**Desenvolvimento de Aplicações com Banco de Dados**

David Buzatto

**Técnico em Informática para Internet**

**Instituto Federal de Educação, Ciência e Tecnologia de São Paulo – IFSP**

**São João da Boa Vista – SP**

**2011**

Indicação de ícones

Os ícones são elementos gráficos utilizados para ampliar as formas de linguagem e facilitar a organização e a leitura hipertextual.

|  |  |
| --- | --- |
|  | **Atenção:** indica pontos de maior relevância no texto. |
|  | **Saiba mais:** oferece novas informações que enriquecem o assunto ou “curiosidades” e notícias recentes relacionadas ao tema estudado. |
|  | **Glossário:** indica a definição de um termo, palavra ou expressão utilizada no texto. |
|  | **Mídias Integradas:** sempre que se desejar que os estudantes desenvolvam atividades empregando diferentes mídias: vídeos, filmes, jornais, ambiente AVEA e outras. |
|  | **Atividades de aprendizagem:** apresenta atividades em diferentes níveis de aprendizagem para que o estudante possa realizá-las e conferir o seu domínio do tema estudado. |

Sumário

Palavra do Professor-Autor

Caro estudante!

Neste caderno iremos trabalhar com o desenvolvimento de aplicações Web, o que em um primeiro momento pode parecer algo um pouco complicado, mas que ao passo que você for se acostumando com a forma de desenvolvimento, você verá que é algo relativamente fácil, desde que você se empenhe em ler atentamente o conteúdo do caderno, implementar os exemplos apresentados e desenvolver todos os exercícios propostos no final de cada aula, permitindo assim que você fixe, da melhor forma possível, o que foi apresentado e aprendido.

Atualmente, com o crescimento e a consolidação da Internet, o desenvolvimento desse tipo de aplicação computacional tem se tornando cada vez mais relevante, sendo assim, é muito importante que durante o desenvolvimento da disciplina, você interaja com seus colegas e com o seu tutor, procurando sempre solucionar seus problemas, não deixando que suas dúvidas, mesmo que você ache que sejam pequenas, lhe atrapalhem durante seu ciclo de aprendizado, fazendo com que você assimile o conteúdo da melhor forma possível e que possa aplica-lo em sua vida profissional.

Espero que você aproveite ao máximo o conteúdo disponibilizado nesse caderno, pois este foi elaborado com o objetivo de facilitar o seu aprendizado, sempre aplicando o que foi apresentado em situações práticas.

Estou muito feliz em poder lhe acompanhar durante essa disciplina! Desejo que tenha um ótimo aprendizado!

Prof. David Buzatto

Apresentação da Disciplina

Nesta disciplina será apresentada uma introdução sobre o desenvolvimento de aplicações Web utilizando banco de dados, sendo que utilizaremos a linguagem Java para o desenvolvimento de tais aplicações, além do MySQL, um SGBD (Sistema Gerenciador de Banco de Dados) relacional amplamente utilizado.

Durante a disciplina iremos aprender os fundamentos que alicerçam o desenvolvimento desse tipo de aplicações em Java, bem como alguns detalhes importantes de como as coisas funcionam nos bastidores, sendo estes fundamentais para o entendimento do fluxo de execução de aplicações Web.

Assim que aprendermos esses fundamentos, iremos colocá-los em prática, desenvolvendo assim uma pequena aplicação para controle de cadastro de clientes, indo desde a análise dos requisitos até a sua implementação. Ao final da disciplina, você será capaz de desenvolver um sistema sozinho, aplicando todos os conceitos e técnicas apresentadas durante o seu aprendizado e também será capaz de continuar seus estudos sobre o desenvolvimento de aplicações Web, aprendendo novas técnicas e tecnologias.

Projeto Instrucional

**Disciplina:** Desenvolvimento de Aplicações com Banco de Dados (carga horária: 60).

**Ementa:** Orientação a objetos e arquitetura cliente-servidor. Modelagem orientada a objetos. Modelagem de bancos de dados relacionais. Ambientes para gerenciamento de SGBDs (Sistemas Gerenciadores de Bancos de Dados) relacionais. Ambientes e linguagens de programação para desenvolvimento de aplicações Web. Conceitos de bases de dados distribuídas.

|  |  |  |  |
| --- | --- | --- | --- |
| **Aula** | **Objetivos** | **Materiais** | **Carga**  **Horária** |
| 1 – Java Para Web | Entender o funcionamento e a arquitetura de aplicações Web desenvolvidas em Java.  Entender o funcionamento dos *Servlets* e dos JSPs.  Aprender a utilizar a IDE NetBeans para o apoio ao desenvolvimento de aplicações Web em Java. | Caderno.  Computador.  *Software* JDK.  *Software* NetBeans.  *Software* Apache Tomcat. | 10 |
| 2 – Processamento de Formulários | Entender o funcionamento de formulários HTML.  Entender a diferença entre os métodos HTTP e como tratá-los. | Caderno.  Computador.  *Software* JDK.  *Software* NetBeans.  *Software* Apache Tomcat. | 10 |
| 3 – *Expression Language* e TagLibs | Entender a sintaxe e o propósito da *Expression Language*.  Aprender a utilizar a *Expression Language*.  Entender o propósito das *tags* JSP.  Aprender a utilizar as *tags* JSP.  Aprender a utilizar as *tags* disponibilizadas na JSTL (*JavaServer Pages Standard Tag Library*). | Caderno.  Computador.  *Software* JDK.  *Software* NetBeans.  *Software* Apache Tomcat. | 10 |
| 4 – Padrões de Projeto – *Factory*, DAO e MVC | Entender e aplicar o Padrão *Factory*.  Entender e aplicar o Padrão DAO.  Entender e aplicar o Padrão MVC.  Aprender a integrar banco de dados com uma aplicação Java. | Caderno.  Computador.  *Software* JDK.  *Software* NetBeans.  *Software* MySQL.  *Software* MySQL *Workbench*. | 10 |
| 5 – Primeiro Projeto – Sistema para Controle de Clientes | Entender e realizar a construção de uma aplicação Web em Java completa. | Caderno. Computador. *Software* JDK.  *Software* NetBeans.  *Software* Apache Tomcat.  *Software* MySQL.  *Software* MySQL *Workbench*.  *Software* astah\* UML. | 10 |
| 6 – Segundo Projeto – Sistema para Locação de DVDs | Aplicar o conhecimento adquirido na construção de uma aplicação Web em Java. | Caderno. Computador. *Software* JDK.  *Software* NetBeans.  *Software* Apache Tomcat.  *Software* MySQL.  *Software* MySQL *Workbench*.  *Software* astah\* UML. | 10 |

# Aula 1 – Java para Web

**Objetivos**

Entender o funcionamento e a arquitetura de aplicações Web desenvolvidas em Java.

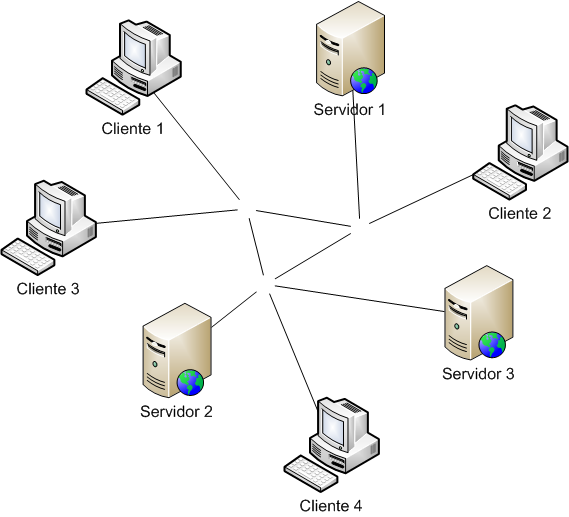
Entender o funcionamento dos Servlets e dos JSPs.

Aprender a utilizar a IDE NetBeans para o apoio ao desenvolvimento de aplicações Web em Java.

## Introdução

Para que você seja capaz de construir aplicações Web, primeiramente é preciso conhecer como esse serviço é estruturado. A *World Wide Web*, ou simplesmente Web, é um serviço executado em diversos computadores interligados em uma rede mundial, sendo que em alguns desses computadores são executados programas chamados de “servidores”, enquanto na maioria dos outros são executados programas chamados “clientes”, que se comunicam com os servidores, estes por sua vez servem recursos para estes clientes. Na Figura 1.1 é ilustrado um recorte desta rede mundial.

Figura .1: Recorte da estrutura da *World Wide Web*



Fonte: do autor

Perceba que no recorte apresentado na Figura 1.1, são mostrados sete computadores, sendo que quatro deles atuam como clientes e os outros três como servidores. É importante entender que o que faz um computador ser cliente ou servidor é o tipo de programa que está sendo usado/executado. No nosso exemplo, as máquinas que atuam como servidores executam um programa denominado Servidor Web, que tem a capacidade de servir (disponibilizar) aos outros computadores da rede, recursos que fazem parte de uma aplicação Web, por exemplo, arquivos HTML (*HyperText Markup Language*), imagens em diversos formatos, arquivos de estilo, arquivos de *script*, etc. Os clientes, por sua vez, são os conhecidos navegadores Web, ou *browsers*, que usamos no nosso dia a dia para acessar a Web e navegar em diversos *sites*.

<Saiba mais:

Atualmente existem diversos navegadores Web. Os mais famosos são o Microsoft Internet Explorer e o Mozilla Firefox. No link a seguir, está disponível uma lista da maioria dos navegadores Web já lançados. http://pt.wikipedia.org/wiki/Navegador

>

Da mesma forma que existem diversos navegadores, existem também alguns Servidores Web, sendo o Apache o mais famoso e o mais utilizado. Como já foi dito, um servidor Web tem a função de servir recursos requisitados pelos clientes. Vamos aprender como isso funciona. Veja a Figura 1.2.

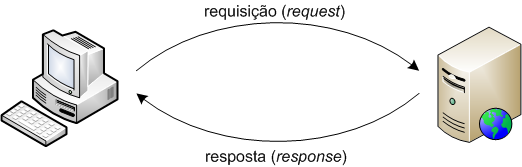
<Glossário: Apache

Detalhes sobre o Servidor Web Apache podem ser encontrados em:

http://projects.apache.org/projects/http\_server.html

>

Figura .2: Processo de requisição e resposta (*request*/*response*)



Fonte: do autor

Na Figura 1.2, é mostrado o processo de requisição e resposta. Neste processo, o cliente à esquerda (navegador), envia uma requisição a um recurso contido no Servidor Web (à direita) através de uma URL (*Uniform Resource Locator*), sendo que nessa requisição, o cliente especifica o protocolo a ser usado, o endereço do servidor e o caminho para o recurso. Sendo assim, uma URL tem a seguinte forma: protocolo://máquina/caminho\_do\_recurso, onde:

* **protocolo:** é a parte da URL que diz ao servidor qual o protocolo a ser utilizado. Quando acessamos páginas Web, o protocolo utilizado é o HTTP;
* **máquina:** é o nome ou o endereço IP da máquina que está executando o servidor Web;
* **caminho\_do\_recurso:** é o caminho completo do recurso desejado que é disponibilizado pelo servidor.

<Atividade de Aprendizagem:

Confuso? Nem tanto. Vamos a um exemplo! Imagine a seguinte situação: Queremos acessar o site do IFSP. Para isso, abra o seu navegador e preencha o campo endereço com “http://www.ifsp.edu.br/” (sem as aspas) e tecle <ENTER>. Fazendo isso, o navegador manda uma requisição através de uma URL, usando o protocolo HTTP para a máquina “www.ifsp.edu.br”, que por sua vez retorna ao navegador uma página HTML que representa aquele endereço.

Perceba que não especificamos o caminho do recurso! Isso não foi necessário, pois os servidores Web são normalmente configurados para ter um comportamento padrão para responder às requisições onde só seja especificado o nome da máquina e esse comportamento padrão é direcionar para o recurso index.html, que é um arquivo HTML. Portanto, usar o endereço “http://www.ifsp.edu.br/” é o mesmo que usar o endereço “http://www.ifsp.edu.br/index.html”. Faça um teste! Coloque o endereço com o caminho do recurso (índex.html) e tecle <ENTER>. O que aconteceu? A mesma página foi exibida não foi? Ótimo!

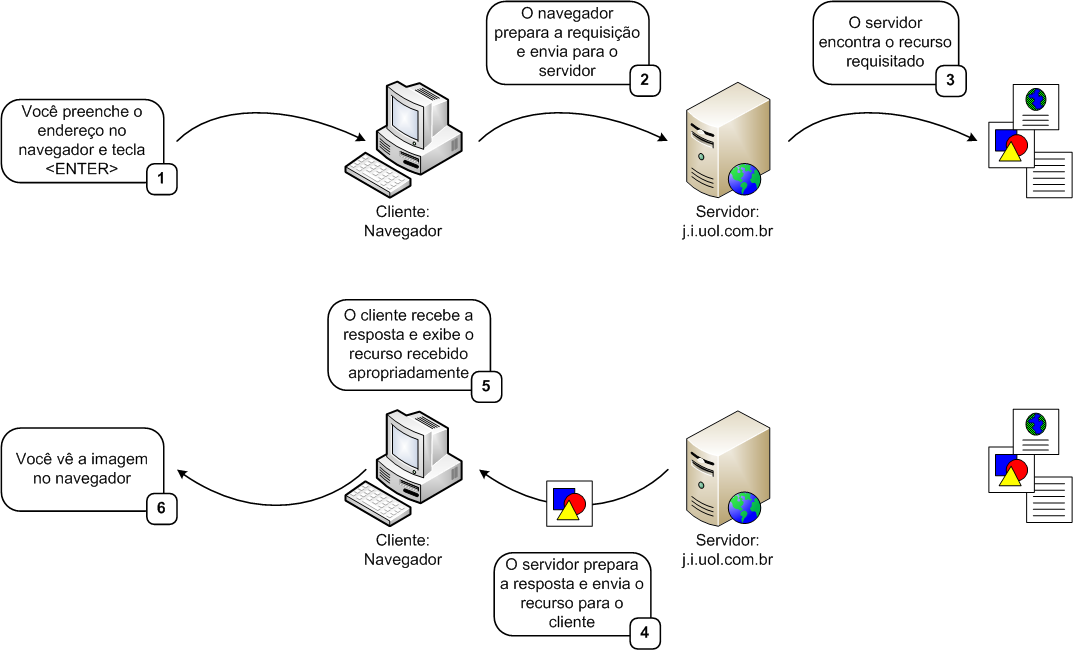
>

<Atividade de Aprendizagem:

Vamos fazer mais um teste? Preencha novamente a barra de endereços no seu navegador com o endereço “http://j.i.uol.com.br/galerias/psp/patapon201.jpg” (sem as aspas) e tecle <ENTER>. Deve ter aparecido uma imagem de um jogo não foi? Vamos analisar a URL então: usamos o protocolo HTTP, para pedir para o Servidor Web que está executando na máquina “j.i.uol.com.br” o recurso “patapon201.jpg”, que está armazenado no caminho “/galerias/psp/”. Este processo está ilustrado na Figura 1.3.

>

Figura .3: Processo de requisição e resposta a um recurso



Fonte: do autor

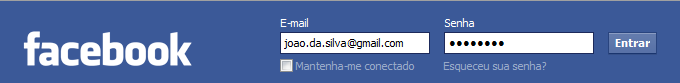
Perceba que os passos 1, 2 e 3 da Figura 1.3 representam o processo de requisição, enquanto os passos 4, 5 e 6 representam o processo de resposta. Note também que ao clicar em qualquer link em uma página, todo esse processo é executado pelo navegador.

Muito bem! Agora que você já sabe como funciona o processo de requisição e resposta entre um navegador (cliente) e um Servidor Web, podemos prosseguir com nossos estudos, agora focando em como a linguagem e a plataforma Java podem ser utilizadas para trabalhar com aplicações Web.

## Container de Servlets – Apache Tomcat

Na seção anterior você aprendeu algumas coisas novas, mas onde que a linguagem Java se encaixa nisso tudo? Quando criamos aplicações Web, além de termos páginas com código HTML e outros recursos como imagens, por exemplo, precisamos ter também componentes que processem informações que queremos que nossa aplicação manipule. Imagine que você vai fazer *login* na sua rede social favorita (Orkut, Facebook, etc.). Você preenche um campo com seu nome de usuário, sua senha e clica no botão para acessar o sistema e então eu lhe pergunto: O que acontece? Quem que vai validar seu usuário e sua senha? Como você deve saber uma página HTML não consegue fazer isso não é mesmo? Veja a tela de *login* do Facebook na Figura 1.4.

Figura .4: Tela de login da rede social Facebook



Fonte: *Print screen*, http://www.facebook.com

O responsável em processar os dados enviados é um componente de *software* que é executado no servidor em que a aplicação Web está implantada. Se a aplicação é feita em PHP, um script PHP vai fazer essa validação e retornar algum resultado com base no que foi verificado. Se a aplicação for feita em ASP acontece o mesmo, ou seja, algum componente vai tratar a requisição (que enviou o usuário e a senha) e vai validá-la. Em Java é a mesma coisa!

Dependendo da tecnologia que queremos utilizar na construção de nossas aplicações Web é preciso instalar os *softwares* necessários, permitindo então que a tecnologia escolhida possa ser utilizada. Como iremos aprender a utilizar a linguagem Java para criar aplicações Web, iremos usar um programa chamado “Container de Servlets”. Esse tal “Container” funciona como um Servidor Web simples, capaz de servir arquivos, da mesma forma que um Servidor Web convencional, mas ele tem um poder especial! Ele consegue executar classes compiladas em Java, além de fazer outras coisas muito legais que vamos aprender durante a disciplina.

Para que possamos utilizar código Java em nossas aplicações Web, recorremos a alguns componentes que podem ser criados dentro delas, sendo que o tipo principal desses componentes é o Servlet. Você se lembra do nosso exemplo do Facebook? Os desenvolvedores do Facebook, se usassem Java, poderiam ter criado um Servlet para receber os dados do formulário de *login*, processá-los e retornar alguma resposta para o cliente (navegador).

Muito bem. Sabemos então que se usarmos Java para desenvolver nossas aplicações Web, os componentes que são capazes de processar dados nas nossas aplicações e retornar resultados são os Servlets e que esses tais Servlets são executados em um programa chamado Container de Servlets, que funciona como um Servidor Web simples, mas com alguns “poderes” a mais.

