result_inserir = FALHA_INSERCAO;
try {
stm = con.createStatement();
stm.execute("INSERT INTO pessoa(nome,endereco,cidade,telefone) VALUES ('"
+ nome + "','"+
endereco + "','" + cidade + "','"+ telefone + "')");
stm.close();
result_inserir = SUCESSO_INSERCAO;
} catch (SQLException e) {
System.err.println ("Erro: "+e);
result_inserir = FALHA_INSERCAO;
}
return result_inserir;
}
public String buscar() throws SQLException {
String result_busca = BUSCA_INVALIDA;
try {
stm = con.createStatement();
rs = stm.executeQuery("SELECT * FROM pessoa WHERE nome = '" + nome +
"'");
if (rs.next()) {
nome = rs.getString(1);
endereco = rs.getString(2);
cidade = rs.getString(3);
telefone = rs.getString(4);
result_busca = BUSCA_VALIDA;
}
else
result_busca = BUSCA_INVALIDA;
rs.close();
stm.close();
} catch (SQLException e) {
System.err.println ("Erro: "+e);
}
return result_busca;
}
}
```

Ainda existirão mais 4 páginas JSP em nosso exemplo:

Sucesso_busca.jsf

esta página informa ao usuário que a busca foi bem sucedida, apresentando os dados referentes ao nome procurado:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<body>
<f:view>
<h:form>
<center><h2> Resultado da Busca </h2></center>
<br>
<table>
<tr>
<td>Nome:</td>
<td>
<h:outputText value="#{agenda.nome}"/>
</td>
</tr>
<tr>
<td>Endereço:</td>
<td>
<h:outputText value="#{agenda.endereco}"/>
</td>
</tr>
<tr>
<td>Cidade:</td>
<td>
<h:outputText value="#{agenda.cidade}"/>
</td>
</tr>
<tr>
<td>Telefone:</td>
<td>
<h:outputText value="#{agenda.telefone}"/>
</td>
</tr>
</table>
</h:form>
<br>
<h:outputLink value="index.jsf">
<f:verbatim>voltar</f:verbatim>
</h:outputLink>
</f:view>
</body>
</html>
```

Falha_busca.jsf

Esta página informa ao usuário que o nome buscado não existe no banco de dados:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
```

```
<html>
<body>
<f:view>
<h:form>
<h3>
<h:outputText value="#{agenda.nome}"/>
não foi encontrado(a)!
</h3>
</h:form>
<br>
<h:outputLink value="buscar.jsf">
<f:verbatim>voltar</f:verbatim>
</h:outputLink>
</f:view>
</body>
</html>
```

Sucesso_insercao.jsf

Informa que os dados foram inseridos com sucesso no banco:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<body>
<f:view>
<h:form>
Dados Inseridos com Sucesso!
</h:form>
<br>
<h:outputLink value="index.jsf">
<f:verbatim>voltar</f:verbatim>
</h:outputLink>
</f:view>
</body>
</html>
```

Falha_insercao.jsf

Informa que os dados não foram inseridos porque já existe no banco um nome igual ao que está se tentando inserir.

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<body>
<f:view>
<h:form>
<h3>
<h:outputText value="#{agenda.nome}"/>
já está cadastrado!Entre com outro nome!
```

```
</h3>
</h:form>
<br>
<h:outputLink value="inserir.jsf">
<f:verbatim>voltar</f:verbatim>
</h:outputLink>
</f:view>
</body>
</html>
```

Arquivos de Configuração

Finalmente faltam os arquivos de configuração *faces-config.xml* e *web.xml*. No arquivo *facesconfig.xml* nós vamos definir as regras de navegação e o managed-bean relativo à nossa classe AgendaDB.

O código está mostrado logo abaixo.

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
<navigation-rule>
<from-view-id>/buscar.jsp</from-view-id>
<navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/sucesso_busca.jsp</to-view-id>
</navigation-case>
<navigation-case>
<from-outcome>failure</from-outcome>
<to-view-id>/falha_busca.jsp</to-view-id>
</navigation-case>
</navigation-rule>
<navigation-rule>
<from-view-id>/inserir.jsp</from-view-id>
<navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/sucesso_insercao.jsp</to-view-id>
</navigation-case>
<navigation-case>
<from-outcome>failure</from-outcome>
<to-view-id>/falha_insercao.jsp</to-view-id>
</navigation-case>
</navigation-rule>
<managed-bean>
<managed-bean-name>agenda</managed-bean-name>
<managed-bean-class>AgendaDB</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>
</faces-config>
```

O código do arquivo *web.xml* pode ser visto abaixo.

```
<?xml version="1.0"?>
<!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>
<context-param>
<param-name>javax.faces.STATE_SAVING_METHOD</param-name>
<param-value>client</param-value>
</context-param>
<context-param>
<param-name>javax.faces.CONFIG_FILES</param-name>
<param-value>/WEB-INF/faces-config.xml</param-value>
</context-param>
<listener>
<listener-class>com.sun.faces.config.ConfigureListener</listener-class>
</listener>
<!-- Faces Servlet -->
<servlet>
<servlet-name>Faces Servlet</servlet-name>
<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
<load-on-startup> 1 </load-on-startup>
</servlet>
<!-- Faces Servlet Mapping -->
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>*.jsf</url-pattern>
</servlet-mapping>
</web-app>
```

A linha <url-pattern>*.jsf</url-pattern> presente no último bloco desse arquivo, indica que as páginas terão a extensão jsf. Por esse motivo que qualquer chamada de uma página dentro de outra a extensão não é jsp e sim jsf.