Da mesma forma que existem navegadores e Servidores Web diferentes, adivinhe só, existem também Containers de Servlets diferentes! Iremos utilizar na nossa disciplina o Apache Tomcat (não confunda com o Apache que é servidor Web). Antes de prosseguirmos, vamos deixar uma coisa clara. Quando estamos no mundo “Java para Web”, várias dúvidas surgem o tempo todo, visto que existe uma infinidade de termos diferentes e que normalmente causam confusão. Uma coisa que deixa muitas pessoas confusas é a diferenciação entre um Container de Servlets e um Servidor de Aplicações.

<Glossário: Apache Tomcat

Detalhes sobre o Container de Servlets Apache Tomcat podem ser encontrados em:

http://projects.apache.org/projects/tomcat.html

>

Em um Container de Servlets são implementadas uma série de especificações que permitem que os desenvolvedores que forem trabalhar com ele usem uma determinada gama de funcionalidades, sendo que essas especificações são baseadas nos Servlets. Um Servidor de Aplicações é como se fosse um Container de Servlets com anabolizantes, pois além de implementar toda a especificação dos Servlets e as especificações ligadas a eles, esse tipo de servidor também implementa uma série de outras especificações da plataforma Java EE (*Java Enterprise Edition*) que fogem do propósito da nossa disciplina.

<Saiba mais:

Ficou curioso? Quer saber mais detalhes sobre o Java EE? Dê uma olhada na Wikipedia: http://pt.wikipedia.org/wiki/Java\_EE e no site da Oracle (dona do Java): http://download.oracle.com/javaee/ (em inglês)

>

O Apache Tomcat é capaz de executar tudo o que vamos precisar para trabalhar com nossas aplicações. Na próxima seção vamos aprender mais alguns detalhes teóricos e na última seção vamos realizar algumas atividades!

## Servlets e JSPs

Um carro serve para dirigir. Uma televisão para assistir. Uma sandália para andar. Cada um tem suas características e vão evoluindo com o tempo. Há muito tempo atrás, quando foi inventada a televisão, a imagem que era gerada não era muito boa e era em preto e branco. Foram passando os anos e a indústria foi evoluindo o aparelho. Primeiro a imagem melhorou, depois colocaram cores, depois foram criando telas cada vez maiores, mais finas, inventaram outras formas de emitir imagens, gastar menos energia elétrica e assim por diante. E daí?

Tudo evolui. O que não evolui é descartado e substituído. A primeira especificação dos Servlets foi lançada em 1997 e de lá para cá, a especificação foi evoluindo, permitindo que os Servlets se tornassem componentes mais versáteis e mais fáceis de utilizar. Na versão 6.0 do Tomcat, que iremos utilizar, é implementada a especificação 2.5 dos Servlets e a especificação 2.1 das JSPs (*JavaServer Pages*).

Já aprendemos que os Servlets são componentes de uma aplicação Web feita em Java e que têm a capacidade de processar dados enviados a eles e gerar respostas. Um Servlet pode gerar como resposta o código HTML de uma página, entretanto essa abordagem era utilizada nas versões mais antigas dos Servlets e é totalmente desencorajada hoje em dia. Como eu disse agora a pouco, tudo evolui. Para que os desenvolvedores não precisassem mais gerar código HTML dentro do código Java de um Servlet, foram inventadas as páginas JSP. Uma página JSP é um arquivo que pode conter – e geralmente contém – código HTML e que pode interagir diretamente com algumas funcionalidades de uma aplicação Web.

A rigor, um JSP é processado pelo Container de Servlets e todo o seu conteúdo é traduzido em um Servlet, que por sua vez é compilado e executado pelo Container. Todo esse processo é executado nos bastidores, então não precisamos nos preocupar com esses detalhes, mas é sempre bom saber um pouquinho como as coisas funcionam não é mesmo?

Por causa desse comportamento de tradução das JSPs para Servlets, nós podemos inserir código Java dentro das JSPs, mas novamente não iremos usar essa abordagem, muito menos irei ensinar como fazer, visto que da mesma forma que gerar HTML dentro de um Servlet manualmente é desencorajado, essa abordagem de inserir código Java dentro das JSPs também não deve ser utilizada. Uma JSP deve ser usada para exibir dados, não para processá-los diretamente usando código Java. Note que não estou dizendo que não iremos manipular dados dentro das JSPs, mas sim que existem formas seguras e corretas para fazer isso. Aprenderemos esses detalhes só na Aula 3, pois até lá, já teremos aprendido outros detalhes que ainda não foram apresentados.

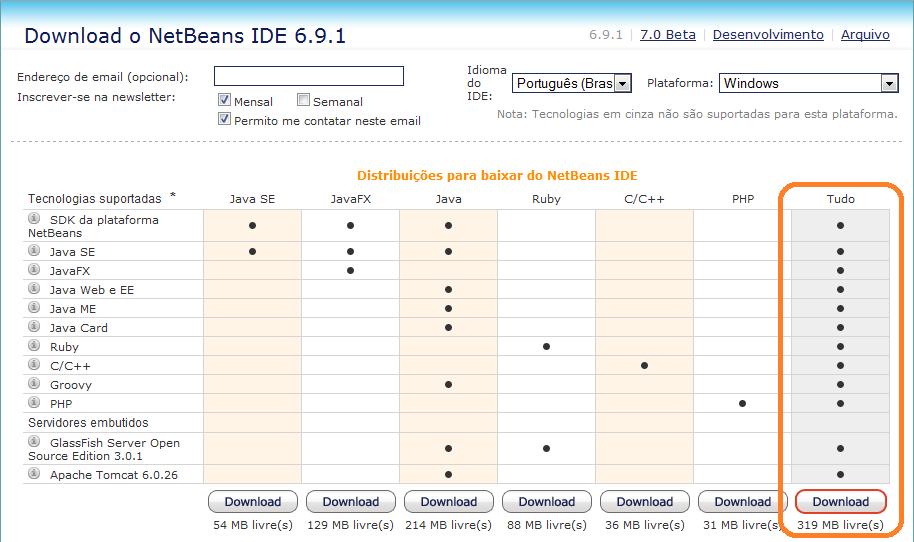
## NetBeans

Você conhece a linguagem Java, aprendeu a teoria e a prática da programação orientada a objetos e provavelmente usou o NetBeans para escrever seus códigos.

Se ainda não usou o NetBeans, não tem problema. Você vai aprender rapidinho, tenho certeza! Para termos um ambiente de desenvolvimento homogêneo, precisamos que nossas instalações no NetBeans sejam parecidas, pelo menos no que vamos usar na nossa disciplina. Vamos a um guia super-rápido de como baixar e instalar o NetBeans. Recomendo que mesmo que o NetBeans já esteja instalado na sua máquina, que você faça os passos a seguir. Sua instalação antiga vai ser apenas atualizada caso esteja faltando algum componente para ser instalado.

1. Acesse o endereço: http://netbeans.org/downloads/
2. Haverá várias opções de *download*. Procure pela maior, especificada como “Tudo”. É a da última coluna. Tem cerca de 319MB. Coloque o arquivo para baixar e vá tomar um café. Se tiver dúvida, dê uma olhada na Figura 1.5. Note que durante o desenvolvimento deste livro, a versão mais atual do NetBeans era a 6.9.1. Você pode baixar a mais recente caso esteja disponível.

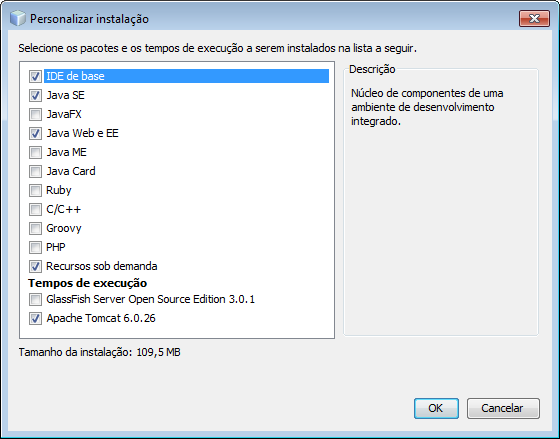
Figura .5: Baixando o NetBeans completo (http://netbeans.org/downloads/)



Fonte: *Print screen*, http://netbeans.org/downloads/

1. Baixou? Execute o instalador. Logo no primeiro passo, clique no botão “Personalizar...” (canto inferior esquerdo) para personalizar a instalação. Como iremos utilizar o Tomcat como Container de Servlets, selecione-o para que ele seja instalado. Na Figura 1.6, estão marcados os componentes que você OBRIGATORIAMENTE precisa instalar (IDE de base, Java SE, Java Web e EE, Recursos sob demanda, Apache Tomcat 6.0). Se quiser instalar outros componentes, fique à vontade.

Figura .6: Personalização da instalação do NetBeans



Fonte: *Print screen*, NetBeans IDE 6.9.1

1. Clique no botão “OK” do diálogo de personalização e vá clicando em “Próximo” até que a instalação termine. Ao terminar, feche o instalador. Um ícone do NetBeans vai ter sido criado na sua área de trabalho. Com isso terminamos nosso tutorial super-rápido de instalação do NetBeans.

Com o NetBeans instalado, abra-o. Caso o esteja executando pela primeira vez, uma página de boas vindas será exibida. Você pode fechá-la se quiser, além de marcar para que não seja exibida novamente. Vamos criar então nosso primeiro projeto Web!

* **Passo 1:** Clique no meu “Arquivo” e depois em “Novo Projeto”. Fazendo isso, o assistente para criação de projetos será aberto. Na lista de categorias, escolha “Java Web” e na lista de tipos de projeto, escolha “Aplicação Web” e clique no botão “Próximo”.
* **Passo 2:** Preencha o campo “Nome do projeto” com “OlaMundoWeb” (sem acentos e sem as aspas e tudo junto). Em “Local do projeto”, defina o diretório onde o projeto será salvo. Deixe a opção “Usar pasta dedicada para armazenar bibliotecas” desmarcada e marque a opção “Configurar como projeto principal”. Clique no botão “Próximo”.
* **Passo 3:** Na opção “Servidor” escolha o Apache Tomcat 6.0.xx. Em “Versão do Java EE” escolha “Java EE 5”. Em “Caminho do Contexto” deixe o valor padrão (/OlaMundoWeb), que é o mesmo nome que demos ao nosso projeto. Clique em “Próximo”.
* **Passo 4:** Neste passo, o assistente pergunta quais *frameworks* nós queremos inserir no nosso projeto. Nós não vamos usar nenhum, então basta clicar em “Finalizar”. Fazendo isso, o novo projeto será criado e será aberto no NetBeans, sendo que por padrão será criado um arquivo JSP (index.jsp) que será a página arquivo inicial da nossa aplicação.

<Glossário: Apache Tomcat 6.0.xx

Entenda “xx” como sendo qualquer número. No meu caso, a versão é a 6.0.26, mas na sua instalação ela pode variar caso a versão do NetBeans seja mais nova. O importante é escolher o Tomcat.

>

<Glossário: *Framework*

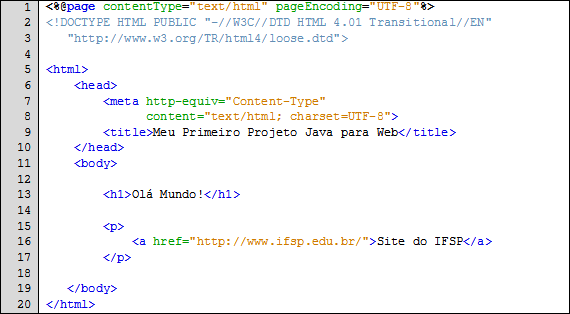
Um *framework* é um conjunto de classes que são usadas para executar uma determinada tarefa específica.

>

Muito bem, criamos nosso primeiro projeto. Vamos executá-lo para ver o que acontece? Na barra de ferramentas do NetBeans tem um botão com uma seta verde, igual a um botão de “play” de um CD Player. Quando você clicar nesse botão, você vai ver que várias mensagens começarão a aparecer na janela de saída do NetBeans. Essas mensagens irão mostrar para nós o que está acontecendo no momento, como a inicialização do Tomcat, por exemplo. O que está esperando? Clique lá no “play”. Assim que tudo estiver pronto, será aberta uma janela do seu navegador, onde será mostrado o conteúdo do index.jsp, que no nosso caso será uma página com um “Hello World!” escrito.

Muito bem! Temos nossa primeira aplicação rodando no Tomcat! Fácil não é mesmo? Por enquanto não vamos nos preocupar com a estrutura do projeto, iremos aprender os detalhes aos poucos. Vamos colocar um pouco de código HTML no nosso index.jsp? Ele deve estar aberto no NetBeans. Se não estiver, procure o arquivo index.jsp na pasta “Páginas Web” do seu projeto e clique duas vezes no arquivo para abri-lo no editor. Vamos mudar o título, escrever “Olá Mundo!” no lugar de “Hello World!” e inserir um link para a página do IFSP. Veja na Listagem 1.1 como ficou o meu código.

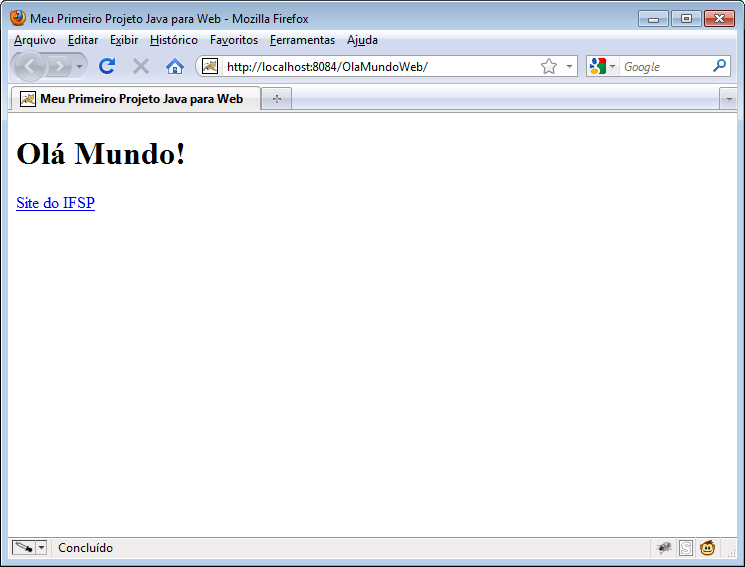
Listagem .1: Código do arquivo index.jsp



Fonte: do autor

Salve o arquivo depois de editá-lo. Se o navegador ainda estiver aberto no index.jsp, volte a ele e aperte a tecla F5 do seu teclado para mandar o navegador atualizar a página. Se não estiver, dê o “play” no projeto de novo. Você vai ver que a página vai exibir as alterações que fizemos. Teste o link para ver se está funcionando. A página deve ter ficado como mostrada na Figura 1.7.

Figura .7: Arquivo index.jsp em exibição



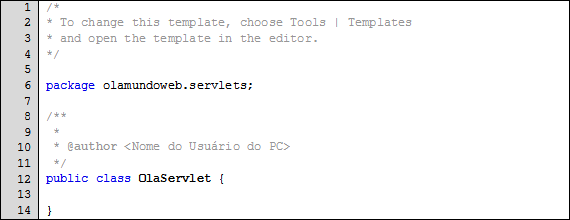
Fonte: *Print screen*, Windows 7

Uma JSP basicamente é um arquivo com código HTML que pode conter outros tipos de estruturas que vamos aprender no decorrer da disciplina. Por enquanto vamos ficar por aqui com as JSPs. Vamos testar os Servlets agora? Como primeiro exemplo, nós vamos criar um Servlet manualmente, enquanto os outros que vamos desenvolver durante o nosso curso serão feitos usando um assistente do NetBeans, mas essa forma fácil nós só vamos aprender a partir da Aula 2.

Aprenderemos como criar manualmente um Servlet, para que possamos aprender alguns detalhes importantes sobre o funcionamento de aplicações Web feitas em Java.

* **Passo 1:** Na árvore que representa a estrutura do projeto, procure pela pasta “Pacotes de código-fonte” e expanda-a (clique no sinal de “+” à esquerda). Dentro dela haverá um pacote com o ícone cinza chamado “<pacote padrão>”. Como vocês devem saber, é desencorajado que se trabalhe com pacotes padrão em Java, então vamos criar um pacote. Clique com o botão direito na pasta “Pacotes de código-fonte” e escolha “Novo”, procure pela opção “Pacote Java...” e clique nela. Se esta opção não estiver sendo exibida, clique na opção “Outro...” (no final da lista), escolha “Java” em “Categorias”, “Pacote Java” nos “Tipos de Arquivos” e clique em “Próximo”. Preencha o campo “Nome do pacote” com “olamundoweb” (sem as aspas) e clique em “Finalizar”. O pacote será criado.
* **Passo 2:** Repita o Passo 1, só que agora clicando com o botão direito no pacote que você criou e crie um pacote chamado “servlets” (sem as aspas). O nome do pacote deverá ser preenchido com “olamundoweb.servlets”. Seu projeto agora terá um pacote chamado “olamundoweb.servlets”.
* **Passo 3:** Clique com o botão direito no pacote “olamundoweb.servlets”, escolha “Novo” e clique na opção “Classe Java...”. Novamente, se não a encontrar, clique em “Outro...” e procure pela “Classe Java” (está na categoria “Java”) e clique em “Próximo”. Preencha o campo “Nome da classe” com “OlaServlet” (sem as aspas) e clique em “Finalizar”. A classe será criada dentro do pacote especificado e será aberta no editor. Você vai ter algo como apresentado na Listagem 1.2.

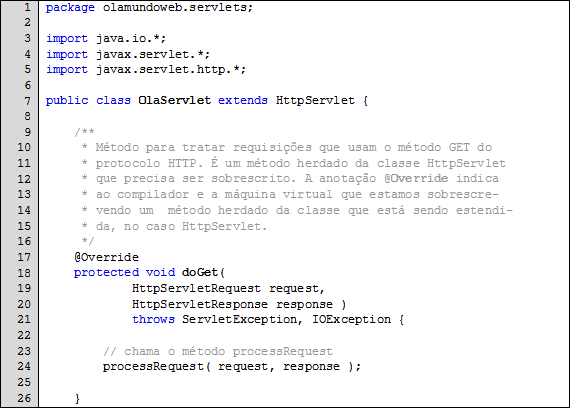
Listagem .2: Código da classe que será um Servlet



Fonte: do autor

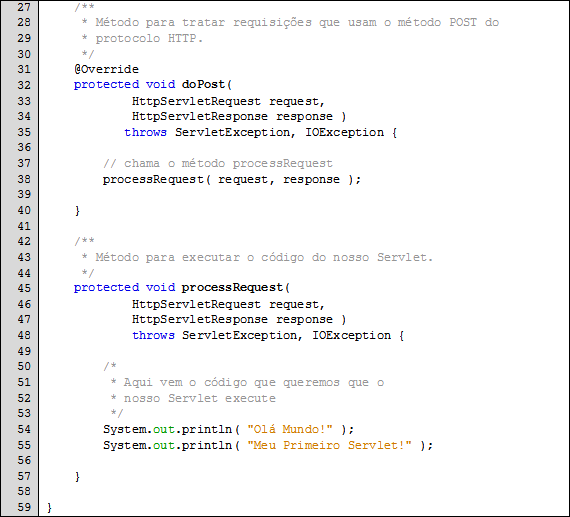
* **Passo 4:** Para que uma classe seja um Servlet, precisamos estender a classe HttpServlet, que está contida no pacote javax.servlet.http e então implementar os métodos HTTP que queremos que nosso Servlet trate. Não se preocupe, ainda vamos aprender sobre os métodos HTTP, então o importante, a saber, por enquanto, é que os métodos HTTP mais usados são o GET (doGet(...) de HttpServlet) e o POST (doPost(...) de HttpServlet). Então teremos que sobrescrever cada um desses métodos e ainda criaremos um terceiro que será invocado a partir dos outros dois. Confuso? Vamos ver como o código ficaria. Leia os comentários e copie o código para o seu editor!

Listagem .3: Implementação do primeiro Servlet (parte 1)



Fonte: do autor

Listagem .4: Implementação do primeiro Servlet (parte 2)



Fonte: do autor

Até agora criamos uma classe chamada OlaServlet, que estende a classe HttpServlet. Sobrescrevemos os métodos doGet e doPost herdados de HttpServlet que tratam respectivamente os métodos GET e POST do protocolo HTTP e criamos um terceiro método, chamado processRequest, que tem a mesma assinatura dos métodos doGet e doPost e que é invocado dentro deles. É no processRequest que iremos colocar o código que queremos executar, sendo que no nosso exemplo, estamos mandando imprimir na saída duas Strings: “Olá Mundo!” e “Meu Primeiro Servlet!”. Ou seja, se chamarmos o Servlet usando o método GET, o método doGet será invocado e passará o controle para o método processRequest que irá imprimir as mensagens na saída. O mesmo acontece para o método POST.

Muito bem, você tem um Servlet totalmente funcional, mas ai você se pergunta: “Como vou chamar esse Servlet através do navegador?”. Então eu respondo: “Precisamos mapeá-lo e dar um endereço para ele.”. Ai você diz: “Mas como?”. E eu respondo: “Siga os passos a seguir.”

* **Passo 1:** Na estrutura do projeto, procure por uma pasta chamada “Arquivos de configuração”. Adivinhe o que tem dentro dela? Arquivos de configuração é claro! O que nos interessa é o web.xml, que vamos chamar de DI (Descritor de Implantação), do termo em inglês *Deployment Descriptor*, e que é um arquivo XML. É no DI que ficam especificadas as configurações na nossa aplicação e é nele que vamos fazer o tal mapeamento que falei. O que está esperando? Abra-o no editor.

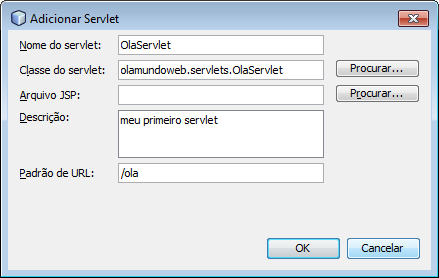
<Saiba mais:

Não sabe o que é XML? Dê uma olhada aqui: http://pt.wikipedia.org/wiki/XML

>

* **Passo 2:** Para nos auxiliar, o NetBeans tem um editor gráfico para o DI. Ao abrir o web.xml, perceba que existem várias guias (Geral, Servlets, Filtros, Páginas, ..., XML). Adivinhe em qual iremos clicar. Clique na guia “Servlets”. Ao clicar, procure pelo botão “Adicionar elemento servlet...” e clique nele. Um diálogo será aberto. Vamos preenchê-lo. Em “Nome do Servlet”, vamos preencher com o mesmo nome da nossa classe Servlet, ou seja, “OlaServlet” (sem as aspas). Em “Classe do servlet” precisamos apontar para a nossa classe. Clique em procurar, expanda “Pacotes de código-fonte” até encontrar o arquivo “OlaServlet.java”. Selecione-o e clique em “Selecione o arquivo”. Deixe o campo “Arquivo JSP” em branco. Preencha descrição com “meu primeiro servlet” (sem as aspas) e por fim preencha o campo “Padrão de URL” com “/ola” (sem as aspas e não esqueça de inserir a barra). Veja a Figura 1.8. Clique em OK e salve o arquivo.

Figura .8: Mapeando o primeiro Servlet



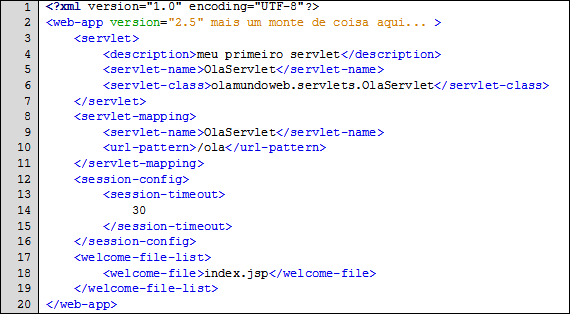
Fonte: *Print screen*, NetBeans IDE 6.9.1

No Passo 2 fizemos o mapeamento para amarrar o Servlet “OlaServlet” a um padrão de URL. Lembra que preenchemos o campo “Padrão URL” com “/ola”? Com isso, podemos agora acessar o OlaServlet a partir de uma URL, que no nosso caso é “http://localhost:8084/OlaMundoWeb/ola”, ou seja, usamos o protocolo HTTP, para a máquina localhost (que é o endereço da nossa máquina), na porta 8084 (que é a porta que o Tomcat ouve), para acessar a aplicação chamada OlaMundoWeb (isso vem do contexto que criamos no Passo 3 da Seção 1.4, volte lá para dar uma olhadinha), para por fim acessar o recurso mapeado sob o nome de “ola” que no caso é o nosso Servlet.

Sei que pode parecer um pouco confuso no começo, mas logo você vai pegar o jeito da coisa. Dê um “Play” no projeto de novo. O navegador vai abrir no endereço da aplicação novamente. Insira o “ola” (sem as aspas) no final da URL e tecle <ENTER>. O que aconteceu? Apareceu uma página em branco não foi? É claro, afinal, nosso Servlet não gera HTML, mas apenas imprime duas mensagens na saída padrão não é mesmo? Mas como podemos ver essas mensagens? Volte no NetBeans e procure, logo abaixo, uma aba chamada “Saída” (ela provavelmente vai estar selecionada). Dentro dela existem outras três abas: **OlaMundoWeb (run)** que deve estar selecionada e que é usada para mostrar o processo de compilação (na verdade não é só o processo de compilação que é exibido, mas sim todo o processo de construção do projeto) na nossa aplicação, **Log Apache Tomcat 6.0.xx** usada para mostrar os logs do nosso Container. Por fim, **Apache Tomcat 6.0.xx**, que exibe a saída padrão do Tomcat. Clique nessa última aba e veja o que está escrito lá embaixo. As duas mensagens que mandamos através do método println dentro do Servlet! Volte ao navegador e tecle <ENTER> novamente no endereço do Servlet. Volte no NetBeans. Mais duas mensagens! Fácil não é mesmo?

Para fechar a nossa primeira Aula, vamos voltar ao nosso DI (Descritor de Implantação). Abra novamente o web.xml e procure pela aba XML (a última). O web.xml é um arquivo XML que, como eu já disse, é responsável por manter todas as configurações da nossa aplicação. Apesar do NetBeans ter um editor gráfico para esse arquivo, sempre é bom entendermos, nem que seja um pouquinho, como as coisas funcionam nos bastidores. Ao clicar na aba XML será exibido o código do DI. Veja na Listagem 1.5.

Listagem .5: web.xml da aplicação



Fonte: do autor

O DI inicia com o preâmbulo padrão de um arquivo XML, <?xml...> e depois a primeira tag, chamada <web-app>, é utilizada. Essa é a tag principal do web.xml, que informa que esse arquivo é um DI. Dentro dela, temos primeiramente a tag <servlet>, que contém a descrição do Servlet, o nome dele e a classe Java correspondente. O que isso te lembra? Aquele diálogo que preenchemos no Passo 2 durante o mapeamento do Servlet (veja na Seção 1.4), não é? Após a tag <servlet>, onde o Servlet foi declarado, vem a tag <servlet-mapping> que é usada para mapear o Servlet. Veja que o nome definido dentro de <servlet-name> é o mesmo que está contido dentro da tag <servlet> e que dentro de <servlet-mapping> está declarado o mapeamento para “/ola”.

A tag <session-config> não é importante agora. A tag <welcome-file-list> é usada para especificar a lista de arquivos que devem ser utilizados como ponto de partida quando acessamos nossa aplicação. Lembra que eu falei que em um Servidor Web nós podemos definir um arquivo padrão para ser aberto? No Tomcat, que é nosso Container, isso é feito dentro dessa tag. Olhe só quem está lá! O index.jsp, que é a página inicial que é mostrada quando acessamos nossa aplicação pelo endereço http://localhost:8084/OlaMundoWeb.

Por mais que nosso exemplo não tenha nenhuma utilidade aparente, ele foi importante para nós entendermos o funcionamento básico de uma aplicação Web feita em Java. Nas próximas aulas vamos colocar o que aprendemos em prática, além de aprender várias outras coisas, para criarmos um sistema de cadastro na forma de uma aplicação Web. Não se esqueça de praticar o que aprendemos até agora.

Resumo

Nesta aula aprendemos o que é e como funciona uma aplicação Web em Java. Aprendemos a criar nosso primeiro projeto e alguns detalhes sobre a tecnologia que estamos utilizando. Executamos nossa aplicação e fizemos algumas modificações nela para vermos o que estava sendo feito. Criamos também – de forma manual – um Servlet, que como aprendemos é um dos componentes principais de uma aplicação Web em Java. Aprendemos também que existe um arquivo, denominado Descritor de Implantação – o web.xml – que descreve as características da nossa aplicação Web e onde fizemos o mapeamento do Servlet que criamos para uma URL válida dentro da nossa aplicação.

Atividades de Aprendizagem

1 – O que é um Servidor Web?

2 – Como são chamados os clientes que utilizamos para acessar aplicações servidas por um Servidor Web? Cite alguns exemplos.

3 – Diferencie um Servidor Web de um Container de Servlets.

4 – Crie um novo projeto Java Web no NetBeans, com o nome de “MinhaPagina”, edite o index.jsp de modo a exibir seus dados pessoais, seus interesses, etc. Tente inserir uma imagem também. Dica: a imagem deve estar dentro do projeto do NetBeans.

5 – Usando o exercício anterior, explique o motivo pelo qual o arquivo index.jsp é mostrado por padrão quando você acessa sua página através da URL HTTP://localhost:8084/MinhaPagina.

6 – Crie um novo projeto Java Web no NetBeans, com o nome de “Contador”. Nesse projeto você deve criar um Servlet manualmente e dentro do método processRequest use um laço de repetição (for) para direcionar para a saída padrão os números de 1 a 30. Não se esqueça de mapear o Servlet no DI e de verificar se ao acessar a URL criada os números são exibidos.

7 – Crie um novo projeto Java Web no NetBeans, com o nome de “Fibonacci”. Nesse projeto, você deve criar um Servlet manualmente e dentro do método processRequest use um laço de repetição (for) para exibir os 30 primeiros termos da série de Fibonacci. Crie um método chamado fibonacci dentro do seu Servlet, sendo que este método deve receber como parâmetro um int e retornar um int. O int que é recebido como parâmetro é o número do termo desejado, enquanto o int que é retornado é o termo correspondente ao parâmetro que foi recebido. A série de Fibonacci é formada inicialmente pelos números 1 e 1, sendo que os próximos números da série são gerados a partir da soma dos dois números anteriores. Os sete primeiros termos da série de Fibonacci são:

1, 1, 2, 3, 5, 8, 13, onde: **2** = 1 + 1, **3** = 1 + 2, **5** = 2 + 3, **8** = 3 + 5, **13** = 5 + 8

Exemplos de chamadas da função fibonacci:

fibonacci(2): retorna 1

fibonacci(5): retorna 5

fibonacci(7): retorna 13

# Aula 2 – Processamento de Formulários

**Objetivos**

Entender o funcionamento de formulários HTML.

Entender a diferença entre os métodos HTTP e como tratá-los.

## Introdução

Nesta aula vamos começar a aprender a criar algo útil! Na Aula 1, aprendemos as bases do desenvolvimento Web em Java, criamos alguns programas de brinquedo para aplicar as técnicas que aprendemos e agora vamos aprender mais alguns detalhes, mas dessa vez nossos programas serão mais elaborados. Vamos começar?

<Glossário: Programa de Brinquedo

Programa de brinquedo é todo programa criado para apresentar algum conceito, mas que não tem uma utilidade prática.

>

## Formulários

A forma tradicional de se desenvolver aplicações para Web que interajam com o servidor é utilizando os chamados formulários. Um formulário é composto normalmente por um conjunto de componentes que permitem que o usuário forneça dados para serem submetidos ao servidor. Quando esses dados são recebidos pelo servidor algum componente da aplicação vai tratá-los, sendo que no nosso caso, esse componente vai ser implementado na forma de um Servlet.

Atualmente existem diversas técnicas para a criação de aplicações Web, sendo que, dependendo da técnica/tecnologia, a forma de submeter dados ao servidor é diferente. Uma dessas técnicas é o chamado AJAX (*Asynchronous JavaScript and XML*). No nosso curso não iremos aprender a usar AJAX, porque infelizmente não teremos tempo, mas garanto que se você se empenhar no decorrer do nosso curso, ao final, você estará apto a aprender a trabalhar com AJAX.

<Saiba mais:

Quer saber o que é AJAX?

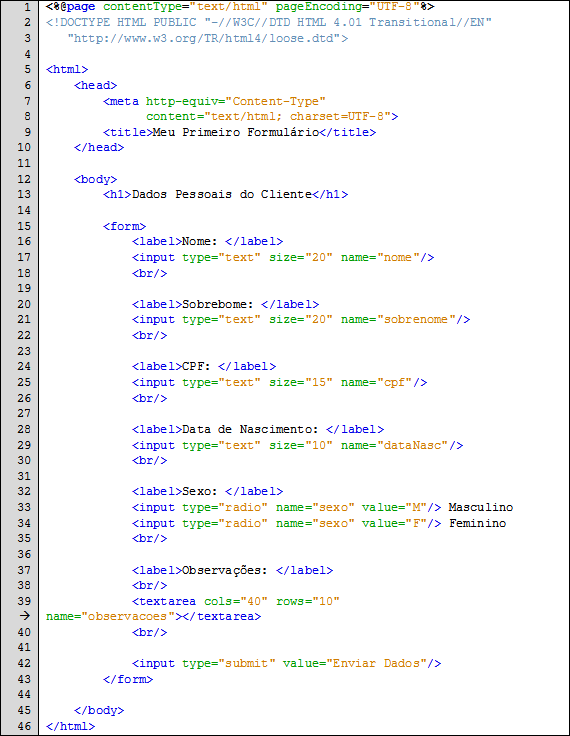
A definição do termo pode ser encontrada no endereço http://pt.wikipedia.org/wiki/AJAX\_(programação)

Um tutorial (em inglês) pode ser visto no endereço http://www.w3schools.com/ajax/default.asp

>

Abra o NetBeans e crie um novo projeto Java Web. Dê o nome de “PrimeiroFormulario” (sem as aspas). Sempre que for criar um novo projeto, siga os passos descritos na Aula 1. Com o projeto criado, vamos editar o index.jsp. Nele vamos alterar o título da página e criar nosso primeiro formulário, que será usado para preenchermos dados pessoais de um cliente. Veja na Listagem 2.1 como ficou o código. Não se esqueça de copiá-lo no seu index.jsp.

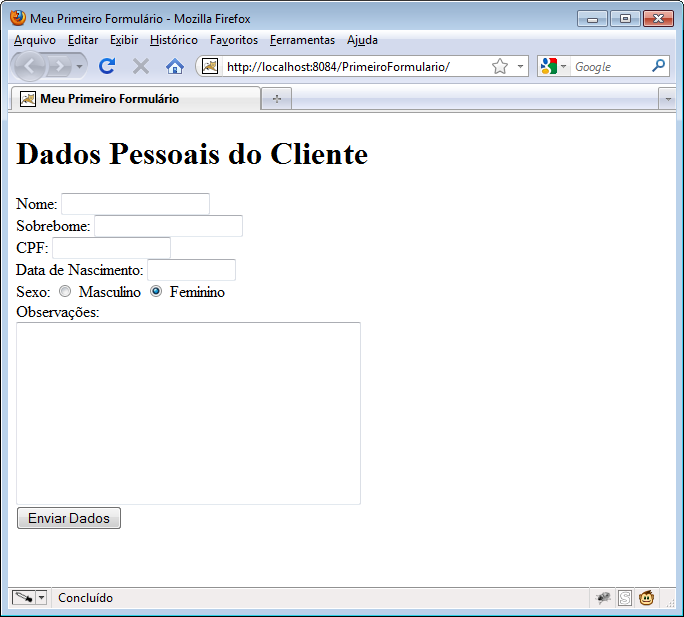
Listagem .1: Protótipo do formulário de dados do cliente



Fonte: do autor

Copiou? Salve o arquivo e execute a aplicação. Você vai ter algo como o mostrado na Figura 2.1.

Figura .1: Visualização do protótipo do formulário de dados do cliente



Fonte: *Print screen*, Windows 7

Quanta coisa! O formulário não ficou uma obra prima, mas esse não é nosso objetivo agora. Precisamos entender o que cada *tag* faz. Vamos agora analisar o código da Listagem 2.1 e entender o protótipo que fizemos. Irei detalhar apenas as *tags* <form> e seus componentes, pois acredito que você já conheça as outras que foram utilizadas. Vamos lá então:

* **Linha 15:** Nesta linha abrimos a *tag* *form* que delimita um formulário HTML. Note que fechamos a *tag* *form* na linha 43. Todas as *tags* que forem inseridas entre <form> e </form> farão parte do formulário.
* **Linha 16:** Criamos um *label* (*tag* <label>) com o conteúdo “Nome: ”. A *tag* *label* é usada para criar um rótulo. Ao invés de usar a *tag* *label*, poderíamos simplesmente ter inserido o texto que queremos mostrar no formulário, mas como o texto que estamos utilizando tem o propósito de ser um rótulo para um campo do formulário, iremos utilizar a *tag* *label* para deixar nosso código mais organizado.
* **Linha 17:** Criamos um *input* (campo de entrada) do tipo *text* (texto) com tamanho de 20 colunas e com o nome de “nome”. Dentre as *tags* que representam componentes nos formulários, a *input* é uma delas. Existem vários tipos de *inputs* (diferenciados pela propriedade *type*) e que vamos aprender aos poucos. A propriedade *size*, como você deve ter percebido, é utilizada para configurar a largura do campo de texto. A propriedade *name* é utilizada pelo navegador para enviar os dados do componente para o servidor. Não entendeu a utilidade da propriedade *name*? Não se preocupe, logo vai fazer sentido.
* **Linha 18:** Usamos a *tag* *br* para pular uma linha.
* **Linha 20:** Os próximos três campos (sobrenome, CPF e data de nascimento) são bem parecidos com o primeiro.
* **Linha 33:** Criamos um *input* do tipo *radio* (botão de rádio), com nome configurado como “sexo” e com o valor (*value*) configurado com “M”.
* **Linha 34:** Idem a linha anterior, com a diferença que o valor é “F”. Note que a propriedade *name* de ambos os *radios* é a mesma, pois eles representam o mesmo campo (sexo). Perceba que no navegador, se você selecionar um deles e depois clicar no outro, o que estava selecionado previamente deixa de ser selecionado. Se a propriedade *name* for diferente, eles serão considerados campos diferentes e então esse comportamento da seleção não existirá.
* **Linha 39:** Nessa linha definimos uma área de texto. Esse componente, representado pela *tag* *textarea* é utilizado, como o próprio nome já diz, para criar uma área de texto livre, onde o usuário poderá digitar uma quantidade arbitrária de texto. Note que para utilizar um *textarea* nós precisamos usar a *tag* de fechamento (<textarea>), ao invés de fazer da forma que estamos fazendo com os *inputs*. A novidade nesse componente são as propriedades *cols* e *rows*, que são usadas respectivamente para definir a quantidade de colunas e de linhas do componente.
* **Linha 42:** Por fim, nessa linha definimos um *input* do tipo *submit*, que é um botão que tem o comportamento padrão de, ao ser clicado, submeter (enviar) os dados do formulário para o servidor. Note que usamos a propriedade *value* para definir o texto do botão.

Agora que já conhecemos alguns dos componentes que podemos utilizar nos nossos formulários, mas você deve estar se perguntando: “Tudo bem, o *input* do tipo *submit* é usado para enviar os dados do formulário para o servidor, mas onde eu digo ao *submit* para onde os dados do formulário devem ser enviados?”. Vamos à resposta!

Eu tenho dito várias vezes que o componente que vai tratar os dados de um formulário na nossa aplicação é o Servlet não é mesmo? Então precisamos criar um Servlet que vai receber esses dados e então configurar o formulário para direcionar os dados inseridos nele para o Servlet apropriado.

Vamos criar o Servlet? Mas agora iremos fazer de uma forma mais automática do que a que estamos fazendo desde que aprendemos a trabalhar com os Servlets. Na pasta de pacotes de código-fonte do projeto, crie um pacote chamado “primeiroformulario.servlets” (sem as aspas). Clique com o botão direito no pacote criado, escolha “Novo” e procure pela opção “Servlet...”. Se não encontrou, escolha a propriedade “Outro...”, selecione “Web” na categoria, “Servlet” no tipo de arquivo e clique em “Próximo”.

Preencha o campo “Nome da classe” com “ProcessaDadosClienteServlet” (sem as aspas) e clique em “Próximo”. Nesse passo, note que o assistente nos pede o nome do Servlet e o padrão de URL. Lembra que criávamos nossa classe do Servlet e implementávamos seu esqueleto, além de fazer seu mapeamento no DI, manualmente? Agora o NetBeans vai fazer tudo isso para nós automaticamente! Deixe o campo “Nome do Servlet” com o valor padrão (que é o nome da classe) e preencha o campo “Padrão(ões) de URL” com “/processaDadosCliente” (sem as aspas). Não iremos aprender sobre os padrões de inicialização na nossa disciplina, mas nada impede que você aprenda para que eles servem, basta consultar a bibliografia recomendada nas referências bibliográficas da apostila tudo bem? Tudo feito? Clique em “Finalizar”.

Ao fazer isso, o nosso Servlet será criado e aberto no editor. Como eu sei que você é uma pessoa super curiosa, dê uma olhada no web.xml para ver que o Servlet que foi criado já foi mapeado lá! Legal hein?

Note que o NetBeans já implementou o esqueleto do Servlet para nós. O primeiro método implementado é o processRequest. Lembra-se dele? É nele que vamos inserir o código que queremos que o Servlet execute. Após o fechamento do bloco deste método, note que existe uma linha onde está escrito “*HttpServlet methods. Click on the + sign on the left to Edit the code.*”. Siga a sugestão da frase e clique no “+”. O que apareceu? A implementação dos métodos GET e POST, sendo que dentro delas é chamado o processRequest! Viu só? Da mesma forma que fazíamos manualmente! Não iremos mexer ali, então você pode contrair novamente esta seção do código clicando no sinal de “–”.

Note que além de implementar o esqueleto do nosso Servlet, o NetBeans também inseriu um trecho de código dentro do processRequest. O código que está inserido configura o tipo de retorno do Servlet (response.setContentType(...)), obtém o canal de escrita do Servlet, escreve uma série de Strings que representam uma página HTML nesse canal e o fecha. Você se lembra que já falei algumas vezes que não iremos implementar Servlets que geram código HTML? Então, vamos limpar esse método, tirando todo o código que foi inserido dentro dele. Vá no editor e apague o conteúdo entre as linhas 30 (response.setContentType(...)) e 45 (“}”), incluindo elas. Seu processRequest deve estar vazio agora.

Antes de codificarmos, vamos a mais um pouquinho de teoria. Você se lembra, lá no comecinho da primeira aula, que eu falei que o cliente manda uma requisição para o servidor e ele manda uma resposta? Nos Servlets, essa requisição e essa resposta são representadas respectivamente por objetos do tipo HttpServletRequest e HttpServletResponse. Note que os três métodos implementados nos nossos Servlets (processRequest, doGet e doPost) recebem dois parâmetros, sendo eles dos tipos que mencionei. Qual a conclusão que você chega então? Os dados enviados pelo cliente ao servidor estão dentro do objeto do tipo HttpServletRequest, que chamamos de *request* no nosso código e os dados que enviamos de volta ao cliente devem ser inseridos no objeto do tipo HttpServletResponse, que demos o nome de *response*.

Sabendo disso, agora ficou fácil! Os dados do formulário de clientes que estamos construindo serão recebidos pelo nosso Servlet através do objeto apontado por *request*! Legal não é? Mas ainda falta um detalhe... Não informamos ao formulário para quem ele deve enviar os dados! Vamos fazer isso agora. Volte no index.jsp e procure pela *tag* *form* (a mesma que está na linha 15 da Listagem 2.1). Para informarmos à *tag* *form* qual é o destino dos seus dados, utilizamos a propriedade *action*, sendo que nesta propriedade, colocamos a URL do componente que deve tratar os dados do formulário. Mapeamos nosso Servlet usando o padrão “/processaDadosCliente” não foi? Então, qual será a URL do nosso Servlet? Resposta: “http://localhost:8084/PrimeiroFormulario/processaDadosCliente”. Vamos editar nossa *tag* *form* então, configurando a propriedade *action* para a URL citada. Na Listagem 2.2 você pode ver como ficou o código da *tag* *form*. Note que não estou listando todo o arquivo.

Listagem .2: Configurando a propriedade *action* da *tag* *form*



Fonte: do autor

Isso que dizer que, quando clicarmos no botão “Enviar Dados” (que é um *input* do tipo *submit*), os dados que estiverem preenchidos nos componentes que estão dentro deste formulário, serão enviados para http://localhost:8084/PrimeiroFormulario/processaDadosCliente, que é o endereço do nosso Servlet! Já alterou o arquivo? Salvou? Legal! Atualize a página, preencha os campos e clique no botão “Enviar Dados”. O que aconteceu? Uma página em branco foi exibida não é? E na barra de endereços, o que apareceu? O endereço do Servlet mais um monte de “coisas”! Os detalhes sobre isso nós iremos aprender na próxima seção, então não se preocupe por enquanto.

Você se lembra dos nossos primeiros exemplos? Acessávamos o endereço do Servlet, uma página em branco era exibida e duas Strings eram direcionadas para a saída do Tomcat, lembra? Quem fazia esse direcionamento era o Servlet não era? Vamos fazer a mesma coisa com o nosso Servlet de dados dos clientes, mas mostraremos os dados que foram preenchidos nos formulários. Vamos lá então?

Volte ao Servlet, vamos implementar o método processRequest. Qual é mesmo o nome do parâmetro do método processRequest que armazena os dados enviados pelo cliente? É o *request* certo? Veja o código da Listagem 2.3. Leia o comentário que fiz.

Listagem .3: Implementação do método processRequest



Fonte: do autor

Copiou o código? Legal. Vamos entender o que está acontecendo. Na linha 14 é declarada uma variável do tipo String com o nome de “nome”. Essa variável é inicializada com o valor obtido ao se chamar o método getParameter( String param ) de request. O parâmetro passado ao método getParameter, que é uma String, é o nome que foi dado ao componente do formulário, que no caso foi “nome”. Estude as linhas 15, 16, 17, 18 e 19 e tente fazer um paralelo com o formulário contido no index.jsp. Note que a String que é passada como parâmetro no método getParameter(...) sempre reflete o nome dado ao componente do formulário através da propriedade *name*. Veja a linha 17. Declaramos uma variável chamada dataNascimento que é inicializada com o valor do parâmetro “dataNasc” (configurado no formulário).

O que fizemos até a linha 19 foi criar uma variável que vai receber o valor de cada componente do formulário. A partir da linha 21, até o final do método, direcionamos para a saída os dados que foram obtidos. Vamos testar? Salve o Servlet e execute o projeto (botão de “play”, lembra?). Na página, preencha o formulário, clique em “Enviar Dados” e volte no NetBeans para ver o que aconteceu. Olhe na janela de saída! Lá estão os dados que você preencheu no formulário! Muito bem! Imagine agora se esse fosse um sistema real. Esses dados recebidos dentro do Servlet poderiam alimentar uma *query* SQL para inserir esse cliente em um banco de dados! Legal não é?! A partir desse ponto, você já deve estar entendendo melhor o que está acontecendo não é mesmo? Mas e aquele monte de coisas escritas no endereço do navegador depois de clicar em Enviar Dados? Esse comportamento está intrinsecamente ligado ao tipo de método HTTP que estamos usando. Vamos para a próxima seção, vou explicar isso lá.

## Métodos HTTP

O protocolo HTTP 1.1, que usamos em nossas aplicações Web, define uma série de métodos que podem ser usados para tratar diversos tipos de requisições. Na nossa vida, como desenvolvedores Web, iremos nos importar somente com os métodos GET e POST. Sendo assim vamos detalhá-los.

<Saiba mais:

Quer conhecer os outros métodos HTTP (HEAD, TRACE, etc.)? Acesse este link: http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html (em inglês)

>

### Método GET

O método GET (*to get* = obter) é usado principalmente para pedir ao servidor algum recurso. Quando nós fazemos uma pesquisa no Google, por exemplo, estamos usando o método GET. Qualquer URL que colocamos na barra de endereço do nosso navegador é enviada ao servidor usando o método GET. Vamos fazer um teste. Entre na página do Google (www.google.com.br) e pesquise por “métodos http” (sem as aspas). Ao clicar em pesquisar, o resultado será mostrado no navegador. Veja a barra de endereços. Haverá algo assim:

http://www.google.com.br/#hl=pt-BR&biw=1440&bih=713&q=métodos+http&aq=f&  
aqi=g10&aql=&oq=&gs\_rfai=&fp=90f65ad7da748e6d

Parece grego, mas não é! Vamos entender a URL. Estamos usando o protocolo HTTP, para acessar a máquina www.google.com.br. O recurso é representado pelo restante da URL:

hl=pt-BR&biw=1440&bih=713&q=métodos+http&aq=f&aqi=g10&aql=&oq=&gs\_rfai=&  
fp=90f65ad7da748e6d

Vamos dividir esse resto da URL nos símbolos “&”. Vamos obter isso aqui:

hl=pt-BR  
biw=1440  
bih=713  
q=métodos+http  
aq=f  
aqi=g10  
...

Note que a forma de cada pedaço da parte correspondente ao recurso é x=y, onde “x” é o nome de uma variável e “y” é o valor. No nosso exemplo, a variável “q” tem o valor “métodos+http” que no caso é a nossa consulta! Ou seja, o componente que trata as pesquisas do Google, entende que a variável “q” vai conter o valor da pesquisa que estamos fazendo!

Execute novamente o nosso projeto, limpe todos os campos e preencha o campo “Nome” com “Juca” (sem as aspas) e o campo “Sexo” com Masculino e clique em “Enviar Dados”. Veja a URL que foi obtida na barra de endereços:

http://localhost:8084/PrimeiroFormulario/processaDadosCliente?nome=Juca&  
sobrenome=&cpf=&dataNasc=&sexo=M&observacoes=

Veja o caminho do recurso!

processaDadosCliente?nome=Juca&sobrenome=&cpf=&dataNasc=&sexo=M&observacoes=

O que isso quer dizer:

Senhor “processaDadosCliente”, por favor, receba as variáveis nome, sobrenome, cpf, dataNasc, sexo e observacoes, utilize seus valores e me devolva o recurso associado a eles. O ponto de interrogação após o mapeamento do Servlet (/processaDadosCliente) indica que o que vem depois dele (do ponto de interrogação) são variáveis HTTP. Cada variável, como já vimos, está na forma x=y, onde “x” é a variável e “y” é o valor, sendo que elas são separadas por “&”. Então temos: “nome” igual a “Juca”, “sobrenome” igual a vazio, “cpf” igual a vazio, “dataNasc” igual a vazio, “sexo” igual a “M” e “observacoes” igual a vazio. A saída no NetBeans deve ter ficado assim:

Dados do Cliente:  
Nome: Juca  
Sobrenome:   
CPF:   
Data de Nascimento:   
Sexo: Masculino  
Observações:

Vamos mandar a requisição novamente para o NetBeans, só que agora modificando a URL ao invés de usar o formulário. Dê o sobrenome de “Santos” ao Juca e defina o CPF como 123456789. Preencheu a URL na barra de endereços? Tecle <ENTER> e veja o que aconteceu no NetBeans. A saída deve ter sido essa aqui:

Dados do Cliente:  
Nome: Juca  
Sobrenome: Santos  
CPF: 123456789  
Data de Nascimento:   
Sexo: Masculino  
Observações:

Então, basicamente, ao usarmos o método GET, indicamos que queremos algum recurso do servidor. Quando enviamos dados através do método GET, esses dados, na forma de variáveis, são codificados na própria URL. O nosso formulário do index.jsp utiliza por padrão o método GET. Qualquer formulário usa por padrão o método GET. Se quisermos mudar o método de envio do formulário, precisamos usar a propriedade *method* da *tag* *form*. Ai você me pergunta: Porque usaríamos outro método? O GET já não funciona? E eu respondo: Sim, o GET funciona, mas imagine a seguinte situação: você vai armazenar os dados de um usuário de um sistema. Você vai mandar vários dados para o servidor, inclusive uma senha. O que acontece? A senha enviada vai aparecer na URL! Afinal, a senha é um campo do formulário! O ideal seria ninguém a ver correto? Outro problema. O tamanho de uma URL é fixo! Então não podemos mandar conteúdos de tamanho arbitrário, visto que iremos perder dados caso usemos o método GET! Então como fazemos? Método POST, ao resgate!

<Saiba mais:

O tamanho fixo das URLs é uma restrição imposta pelos navegadores e/ou pelos servidores, visto que o protocolo HTTP não fixa o tamanho de uma URL. Mais detalhes aqui: http://www.boutell.com/newfaq/misc/urllength.html (em inglês)

>

### Método POST

O método POST (*to post* = postar) é usado para enviar dados ao servidor. Ao contrário do método GET, ao usar o método POST, as variáveis de um formulário não são inseridas na URL, mas sim no corpo da requisição. Sendo assim, a quantidade dos dados enviados usando o método POST pode ter qualquer tamanho, desde apenas uma variável, até arquivos de tamanhos variados. Você já enviou uma foto para o Orkut não enviou? Saiba que ela foi enviada usando o método POST.

Como eu já disse na seção anterior, por padrão, o navegador envia os dados de um formulário usando o método GET. Caso queiramos mudar esse comportamento, basta usar a propriedade *method* da *tag* *form*. Vamos fazer isso? Vá ao NetBeans, abra o arquivo index.jsp caso não esteja aberto, procure pela *tag* *form* e insira a propriedade *method*. Veja na Listagem 2.4 como deve ficar:

Listagem .4: Usando o método POST para enviar a requisição para o Servlet



Fonte: do autor

Salve o arquivo e execute o projeto novamente. Preencha o formulário e clique em “Enviar Dados”. Verifique a URL, pois agora as variáveis não serão mais codificadas nela. Verifique a saída no NetBeans para constatar que os dados continuam a ser enviados. Edite novamente o index.jsp e mude o método para GET. Teste novamente. As variáveis devem estar aparecendo novamente na URL não é? De novo, edite o index.jsp e volte para o método POST e teste de novo.

Muito bem! Estamos quase acabando. Tenho certeza que você deve estar entendendo tudo. Se não estiver, releia o que está com dúvida tudo bem? Vamos à nossa última seção, que vai tratar de mais um pouquinho de teoria.

### Tratando Métodos HTTP

Você deve lembrar que quando criamos um Servlet manualmente, nós criávamos três métodos, o processRequest(...), o doGet(...) e o doPost(...). Quando criamos um Servlet usando o assistente do NetBeans, ele também cria uma estrutura parecida com a que a gente criava manualmente, além de já realizar o mapeamento do nosso Servlet. Na seção anterior falamos dos métodos GET e POST do protocolo HTTP, que são os que nós usamos como desenvolvedores Web. Você já deve ter notado, e eu também já falei, que um Servlet deve implementar o método HTTP que ele deve tratar. A implementação de um método HTTP em um Servlet deve ser feita dentro de métodos que por padrão são nomeados “doXXX(...)”, onde XXX deve ser trocado pelo nome do método HTTP em questão. Sendo assim, requisições usando o método GET são tratadas dentro do método doGet(...) do Servlet. Requisições usando o método POST são tratadas dentro do método doPost(...) do Servlet e assim por diante.

Note então que todos os Servlets que criamos até agora se comportam da mesma forma tanto para o método GET quanto para o método POST, pois sempre que a requisição chega no Servlet, o método apropriado é escolhido, entretanto, tanto o método doGet(...), quanto o método doPost(...), direcionam o fluxo de execução para o método processRequest(...)! Nós sempre iremos fazer assim, pois é muito mais prático e não precisamos ficar nos preocupando qual método um Servlet trata, precisamos apenas nos preocupar em modificar a propriedade *method* da *tag* *form* quando quisermos mudar o método que deve ser utilizado.

Muito legal não é mesmo? Com isso fechamos nossa segunda semana. Na próxima aula, iremos aprender a trabalhar com dois recursos superimportantes da especificação das JSPs: EL (*Expression Language*) e TagLibs. Após aprender essas duas funcionalidades, estaremos prontos para começar a criar nosso primeiro projeto que trabalha com banco de dados, mas antes disso ainda iremos formalizar e aprender algumas coisinhas. Ah, não se esqueça de praticar o que aprendemos até agora! Vamos ao resumo da semana.

Resumo

Nesta aula demos um passo muito importante para a nossa vida como desenvolvedores Web, pois aprendemos a trabalhar com formulários e entendemos o funcionamento dos métodos GET e POST que fazem parte do protocolo HTTP. Como você já deve ter percebido, os formulários desempenham um papel importantíssimo nas aplicações Web. Tenho certeza que de agora em diante, sempre que você usar uma aplicação Web, você saberá como aquele formulário funciona. Para colocar em prática o que aprendemos, criamos um projeto Java Web no NetBeans e fizemos diversos testes.

Atividades de Aprendizagem

1 – Qual a diferença entre os métodos GET e POST? Quando devemos utilizar um ou o outro?

2 – Incremente o projeto que criamos durante a aula inserindo mais alguns campos no formulário: email, logradouro, número, complemento, cidade, estado, CEP, se o cliente tem ou não filhos. Utilize apropriadamente os tipos de *input* que aprendemos até agora.

3 – Crie um novo projeto Java Web, com o nome de “FormularioDVD” (sem as aspas), que deve ter um formulário usado para enviar dados de um DVD. Um DVD, no nosso caso, deve ter: número, título, ator/atriz principal, ator/atriz coadjuvante, diretor/diretora e ano de lançamento. Para tratar o formulário, crie um Servlet usando o assistente do NetBeans. Esse Servlet deve obter os dados enviados através do formulário e imprimi-los na saída padrão (usando System.out.println(...)), como foi feito no exemplo construído durante esta aula. Esse formulário deve usar o método POST.

4 – Crie um novo projeto Java Web, com o nome de “FormularioProduto” (sem as aspas), que deve ter um formulário usado para enviar dados de um Produto. Um Produto, no nosso caso, deve ter: código de barras, descrição, unidade de medida (unidade ou kg), quantidade por embalagem, fabricante (nome). Para tratar o formulário, crie um Servlet usando o assistente do NetBeans. Esse Servlet deve obter os dados enviados através do formulário e imprimi-los na saída padrão (usando System.out.println(...)), como foi feito no exemplo construído durante esta aula. Esse formulário deve usar o método POST.

5 – Crie um novo projeto Java Web, com o nome de “CalculadoraWeb” (sem as aspas), que deve ter um formulário usado para atuar como uma calculadora. Nesse formulário, deve haver dois campos (“número 1” e “número 2”) e um conjunto de radios para representar a operação a ser realizada (adição, subtração, multiplicação e divisão). Para tratar o formulário, crie um Servlet usando o assistente do NetBeans. Esse Servlet deve obter os dados enviados através do formulário, executar a operação escolhida pelo usuário e imprimir o resultado na saída padrão (usando System.out.println(...)), como foi feito no exemplo construído durante esta aula. Esse formulário deve usar o método GET. Faça testes de envio dos dados usando apenas a URL gerada depois da primeira submissão.

6 – Crie um novo projeto Java Web, com o nome de “TamanhoString” (sem as aspas), que deve ter um formulário com apenas um campo usado para enviar uma String de qualquer tamanho para um Servlet. Utilize uma *textarea* para o usuário poder inserir essa String no formulário. O Servlet deve obter a String enviada e imprimir o a quantidade de caracteres da String na saída padrão (usando System.out.println(...)), como foi feito no exemplo construído durante esta aula. Qual método HTTP deve ser utilizado nessa situação? Justifique sua resposta.

7 – Desafio – Crie um novo projeto Java Web, com o nome de “EhPrimo” (sem as aspas), que deve ter um formulário usado para enviar um número inteiro para um Servlet, que por sua vez deve verificar se este número é primo. O resultado do teste deve ser impresso na saída padrão. Esse formulário deve usar o método GET.

8 – Desafio – Crie um novo projeto Java Web, com o nome de “EquacaoSegundoGrau” (sem as aspas), que deve ter um formulário usado para enviar os coeficientes de uma equação de segundo grau para um Servlet, que por sua vez deve calcular as raízes da equação em questão e imprimir essas raízes na saída padrão. As raízes de uma equação do segundo grau podem ser determinadas usando a fórmula de Bhaskara (http://pessoal.sercomtel.com.br/matematica/fundam/eq2g/eq2g.htm, http://pt.wikipedia.org/wiki/Bhaskara\_II). O Servlet deve verificar também se os coeficientes passados representam uma equação do segundo grau válida. Esse formulário deve usar o método GET.

# Aula 3 – *Expression Language* e TagLibs

**Objetivos**

Entender a sintaxe e o propósito da *Expression Language*.

Aprender a utilizar a *Expression Language*.

Entender o propósito das *tags* JSP.

Aprender a utilizar as *tags* JSP.

Aprender a utilizar as *tags* disponibilizadas na JSTL (*Java Standard Template Library*).

## Introdução

Nesta aula iremos aprender duas funcionalidades muito importantes e úteis do mundo Java Web: EL (*Expression Language*) e as TagLibs. Essas duas funcionalidades nos ajudarão na tarefa de não misturar código Java nas nossas JSPs. Você se lembra quando falei que era possível inserir código Java dentro das JSPs, não lembra? Falei também que isso não deveria ser feito, pois deixa o código difícil de ler e de manter, além de fazer com que o trabalho do Web designer (que normalmente conhece mais HTML) se torne difícil, visto que ele teria que ter um bom conhecimento em Java e em como as JSPs funcionam. Usando a EL e as TagLibs, a manipulação de dados, provenientes dos Servlets, se torna muito mais fácil, pois utiliza uma sintaxe simples e fácil de entender, ajudando no trabalho de quem não conhece muito bem o funcionamento de aplicações Web em Java. Vamos começar?

## *Expression Language* (EL)

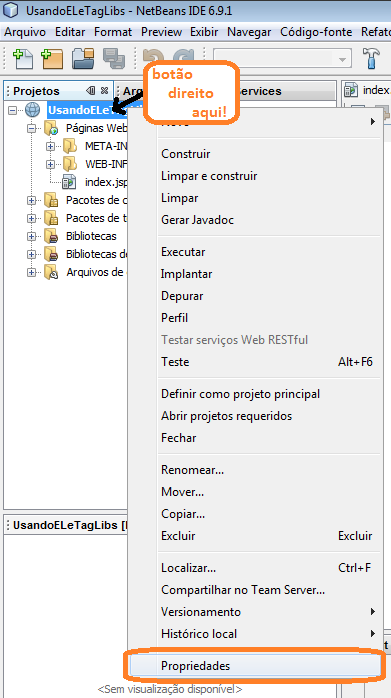
A EL é um recurso da especificação das JSPs que permite utilizarmos uma sintaxe especial para obter dados que gostaríamos de mostrar nas nossas páginas, além de permitir que façamos algumas outras coisas, como por exemplo, avaliar uma expressão lógica. Como de costume, iremos utilizar um projeto para aprendermos o recurso que estamos estudando. Vamos criar então o projeto, mas fique atento, pois agora iremos configurar algumas coisas no projeto que ainda não estávamos fazendo.

Clique no menu “Arquivo” e escolha a opção “Novo Projeto...”. A janela do assistente para criação de projetos será aberta. Em “Categorias” escolha “Java Web”, em “Projetos” escolha “Aplicação Web” e clique em “Próximo”. Até agora, como você deve ter percebido, o processo que estamos realizando é o mesmo que fizemos nas semanas anteriores. A partir do segundo passo iremos fazer algumas modificações. Não se esqueça de fazê-las quando você estiver criando seus próximos projetos!

Preencha o campo “Nome do projeto” com “UsandoELeTagLibs” (sem as aspas). Marque a opção “Usar pasta dedicada para armazenar bibliotecas”. O NetBeans vai sugerir “.\lib” como “Pasta de bibliotecas”. Deixe assim. No decorrer da aula, iremos inserir bibliotecas externas aos nossos projetos e fazendo esse passo que descrevi, nossas bibliotecas serão copiadas e referenciadas dentro do nosso projeto, permitindo que possamos abrir nosso projeto em qualquer instalação do NetBeans, mesmo que a versão usada não tenha as bibliotecas que utilizamos no nosso projeto. Clique em “Próximo” e siga os próximos passos da mesma forma que você tem feito.

Assim que o projeto for criado, iremos fazer uma modificação nas suas propriedades. Fique atento, pois como falei, a partir de agora, todos os nossos projetos serão modificados dessa forma. Clique com o botão direito no nó raiz do projeto e selecione a opção “Propriedades”. Veja a Figura 3.1.

Figura .1: Acessando as propriedades do projeto

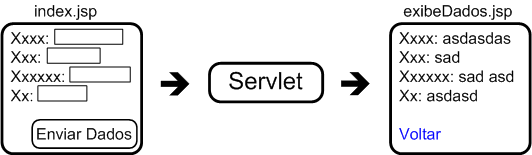


Fonte: *Print screen*, NetBeans IDE 6.9.1

Ao fazer isso, o diálogo de propriedades do projeto será exibido. Procure em “Categorias” por “Executar” e clique nesse nó. Do lado direito do diálogo, serão exibidas as propriedades de execução do projeto, ou seja, como ele deve ser executado. Note que existe uma opção chamada “Implantar ao salvar”. Desmarque essa opção e clique em “OK”. As propriedades do projeto serão salvas e o diálogo será fechado. Ao desmarcar a opção “Implantar ao salvar”, estamos dizendo ao NetBeans que ele não deve implantar a aplicação no servidor (no caso o nosso container) toda vez que salvarmos algum arquivo. Isso faz com que o Tomcat não fique sobrecarregado durante o processo de desenvolvimento.

O fluxo do projeto que vamos desenvolver é representado na Figura 3.2.

Figura .2: Fluxo de execução do projeto



Fonte: do autor

Analisando a Figura 3.2, podemos perceber que teremos um formulário no index.jsp que terá seus dados tratados por um Servlet, que por sua vez irá fazer algum processamento e direcionar o resultado gerado para uma segunda JSP, chamada exibeDados.jsp.

Edite o index.jsp e insira um formulário. O meu ficou como mostrado naListagem 3.1. Copie o código para o seu index.jsp e teste.

Note que nesse formulário temos algumas coisas novas. A primeira novidade é a *tag* *style* (linha 11). Usamos essa *tag* para criar estilos de formatação para o nosso documento. Tudo que usarmos de formatação como: alinhamento, cor de texto, etc., será definido usando estilos. Esses estilos são codificados usando as folhas de estilo (CSS – *Cascading Style Sheets*). A sintaxe é muito simples. As definições em CSS são chamadas de seletores. Sendo assim, a definição “.alinharDireita” (linha 12) indica que queremos criar um seletor que é uma classe de formatação (denotada pelo ponto (.)) que tem como nome “alinharDireita”. Todas as propriedades de formatação de um seletor são inseridas entre chaves. Note que usei a propriedade “*text-align*” com o valor de “*right*”. Ou seja, todas as *tags* que usarem a classe “.alinharDireita” vão ser formatadas para alinhar seu texto a direita.

Você já deve conhecer as tabelas do HTML não é mesmo? Note que organizei todo o formulário dentro de uma tabela e que a primeira coluna da tabela usa a classe “.alinharDireita”, que definimos lá no começo do arquivo. Verifique a linha 27 para ver um exemplo. Outra modificação que fiz foi em relação à *action* da *tag* *form*. Note que agora não estou mais colocando o caminho completo do Servlet (que vai ser mapeado como “/processaDadosProduto”), pois o Servlet é acessado no mesmo nível da nossa JSP (index.jsp). Esse tipo de caminho se chama “caminho relativo”, pois o mapeamento do Servlet (http://localhost:8084/UsandoELeTagLibs/processaDadosProduto) está no mesmo diretório em relação ao index.jsp (http://localhost:8084/UsandoELeTagLibs/index.jsp), sendo assim, não precisamos colocar o caminho completo, pois os dois recursos estão no mesmo diretório. Ao prosseguirmos com o conteúdo, irei ensinar uma técnica muito útil para não termos problema com os caminhos dos recursos.

A última novidade no nosso formulário é o uso da *tag* *select* que é usada para criar uma caixa de seleção (*combo box*). Veja que a propriedade “*name*” é definida na *tag* *select* e que dentro dessa *tag* existem três *tags* do tipo “*option*” (opção). A *tag* *option* é usada para criar um item da caixa de seleção. Cada *option* tem um valor associado que será enviado para o servidor com base na seleção feita. Por exemplo, se a opção “Unidade” for selecionada, será enviado o valor “un” na variável “unidade” (definida na propriedade “*name*” da *tag* *select*).

Listagem .1: Formulário para envio de dados de um produto



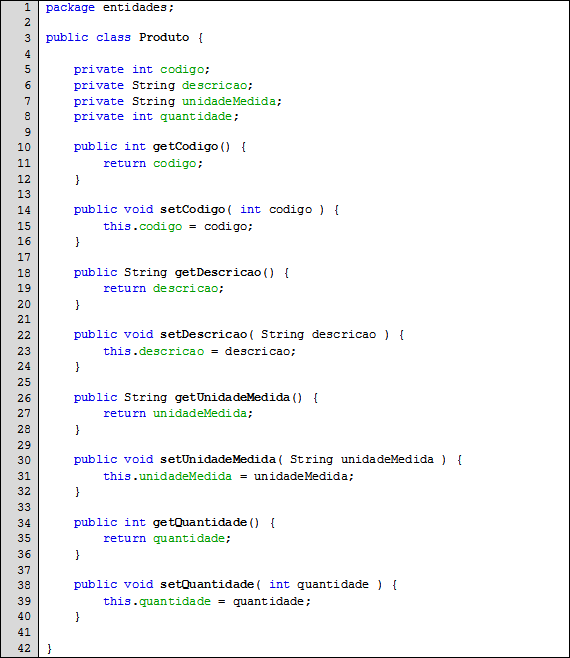
Fonte: do autor

Com o formulário pronto, vamos criar uma classe que vai representar o nosso produto. Na pasta “Pacotes de código-fonte”, crie um novo pacote chamado “entidades” (sem as aspas). Nesse pacote, crie uma classe com o nome de “Produto” (sem as aspas). Nosso produto contém um código, uma descrição, uma unidade de medida e uma quantidade em estoque. Sendo assim, nossa classe também terá esses quatro campos, que devem ser implementados como membros privados. Você deve ter aprendido que quando criamos uma classe, devemos tornar seus campos privados e então criar métodos públicos para configurar e obter esses dados. Em Java, nós usamos um padrão chamado *JavaBeans*, que define algumas regras para se nomear os métodos que serão usados. Por exemplo, nosso produto terá um membro privado chamado “quantidade”, então teremos dois métodos públicos para acessar esse campo. O método setQuantidade (configureQuantidade) será usado para configurar a quantidade, enquanto o método getQuantidade (obtenhaQuantidade) será usado para obter o valor da quantidade. Uilizando essa abordagem de usar métodos para obter e configurar certos campos, nós obtemos o que chamamos de propriedades de uma determinada classe, pois independente de como os métodos são implementados, os usuários da classe só enxergam os métodos públicos. Lembra-se da propriedade do encapsulamento da programação orientada a objetos? Olha ela ai! Note que usamos os prefixos “*set*” e “*get*” para nomear métodos que respectivamente alteram e obtenham uma determinada propriedade do objeto. O uso desses prefixos, entre outros detalhes, é descrito no padrão *JavaBeans*.

Quantos detalhes hein? Veja na Listagem 3.2 como ficou a implementação da classe Produto.

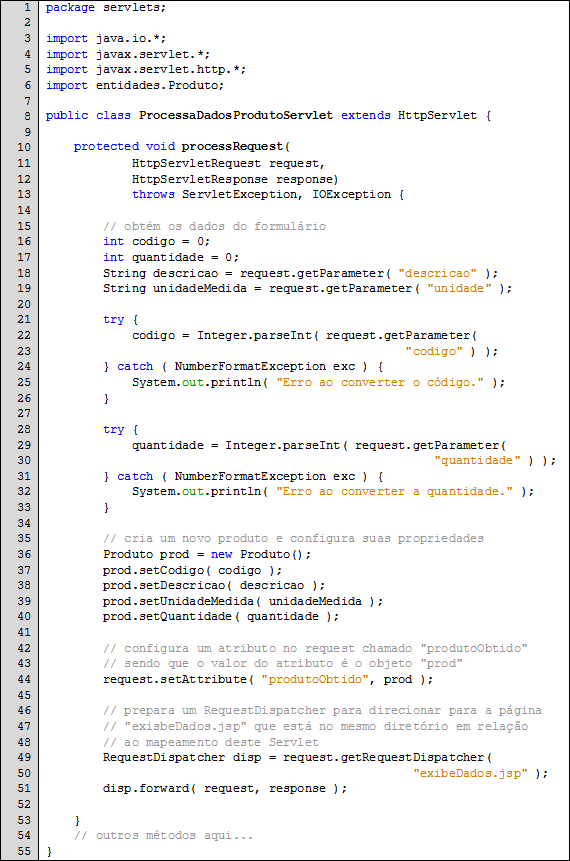
Com a classe “Produto” implementada, agora podemos criar objetos do tipo “Produto” que vão conter os dados obtidos no formulário. Quem obtém e processa os dados enviados pelo formulário são os Servlets, então vamos criar um. Crie o pacote “servlets” para conter o Servlet que será criado. A classe do nosso Servlet, que deverá estar dentro do pacote recém-criado, terá o nome de “ProcessaDadosProdutoServlet” e deverá ser mapeada para “/processaDadosProduto” em “Padrão(ões) de URL”. A implementação do método processRequest do Servlet criado pode ser vista na Listagem 3.3.

Listagem .2: Implementação da classe Produto



Fonte: do autor

Listagem .3: Implementação do Servlet que cria um novo Produto



Fonte: do autor

Vamos às novidades apresentadas nesse Servlet. Entre as linhas 16 e 19 declaramos as variáveis que vão conter o valor dos campos do formulário, sendo que só obtemos os valores da descrição e da unidade de medida, pois o método getParameter retorna Strings.

Para as variáveis inteiras, nós precisamos converter o valor retornado de “código” e “quantidade” para inteiro. Você já deve conhecer esse tipo de conversão não é mesmo? Entre as linhas 21 e 33 usamos um *try* para verificar se a conversão de cada valor que deve ser inteiro foi bem sucedida. Caso você não conheça essa construção da linguagem, segue então uma explicação bem rápida.

O bloco *try* (tentar) é usado na linguagem Java para englobar trechos de código que, ao serem executados, podem emitir certos tipos de “erros”. Esses “erros” são chamados de exceções. O método parseInt da classe Integer pode gerar um tipo desses erros quando é passado para ele uma String que não representa um número. Exemplo: imagine que no formulário dos dados do produto, você preencheu “um” ao invés de “1” no campo código. Esse valor (“um”) vai para o Servlet e quando o método parseInt tenta convertê-lo para um inteiro, ele verifica que “um” não representa um número, então ele dá um tipo de “grito”, que avisa quem está usando o método que alguma coisa errada aconteceu. Para ouvir esse grito, precisamos usar o bloco *try* e, logo em seguida, usar um *catch*, que é como se fosse um tipo de ouvido que só ouve um tipo de grito. O parseInt tentou converter “um”, não conseguiu e então gritou: “NumberFormatException!!!” Como temos um *catch* (ouvido) configurado para ouvir esse tipo de grito, quando o parseInt gritar, o *catch* vai entender o grito e vai fazer alguma coisa, que no caso é mostrar na saída “Erro ao converter o código.”. A mesma coisa é feita para o valor da quantidade. Resumindo – o *try* é usado para englobar uma ou mais linhas de código que potencialmente podem lançar algum tipo de exceção, sendo que a exceção que é lançada dentro do *try* deve ser capturada em um *catch* correspondente. Essa explicação é uma forma bem simples de entender o mecanismo de tratamento de exceções do Java, visto que existem muitos outros detalhes.

<Saiba mais:

Quer saber mais sobre tratamento de exceções? Veja os links a seguir:

http://imasters.com.br/artigo/11674/java/tratamento\_de\_excecoes\_na\_pratica/

http://www.inf.puc-rio.br/~java/progjava/protected/apostilas/transparencias/Excecoes.pdf

http://pt.wikipedia.org/wiki/Tratamento\_de\_exceções

>

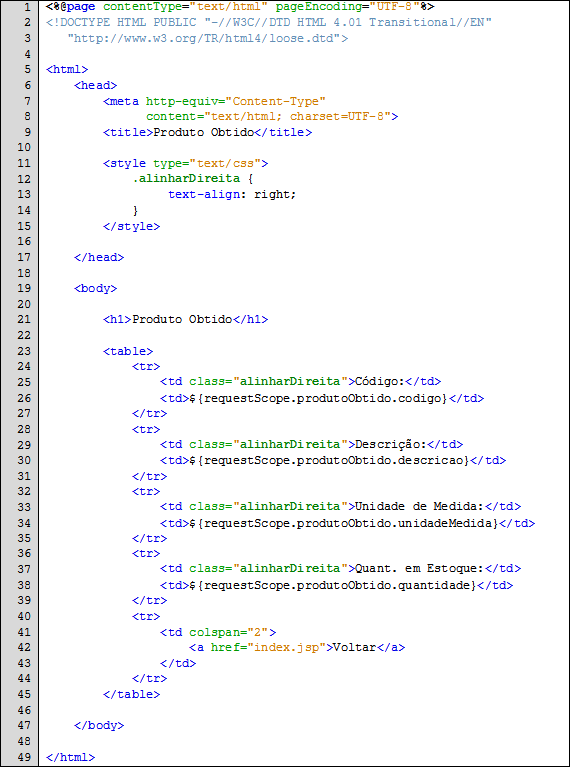
Voltando ao Servlet... Na linha 36 da Listagem 3.3 é instanciado um novo Produto e este é atribuído a uma referência do tipo Produto chamada “prod”. Entre as linhas 37 e 40 são configuradas as propriedades do produto a partir dos dados obtidos através do formulário. Na linha 44, inserimos um atributo no request. Damos o nome de “produtoObtido” a esse tributo e configuramos seu valor como sendo o produto que criamos e configuramos entre as linhas 36 e 40. Isso quer dizer que a próxima página ou Servlet que receber a requisição a partir deste Servlet, vai receber um objeto *request* com esse atributo, ou seja, o objeto “prod” (que é um produto) vai ficar acessível a outro componente da nossa aplicação! Confuso? Já você vai entender, fique calmo.

Na linha 49 criamos um RequestDispatcher, que é usado para direcionar o fluxo de execução do Servlet que está sendo executado para um outro recurso. No caso, esse recurso foi definido como “exibeDados.jsp”, uma página JSP que ainda vamos criar e que vai usar o atributo “produtoObtido” configurado no *request* para exibir os dados do produto.

Por fim, na linha 51, o método *forward* de “disp”, que é o nosso RequestDispatcher para o recurso “exibeDados.jsp”, é invocado, passando como parâmetro o *request* e o *response* do Servlet. Quando o método *forward* é invocado, o container direciona o fluxo para o recurso configurado e devolve o controle para o navegador caso o recurso deva ser exibido por ele. Uma JSP, por padrão, é usada para isso não é mesmo?

O que a página “exibeDados.jsp” vai fazer é pegar o atributo “produtoObtido” configurado no *request* e mostrar seus dados. Vamos criar então essa página. No NetBeans, procure pela pasta chamada “Páginas Web”. Clique com o botão direito nela, escolha “Novo” e procure por “JSP...”. Se não achar a opção JSP, você já deve saber como proceder não é mesmo? Preencha o campo “Nome do arquivo” com “exibeDados” (sem as aspas) e clique em “Finalizar”. O arquivo será criado e exibido no editor. Veja na Listagem 3.4 como ficou o código depois de ser editado.

Listagem .4: Código da página exibeDados.jsp

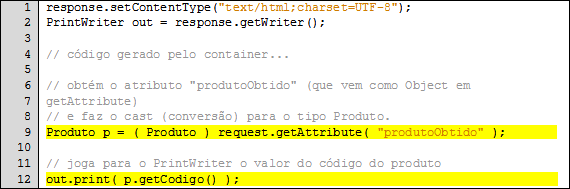


Fonte: do autor

Copiou? Salvou? Faça um teste então! Execute o projeto, preencha o formulário e clique em “Enviar Dados”. O Servlet será invocado, processará os dados e vai redirecionar para a página “exibeDados.jsp”, que por sua vez vai mostrar os dados do produto. Legal não é? Mágica? Não! Vamos entender o que está acontecendo na página “exibeDados.jsp”. Veja as linhas 26, 30, 34 e 38. A construção ${...} é a EL! Usando a EL, nós podemos acessar valores que estão configurados no *request* e em outros “lugares” (escopos) também (que vamos aprender depois). No caso, o objeto *requestScope* da EL faz referência ao objeto request do Servlet que é gerado a partir da JSP. Lembre-se que uma JSP é convertida em um Servlet automaticamente pelo container!

Usar a expressão “${requestScope.produtoObtido.codigo}” em EL quer dizer: obtenha o código, do objeto configurado no atributo “produtoObtido” do request. Lembre-se, nós configuramos no atributo “produtoObtido” no *request* (linha 44 da Listagem 3.3) um produto, que por sua vez tem um código (acessado pelo método getCodigo). O código Java equivalente a “${requestScope.produtoObtido.codigo}” em um Servlet é mostrado na Listagem 3.5 nas duas linhas destacadas.

Listagem .5: Trecho de código correspondente à EL



Fonte: do autor

O propósito da EL é obter objetos que estão ativos nos diversos escopos da aplicação e poder obter suas propriedades, sem precisar lidar diretamente com código Java. Sei que esse foi um exemplo bem simples, mas tenho certeza que você deve ter entendido. Veja que o “codigo” usado na EL é relativo ao método getCodigo() e não ao membro privado da classe Produto chamado “codigo”. Essa mágica se dá pelo uso do padrão *JavaBeans*! Como exercício mental, analise as linhas 30, 34 e 38 e tente imaginar o que está acontecendo. Agora que já sabemos o que é a EL e como utilizá-la, vamos às *tags* JSP.

## *Tags* JSP

Na especificação das JSPs, existem uma série de *tags* especiais que devem ser implementadas por quem implementa a especificação. Essas *tags* são nomeadas usando o prefixo “jsp”. Na verdade, nós quase não iremos utilizar essas *tags* e as que utilizarmos, explicarei no momento oportuno, mas para você ter uma ideia de como elas funcionam, crie uma página JSP chamada “testesTags” na pasta “Páginas Web” do projeto que estamos trabalhando. Veja na Listagem 3.6 o código que você deve copiar para o arquivo “testesTags.jsp”.

Listagem .6: Exemplo das *tags* *jsp:useBean* *jsp:setProperty*



Fonte: do autor

Copie o código no seu arquivo, salve e entre no endereço http://localhost:8084/UsandoELeTagLibs/testesTags.jsp no seu navegador para testar a página. O que aconteceu? O produto criado foi exibido assim “4, Arroz, kg, 100” não foi? Vamos analisar o código no arquivo “testesTags.jsp”. Entre as linhas 14 e 17 definimos um comentário, que nas JSPs é delimitado entre “<%--“ e “--%>”. Esse comentário não pode ser visto no código-fonte da página HTML gerada! Nas linhas 18 e 19 usamos a *tag* *jsp:useBean* para criar um objeto com nome de “meuProduto” (configurado pela propriedade *id)*, do tipo “entidades.Produto” (configurado pela propriedade *class*), que vai existir no escopo da página, ou seja, esse objeto só existe nesta página. Note que precisamos colocar o caminho completo da classe na propriedade *class* para informarmos qual o tipo de objetos que queremos instanciar. A partir da linha 22, usamos a *tag* *jsp:setProperty* para configurar as propriedades do objeto chamado “meuProduto” que foi criado usando a *tag* *jsp:useBean*. No exemplo da linha 22 e 23, referenciamos o objeto “meuProduto” e configuramos a propriedade “codigo” ((*property*=“codigo”) com o valor “4”. A instrução em Java equivalente a estas duas linhas é “meuProduto.setCodigo(4)”. A partir da linha 33, mostramos então os dados do objeto “meuProduto” que foi criado usando EL. Note que desta vez, usamos “pageScope” ao invés de “requestScope”, visto que o objeto existe apenas no escopo da página (veja na linha 19).

Da mesma forma que existem as *tags* JSP padrão, você pode criar suas próprias *tags* que podem ter comportamentos dos mais variados possíveis, entretanto nós não iremos aprender a fazer isso. Como é possível criar *tags* personalizadas, nós podemos usar conjuntos de *tags* que são implementadas por terceiros em nossos projetos. Um desses conjuntos é a JSTL (*JavaServer Pages Standard Tag Library*), que vamos aprender na próxima seção. Vamos lá então!

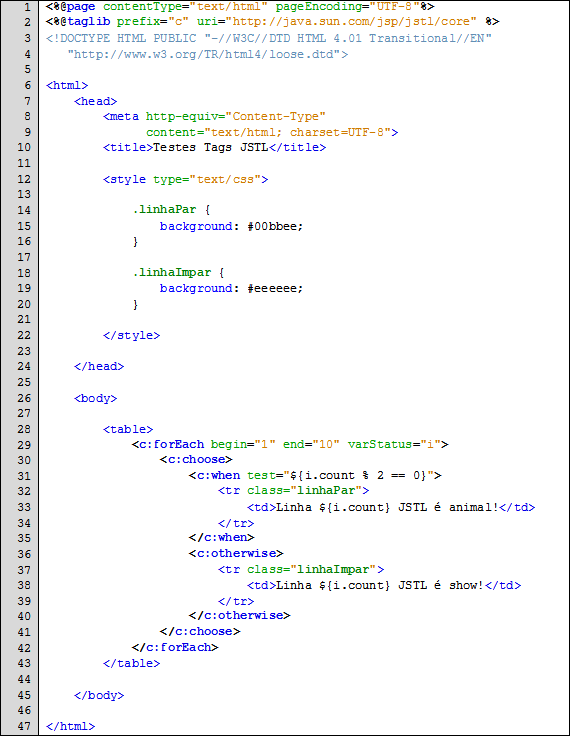
## JSTL – *JavaServer Pages Standard Tag Library*

A JSTL é uma biblioteca formada por um conjunto de *tags* (TagLib = *Tag Library* = Biblioteca de *Tags*) que visam apoiar o desenvolvedor na tarefa de construir suas páginas JSP, permitindo que várias coisas possam ser feitas sem o uso direto de código Java, por exemplo, iterar por uma lista de objetos, executar testes lógicos, formatar dados, entre muitos outros. Quando formos implementar nosso primeiro projeto na Aula 5, iremos utilizar muitos recursos da JSTL, mas por agora vamos aprender apenas como inseri-la no nosso projeto e fazer um pequeno exemplo.

Procure no seu projeto uma pasta chamada “Bibliotecas”, clique nela com o botão direito e escolha “Adicionar biblioteca...”. Ao clicar neste botão, o diálogo de inserção de bibliotecas aparecerá. Nele existirão três bibliotecas configuradas, entretanto o NetBeans vem com várias bibliotecas prontas para usar. Você se lembra que quando criamos esse projeto no início da aula (Seção 3.2), nós informamos que queríamos que as bibliotecas do nosso projeto fossem armazenadas dentro dele? Por isso que temos apenas três bibliotecas configuradas, que já foram importadas automaticamente para o nosso projeto. Se quisermos usar outra biblioteca que é disponibilizada no NetBeans, precisamos importá-la. No diálogo que está aberto, clique no botão “Importar...”. Será aberto outro diálogo com as bibliotecas disponíveis. Procure pela JSTL 1.1, clique nela para selecioná-la e então clique no botão “Importar Biblioteca”. Fazendo isso, o NetBeans vai fazer uma cópia dessa biblioteca para o nosso projeto, mas ainda não vai referencia-la para podermos usar, mostrando novamente o diálogo anterior. Agora, selecione a JSTL que já foi importada e clique no botão “Adicionar biblioteca”. Ao fazer isso, o NetBeans vai referenciar essa biblioteca para podermos usar no nosso projeto. Expanda a pasta das bibliotecas para ver que a JSTL 1.1 foi inserida (são dois arquivos .jar: standard,jar e jstl.jar).

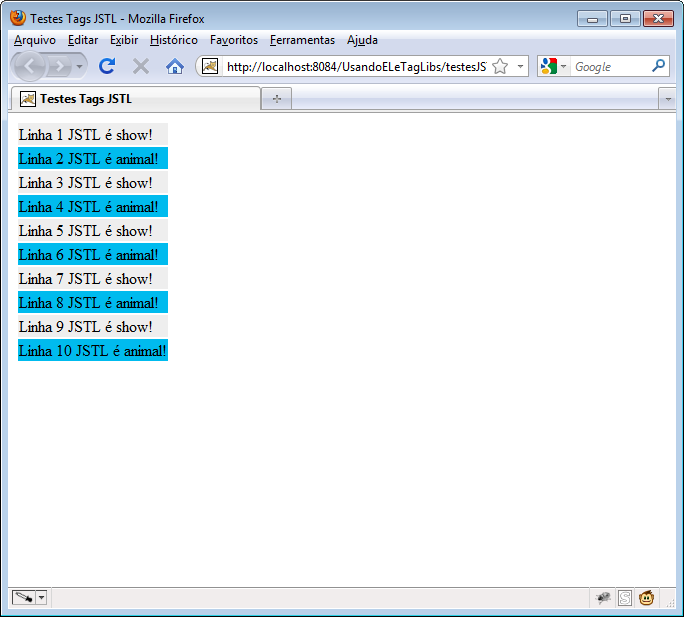
Fazendo isso, podemos agora usar a JSTL. Vamos fazer um exemplo bem simples. Iremos criar um laço de repetição “*for*” usando *tags* da JSTL. Crie mais uma página JSP, com o nome de “testesJSTL”. O NetBeans vai abrir o arquivo quando este for criado. Vamos editá-lo? Copie então o código da Listagem 3.7.

Listagem .7: Exemplo de uso da JSTL

Fonte: do autor

Copiou? Testou? Se tudo deu certo, você deve ter visto uma tabela zebrada (cor sim / cor não). Veja a Figura 3.3.

Figura .3: Visualização da página testesJSTL.jsp



Fonte: *Print screen*, Windows 7

Vamos analisar o código da Listagem 3.7. Talvez você tenha se assustado, mas não se preocupe, estou aqui para te explicar. Vamos começar pela primeira linha. Nessa linha, como em todos os JSPs que criamos até agora, usamos a diretiva “*page*”. As diretivas nos JSPs são delimitadas por “<%@” e “%>” e são usadas para realizar algumas configurações no Servlet que será gerado a partir do JSPs. A diretiva “*page*”, no nosso caso, é usada para configurar o *response* do Servlet, informando que ele vai conter um documento do tipo “text/html” (propriedade *contentType*) e que o *encoding* utilizado (como os caracteres são codificados) é o UTF-8 (propriedade *pageEncoding*).

Na linha 2 utilizamos a diretiva “*taglib*”. Essa diretiva vai permitir que nós digamos qual TagLib queremos utilizar. A propriedade “*uri*” é usada para informarmos qual biblioteca de *tags* queremos utilizar. No nosso caso, queremos utilizar as funcionalidades principais da JSTL, que são chamadas de “*core*”. A URI para definir isso é a “http://java.sun.com/jsp/jstl/core”. A outra propriedade, “*prefix*” (prefixo), nos permite definir um prefixo para usar as *tags*. O prefixo padrão para a parte “*core*” da JSTL é “c”, mas podemos usar o prefixo que quisermos. Para manter o padrão, iremos usar o “c” mesmo. Assim, quando qualquer desenvolvedor Web que conheça a JSTL bater o olho no código e ver alguma *tag* que inicie com “c:” vai saber que a *tag* que está sendo utilizada faz parte do *core* da JSTL.

Entre as linhas 12 e 22 definimos duas classes CSS. Sendo que uma usaremos para colorir o fundo das linhas pares de uma tabela, enquanto a outra será usada para colorir o fundo das linhas ímpares. A propriedade usada em ambas as classes é a “*background*” (fundo), sendo que em cada uma usamos uma cor diferente usando a notação RGB (*Reg Green Blue*) em hexadecimal. Nas linhas 28 e 43, delimitamos uma tabela.

<Saiba mais:

Nunca ouviu falar de cores na notação em hexadecimal? Veja os links abaixo!

http://dematte.at/colorPicker/

http://colorschemedesigner.com/

http://pt.wikipedia.org/wiki/Tripleto\_hexadecimal

http://en.wikipedia.org/wiki/Web\_colors (em inglês)

>

Na linha 29, usamos a *tag* *c:forEach* (olhe o prefixo!), usada para iterar um determinado número de vezes ou sobre alguma lista de objetos. No nosso caso, queremos que o que está entre <c:forEach> e </c:forEach> seja executado dez vezes, pois definimos que a iteração deve iniciar em 1 (usando a propriedade “*begin*”) e ir até 10 (usando a propriedade “*end*”). Queremos também que o *status* da iteração seja armazenado na variável “i”, definida na propriedade “*varStatus*”. Então temos um *for* que vai executar dez vezes. Durante estas dez iterações, nós vamos construir nossa tabela, inserindo linhas nela com apenas uma coluna, mas queremos que as linhas pares sejam coloridas usando a classe linhaPar, enquanto que as linhas ímpares sejam coloridas usando a classe linhaImpar. Sabemos que todo número par tem resto igual à zero numa divisão por dois, correto? Então precisamos saber em qual iteração estamos, calcular o resto e verificar se é zero. Se for, cria uma linha da tabela usando a classe linhaPar, caso contrário, usa linhaImpar.

Para criar séries de testes lógicos como numa estrutura *if* / *else if* / *else*, nós usamos a *tag* *c:choose* (*to choose* = escolher) e dentro dela colocamos as condições que queremos testar usando a *tag* *c:when* que é equivalente aos *if’s* e *else if’s* e, por fim, se necessário, usamos a *tag* *c:otherwise* que é equivalente ao *else*. Vamos analisar o código então: entre as linhas 30 e 41, nós definimos nossa estrutura condicional usando a *tag* *c:choose*. Dentro dela, definimos na linha 31 uma *tag* *c:when*, que testa (propriedade “*test*”) se a divisão do atributo “*count*” da variável “i” por dois é igual a zero (par). Note o uso da EL e que podemos executar operações aritméticas dentro dela! Se o resultado for *true*, o número é par e o conteúdo deste *c:when* é gerado, ou seja, uma linha da tabela usando a classe linhaPar. Caso contrário, como não temos mais nenhum *c:when*, é gerado o código dentro do *c:otherwise*, que por sua vez também gera uma linha da tabela, só que usando a classe linhaImpar. Fique à vontade para mudar o conteúdo gerado dentro de cada linha, bem como as classes.

Viu como não é tão complicado? Na verdade é bem simples e fácil de usar, mas precisamos praticar para ficarmos craques. Depois dessa introdução à EL e a JSTL, nós já estamos quase prontos para começarmos o nosso primeiro projeto Web de verdade, mas ainda temos que formalizar algumas coisas que serão vistas na próxima aula. Novamente, não se esqueça de praticar o que fizemos até agora.

Resumo

Nesta aula nós aprendemos a criar um formulário que teve seus dados tratados por um Servlet, que por sua vez redirecionou o fluxo da aplicação para outra página JSP, utilizada para mostrar os dados informados no formulário. Com isso, tivemos uma noção do que é a EL e como ela funciona. Depois aprendemos um pouco sobre as *tags* padrão da especificação dos JSPs e, por fim, aprendemos a inserir a JSTL em nosso projeto, além de fazermos alguns testes. Na próxima aula nós vamos dar uma paradinha com os JSPs e Servlets para podermos aprender como estruturar um projeto Web que trabalha com banco de dados. Tenho certeza que você vai gostar bastante.

Atividades de Aprendizagem

1 – Explique, com suas palavras, qual a importância da EL.

2 – Justifique porque é melhor usar a JSTL ou qualquer outra TagLib ao invés de usar código Java diretamente nas JSPs.

3 – Modifique o projeto feito na atividade 2 da Aula 2 para executar da mesma forma que o projeto criado nesta aula, ou seja, o formulário deve submeter os dados para um Servlet, que por sua vez deve criar e enviar um objeto através do request para um segundo JSP, que deve exibir ao usuário usando EL.

4 – Modifique o projeto feito na atividade 3 da Aula 2 para executar da mesma forma que o projeto criado nesta aula, ou seja, o formulário deve submeter os dados para um Servlet, que por sua vez deve criar e enviar um objeto através do request para um segundo JSP, que deve exibir ao usuário usando EL.

5 – Você já sabe que uma JSP na verdade é um Servlet não é mesmo? Será então que podemos definir na *action* de um formulário o endereço de um arquivo JSP ao invés de um Servlet? Crie um novo projeto Java Web, chamado “JSPTrataFormulario”, onde você deve ter um formulário no index.jsp que contenha dois campos: nome e idade. A *action* deste formulário deve apontar para um arquivo JSP chamado “exibeDadosForm.jsp” (sem as aspas). Nesse arquivo, você deve mostrar os dados recebidos do formulário do index.jsp usando EL. Dica: para acessar os parâmetros do request usando EL, usa-se ${param.nomeDoParametro}. Por exemplo, o parâmetro idade é acessado usando ${param.idade}.

6 – Crie um novo projeto Java Web, com o nome de “TabelaArbitraria” (sem as aspas e sem acentos), onde no index.jsp você deve ter uma formulário que pede ao usuário que seja digita a quantidade de linhas e de colunas que ele deseja que uma tabela seja gerada. Aponte a *action* deste formulário para outro arquivo JSP, chamado “montadorTabela.jsp”, que obtém os dados enviados pelo formulário do index.jsp usando EL e usa dois *c:forEach* aninhados para construir a tabela de dimensões arbitrárias. Monte a tabela somente se o tamanho de linhas e de colunas for maior que zero. Se não for, exiba uma mensagem ao usuário. Dica: para testar se duas condições são verdadeiras, ou seja, se param.colunas > 0 e param.linhas > 0, use o operador “and” da EL, enquanto o operador maior que é o “gt”. Ou seja, test=“${(param.linhas gt 0) and (param.colunas gt 0)}”.

# Aula 4 – Padrões de Projeto – *Factory*, DAO E MVC

**Objetivos**

Entender e aplicar o Padrão *Factory*.

Entender e aplicar o Padrão DAO.

Entender e aplicar o Padrão MVC.

Aprender a integrar banco de dados com uma aplicação Java.

## Introdução

Nesta aula iremos dar um passo importantíssimo em nossos estudos sobre desenvolvimento de *software*, pois iremos aprender a integrar uma base de dados em um sistema de testes que iremos construir. Isso nos dará o embasamento necessário para que na próxima aula nós consigamos utilizar uma base de dados em aplicações Web. Além de aprendermos a conectar nossa aplicação com uma base de dados, iremos aprender também alguns padrões de projeto que nos ajudarão a organizar nossa aplicação de forma a melhorar sua manutenção.

Antes de começarmos a discussão e a implementação desses padrões, nós precisamos preparar nosso ambiente de desenvolvimento. O NetBeans já temos instalado. O SGBD que iremos utilizar, o MySQL, você provavelmente já deve ter instalado também. Com isso pronto, precisamos instalar uma ferramenta para nos ajudar a criar nossa base de dados. Iremos utilizar o MySQL Workbench, uma ferramenta gratuita para gerenciamento do MySQL.

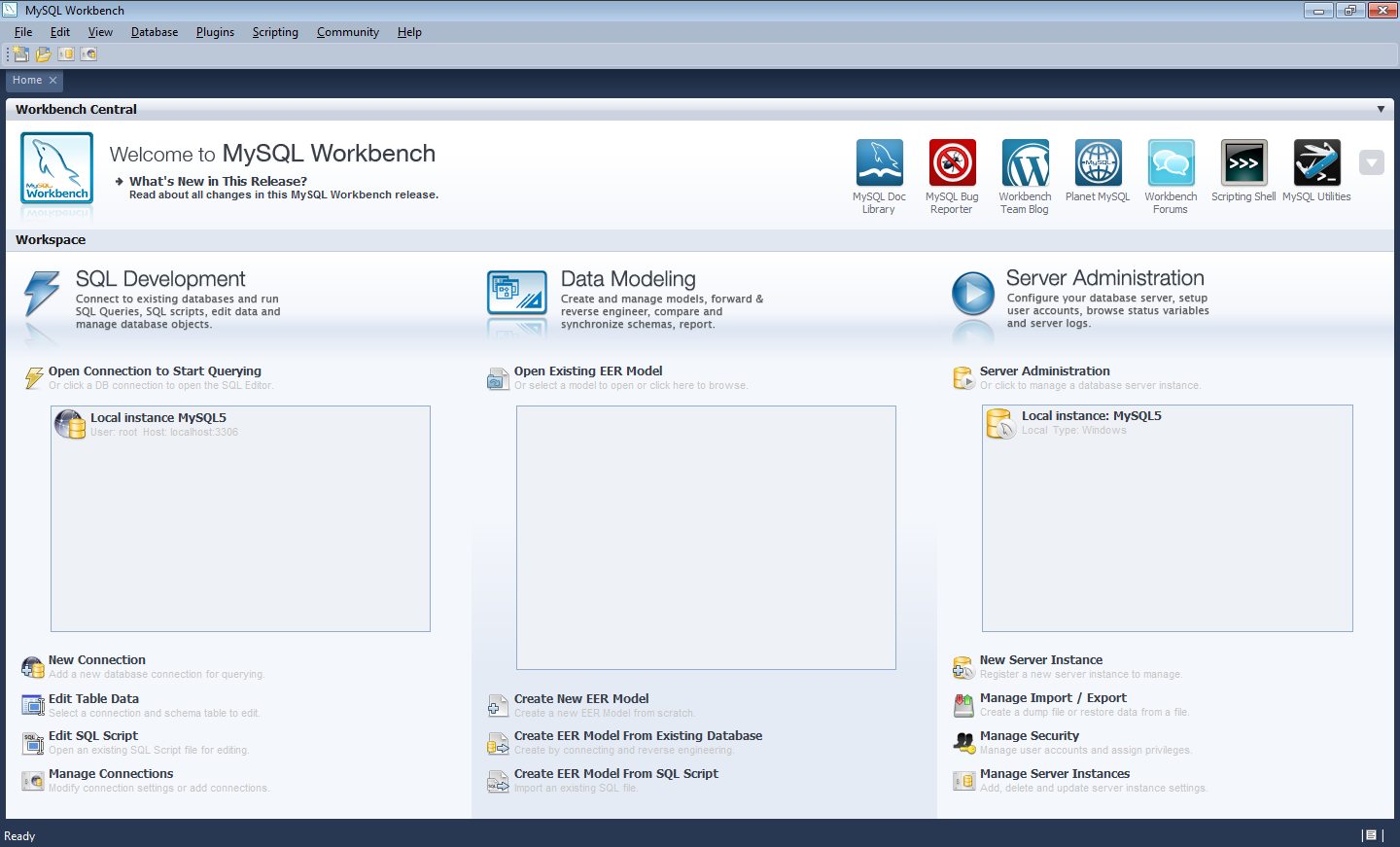
## Preparando o Ambiente

Para começar, vamos fazer o *download* da ferramenta. Acesse o endereço http://www.mysql.com/downloads/workbench/, escolha Microsoft Windows como plataforma, que provavelmente é o sistema operacional que você está utilizando. Existem três opções de *download*, clique no botão “*Download*” da primeira delas (*Windows (x86, 32-bit), MSI Installer).* Fazendo isso, você será direcionado para uma página onde é requisitado um nome de usuário e senha. Se você não quiser se cadastrar (não precisa), clique no *link* embaixo do formulário de *login* onde está escrito “*No thanks, just take me to the downloads!*”. Diversos *links* de *download* serão apresentados. Escolha qualquer um dos *links* que estão no Brasil. Podem usar os que estão como HTTP. Baixe o instalador.

Baixou? Legal! Execute o instalador. No primeiro e no segundo passos, clique em “*Next*”. No terceiro, escolha a instalação completa (“*Complete*”) e clique em “*Next*”. No último passo, escolha “*Finish*”. A instalação vai iniciar. Ao terminar, deixe marcada a opção “*Launch MySQL Workbench now*” e clique em “*Finish*”. Aguarde o MySQL Workbench abrir. Quando precisar abri-lo de novo, ele vai estar sob o menu “MySQL” do menu “Iniciar” do Windows.

A interface principal da versão 5.2 do Workbench pode ser vista na Figura 4.1.

Figura 4.1: Interface principal do MySQL Workbench



Fonte: *Print screen*, MySQL Workbench

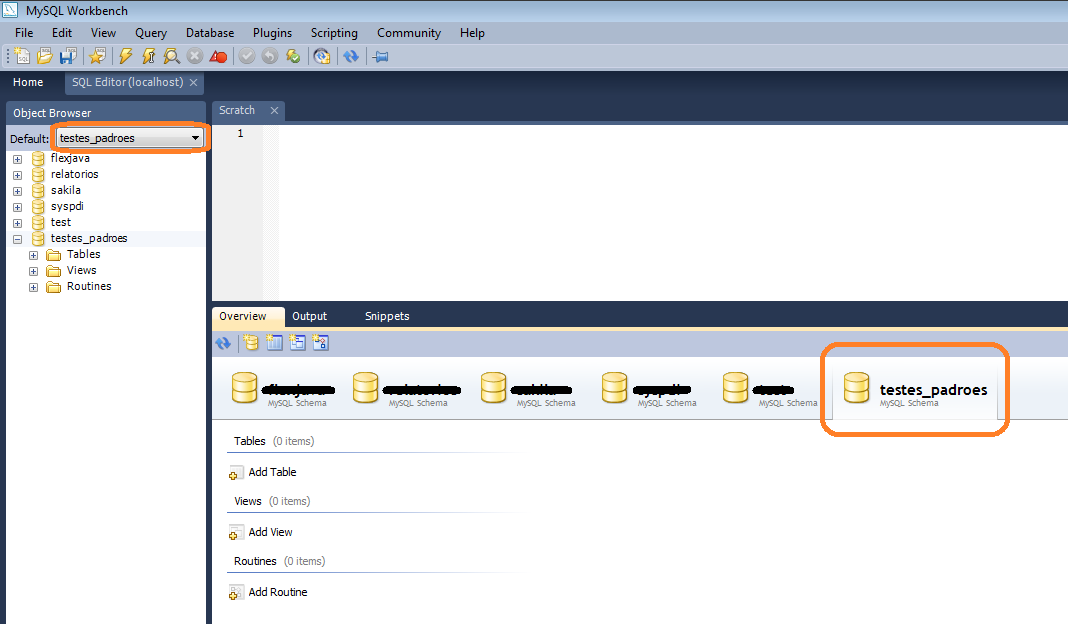
Essa interface é dividida em três partes: “*SQL Development*”, “*Data Modeling*” e “*Server Administration*”. Vamos mexer primeiro na “*Server Administration*”. Verifique se existe alguma instância configurada nessa lista, se não tiver, clique em “*New Server Instance*”. Um assistente aparecerá. Marque a opção “localhost”, pois o nosso servidor de banco de dados está instalado na nossa máquina e clique em “*Next*”. Em “*Connection Name*”, preencha com “localhost” (sem as aspas) caso não esteja preenchido. Em “*Connection Method*” escolha “*Standard (TCP/IP)*”. “*Hostname*” deve estar preenchido como “locahost” (sem as aspas) e “*Port*” como “3306” (sem as aspas). “*Username*” deve estar preenchido como “root” (sem as aspas). No campo “*Password*”, clique em “*Store in Vault...*” Ele vai pedir a senha para o usuário root. Eu normalmente defino a senha do usuário root como “root” também, mas algumas pessoas deixam a senha do root vazia. Coloque a senha que você configurou quando instalou o MySQL e clique em “OK”. Deixe o campo “*Default Schema*” vazio e clique em “*Next*”. Clique em “*Next*” novamente. Próximo passo, clique em “*Next*”. “*Next*” de novo. Antes do Workbench aplicar as configurações, se estiver tudo OK, ele vai avisar que está tudo correto e vai te perguntar se você quer continuar sem revisar as configurações. Clique em “*Continue without review*”. Último passo. O Workbench vai preencher o campo “*Server Instance Name*” com “mysqld@localhost” deixe assim mesmo e clique em “*Finish*”. Com isso a nova instância foi configurada. Se houver mais alguma na lista, apague-a clicando com o botão direito e escolhendo “*Delete Server Instance*”.

Agora vamos em “*SQL Development*”. Se você seguiu todos os passos anteriores corretamente, nessa seção da interface deve haver uma conexão chamada “localhost”. Se houver mais de uma conexão listada, apague as que não se chamarem “localhost”. Pronto? Muito bem! Sempre que abrirmos o Workbench, essas configurações já estarão feitas, não se preocupe. Agora nós vamos criar uma base de dados para trabalharmos nos exemplos desta aula.

Em “*SQL Development*”, clique duas vezes na conexão criada, chamada “localhost”. Na interface que aparecerá, verifique o lado esquerdo (“*Object Browser*”). Vão estar listados todos os esquemas do servidor de banco de dados. Clique com o botão direito na parte branca desse painel e escolha “*Create Schema*”. Preencha o campo “*Name*” com “testes\_padroes” (sem as aspas) e deixe o campo “*Collation*” como “Server Default”. Clique em “*Apply*”. Uma janela será aberta para mostrar o código SQL que será executado. Clique em “*Apply*” e depois em “*Finish*”. A janela anterior aparecerá novamente. Clique em “*Close*”. Fazendo isso, o novo esquema (vamos chamar os esquemas de base de dados) será criado e estará listado na lista “*Object Browser*”.

Logo acima do painel “*Object Browser*”, em “*Default*”, escolha a base que criamos (testes\_padroes). Na aba “*Overview*”, clique na base “testes\_padroes”. Veja a Figura 4.2.

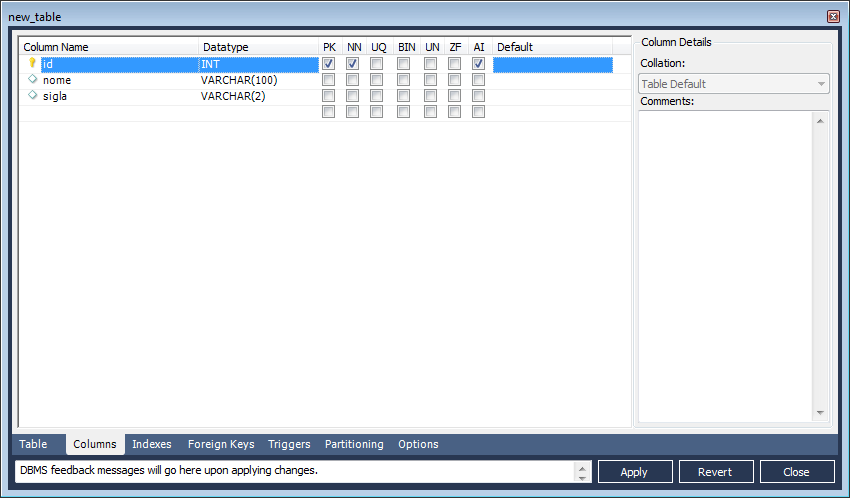
Figura .2: Selecionando a base de dados “testes\_padroes”



Fonte: *Print screen*, MySQL Workbench

Com a base “testes\_padroes” selecionada, ainda na aba “*Overview*”, clique duas vezes em “*Add Table*”. Vamos criar agora uma tabela que vai armazenar os dados de teste que iremos mexer durante esta aula. Preencha “*Name*” com “pais” (sem as aspas). Pais é país, mas vamos omitir o acento tudo bem? Deixe o valor padrão nos campos “*Collation*” e “*Engine*”. Lá no final da janela, clique na aba “*Collumns*”. O assistente já vai ter preenchido uma coluna com o nome de “idpais”. Edite esse nome e deixe apenas “id”. O tipo (*datatype*) deve ficar como INT (inteiro) e as colunas PK (chave primária), NN (*not null*) e AI (*auto-increment*) devem ficar marcadas. Essa coluna da tabela será nossa chave primária. Agora vamos às outras colunas. Nossos países terão um nome e uma sigla. Então precisamos criar mais duas colunas, uma com nome de “nome” e a outra com nome de “sigla”. A coluna “nome” terá o tipo VARCHAR(100), ou seja, um VARCHAR de 100 posições e a coluna “sigla” terá o tipo VARCHAR(2). Não vamos mexer em mais nada. Veja como ficou na Figura 4.3.

Figura .3: Criando a tabela “pais”



Fonte: *Print screen*, MySQL Workbench

Feito isso, clique em “*Apply*”. Uma janela será aberta para exibir o código SQL que será executado. Clique em “*Apply*” e depois em “*Finish*”. Por fim, clique em “*Close*”. Pronto, criamos a nossa tabela para armazenar dados dos países. Deixe o Workbench aberto e abra o NetBeans.

## Padrão de Projeto *Factory*

Primeiramente, vamos criar um novo projeto no NetBeans, só que agora ele não será um projeto Web, pois o que vamos fazer não precisa ser em um projeto desse tipo. Siga os passos que você está acostumado a fazer ao criar um projeto Java Web, só que desta vez escolha “Java” na categoria e “Aplicativo Java” nos tipos de projeto. Dê o nome de “PadroesEmPratica” para o projeto. Não se esqueça de marcar a opção “Usar pasta dedicada para armazenar bibliotecas”.

Como iremos conectar no banco de dados MySQL, precisamos adicionar no nosso projeto a biblioteca que implementa esse conector. Para nossa comodidade, o NetBeans já vem com essa biblioteca. Sendo assim, com o projeto criado, vá à pasta “Bibliotecas”, clique com o botão direito e escolha “Adicionar biblioteca...” e importe a biblioteca “MySQL JDBC Driver”. Depois de importar, selecione-a e adicione no projeto. Expanda a pasta “Bibliotecas” e veja que o arquivo .jar, que contém as classes do conector, estará inserido no projeto.

A partir de agora, podemos conectar ao banco de dados a partir do nosso código. Como você deve saber, para que possamos usar um banco de dados em Java, nós usamos uma especificação chamada JDBC (*Java Database Connectivity*). Essa especificação deve ser implementada pelos fabricantes dos SGBDs, permitindo assim que os programas feitos em Java possam se comunicar com estes SGBDs. Normalmente, esses pacotes que implementam o JDBC são chamados de “Drivers JDBC” e são obtidos nos sites dos fabricantes dos SGBDs. Como eu já disse, o NetBeans já vem com o *driver* do MySQL, então basta importarmos para o nosso projeto e utilizá-lo.

A questão agora é: “Como conectar no banco de dados?”. Para que possamos conectar no MySQL, existem uma série de “comandos” que precisamos fazer cada vez que queremos estabelecer uma conexão. Você, como desenvolvedor, sabe que uma boa prática de programação é encapsular trechos de código que fazem uma determinada tarefa em funções e que as funções em Java são chamadas de métodos.

Legal, mas e o que o tal do “padrão de projeto” tem haver com isso? Ou melhor, o que é um padrão de projeto? O termo padrão de projeto (*design pattern* em inglês) foi criado por Christopher Alexander (ALEXANDER; ISHIKAWA; MURRAY, 1977) na década de 1970 para designar soluções de sucesso para problemas recorrentes na área da arquitetura. Alexander definiu nessa época uma série de padrões que apresentavam soluções padrão para problemas que aconteciam de forma corriqueira, sendo que uma série de padrões correlatos foram o que podemos chamar de linguagem de padrões. A partir do termo cunhado por Alexander, alguns programadores começaram a criar padrões de projeto para a computação, iniciando esse trabalho do domínio da programação orientada a objetos.

<Saiba mais:

Quer aprender mais sobre padrões de projeto? Acesse os links a seguir:

http://pt.wikipedia.org/wiki/Padrão\_de\_projeto\_de\_software

http://s2i.das.ufsc.br/seminarios/apresentacoes/padroes\_de\_projeto.pdf

http://en.wikipedia.org/wiki/Design\_pattern\_(computer\_science) (em inglês)

>

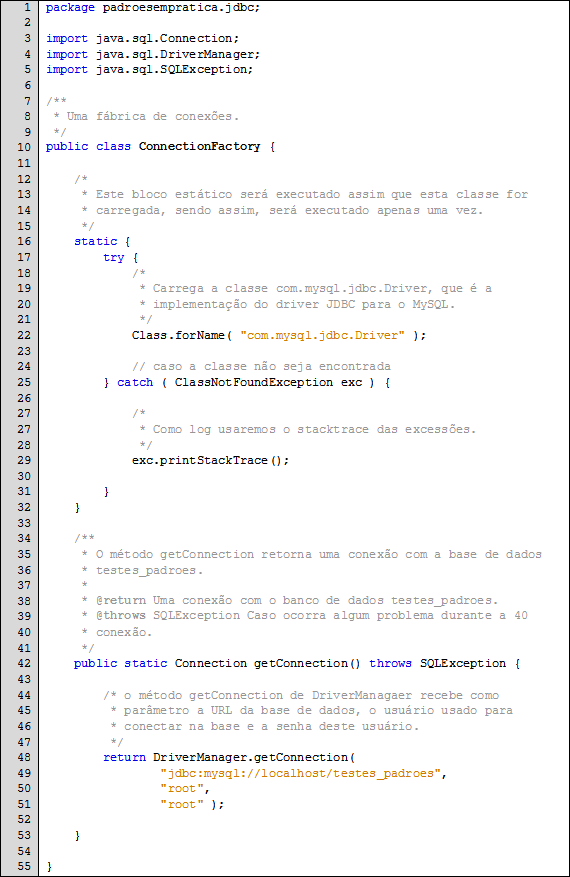
O livro “Padrões de Projeto” (GAMMA et al., 2000) é a referência básica para os padrões de projeto criados para resolver problemas relacionados ao desenvolvimento orientado a objetos. A partir de agora, quando você ler “padrão de projeto”, entenda que estarei falando de padrões relacionados ao desenvolvimento de software orientado a objetos, tudo bem? Sendo assim, o primeiro padrão que iremos aprender se chama *Factory* (fábrica).

Vamos entender o contexto do padrão. Imagine que no seu programa você precisa utilizar um determinado tipo de objeto muitas e muitas vezes e que este objeto precisa ser inicializado com uma série de valores, sendo que esses valores normalmente são sempre os mesmos. Assim, cada vez que você instância esse objeto, você precisa executar uma determinada quantidade de código. Como resolver isso? Criar um método que faça essa tarefa é uma boa solução não é mesmo? Mas em qual classe eu vou escrever esse método? No padrão *Factory*, nós criamos classes especializadas que serão fábricas de objetos e que terão um método que executará a tarefa da fábrica, ou seja, criar um determinado tipo de objeto.

No nosso projeto, um tipo de objeto que usaremos muito, é um objeto que representa a conexão entre nosso programa escrito em Java e o SGBD. Sendo assim, nós precisamos de uma fábrica de conexões! Vamos implementar a fábrica? No projeto que você criou agora a pouco, o NetBeans gerou por padrão um pacote chamado “padroesempratica” dentro da pasta “Pacotes de código-fonte”. Crie dentro deste pacote outro com o nome de “jdbc” (sem as aspas) e dentro do pacote “padroesempratica.jdbc”, crie uma classe Java com o nome de “ConnectionFactory” (sem as aspas).

Essa classe conterá o método que vai fabricar a conexão. Veja o código dela na Listagem 4.1.

Listagem .1: Código da fábrica de conexões

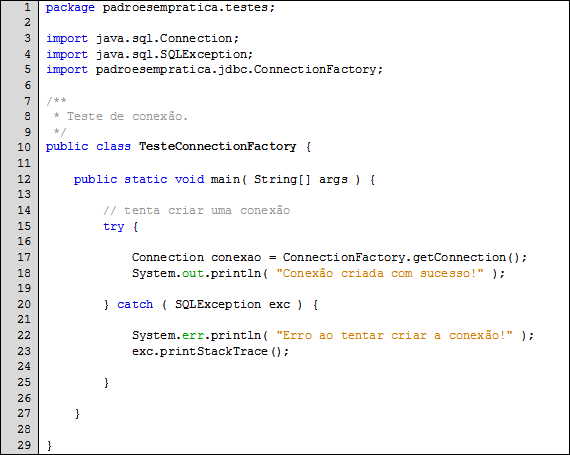


Fonte: do autor

Copiou o código? Ótimo! Analisando o código desta classe, você vai perceber que existe um bloco estático logo no início do arquivo. Blocos estáticos são executados apenas uma vez durante o ciclo de vida de um programa, pois sua execução se dá depois que a classe em questão é carregada pela máquina virtual Java. Esta classe tem um método chamado getConnection() que vai fabricar a conexão para nós e que caso ocorra algum problema, vai lançar uma exceção do tipo SQLException. Toda vez que chamarmos esse método, ele vai retornar uma nova conexão para nós.

Agora precisamos testar esse método para ver se não está havendo nenhum erro. Clique novamente com o botão direito no pacote “padroesempratica” e crie um novo pacote chamado “testes”. Dentro do pacote “padroesempratica.testes”, crie uma classe chamada “TesteConnectionFactory”. Como você deve saber, para que uma classe em Java possa ser executada, nós precisamos implementar o método “main” com uma determinada assinatura. O que vamos fazer no método “main” da classe “TesteConnectionFactory” é tentar criar uma conexão e ver se nenhum erro é retornado. Na Listagem 4.2 você pode ver o código desta classe.

Listagem .2: Classe para teste de conexão



Fonte: do autor

Copie o código para a classe e salve o arquivo. Para executarmos apenas uma classe que tem o método main, basta clicar com o botão direito no editor e escolhe a opção “Executar arquivo”, ou então, com o arquivo aberto no editor, usar o atalho Shift+F6. Fazendo isso, a classe vai ser compilada e executada pelo NetBeans. Se tudo estiver correto, você verá na saída a mensagem “Conexão criada com sucesso!”. Deu erro? Verifique se o código da classe “ConnectionFactory” está correto e se a senha do root, definida dentro do método getConnection está correta. Agora que está tudo certo, vamos simular um erro. Entre na classe ConnectionFactory e coloque uma senha inválida (terceiro parâmetro do método getConnection de DriverManagaer) para o usuário root, por exemplo, “123”. Volte na classe de testes e execute-a novamente (Shift+F6). Gerou um erro não foi? A mensagem “Erro ao tentar criar a conexão!” deve ter sido exibida, seguida de várias linhas que explicam o erro ocorrido. A primeira linha dos erros diz que o acesso foi negado para o usuário “root@localhost” não foi? Por que aconteceu isso? Porque a senha está errada! Volte na fábrica de conexões e coloque a senha correta. Teste novamente. Agora deve estar tudo certo.

Legal, temos uma fábrica de conexões, mas do que adianta uma fábrica de alguma coisa se a gente não usar o que é fabricado? Vamos para o próximo padrão, onde iremos organizar uma camada de persistência para nossa aplicação e usaremos a fábrica de conexões para viabilizar a comunicação entre nossa aplicação e o SGBD.

## Padrão de Projeto DAO – *Data Access Object*

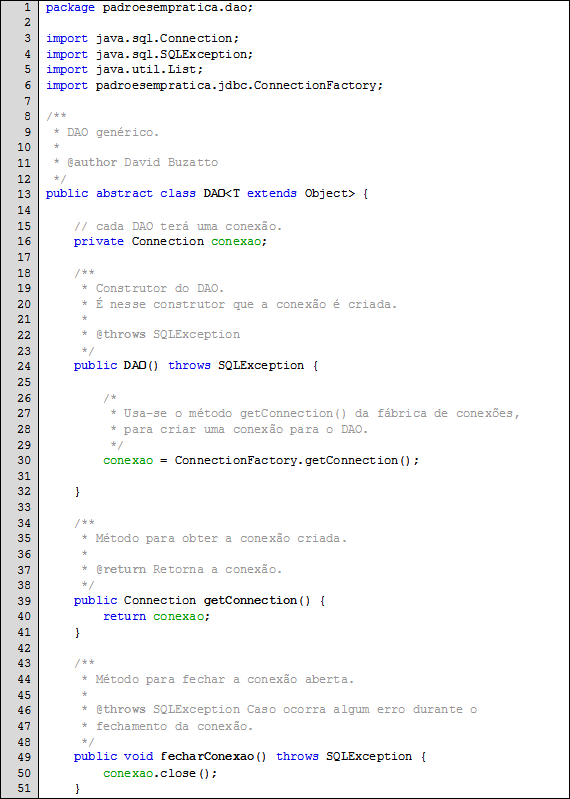
Quando temos o nosso primeiro contato com JDBC, normalmente nos é ensinado a colocar todo o código SQL que vai executar uma determinada operação em um método que trata o “clique” de um botão. Sendo assim, imagine uma interface *desktop* (em Swing) onde poderíamos criar, alterar e/ou excluir países. Quando implementamos o botão responsável por criar um novo país, somos ensinados a inserir todo o código SQL para fazer isso, sendo que esse código é implementado usando uma cláusula INSERT. O mesmo aconteceria para os botões alterar e excluir. Será que essa é a melhor solução? Será que a classe que implementa nossa interface gráfica tem que ter essa responsabilidade, ou seja, lidar com SQL? E se por algum motivo nós quiséssemos usar o código para criar um novo país em alguma outra tela? Teríamos que copiar o código de inserção novamente? Tenho certeza que para essa última pergunta você já deve ter respondido mentalmente que “NÃO!”, pois podemos criar métodos que executariam essa operação, mas então te pergunto: Como fazer isso?

Para resolver esse problema, existe um padrão de projeto onde a ideia é isolar todo acesso ao banco de dados em classes que seriam responsáveis em fazer a comunicação entre a aplicação Java, ou qualquer aplicação escrita usando uma linguagem orientada a objetos, e o banco de dados. Esse padrão se chama DAO (*Data Access Object* – Objeto de Acesso a Dados), sendo este um dos mais famosos. Vamos aprender como implementá-lo?

Um objeto do tipo DAO deve ter a capacidade de executar as operações básicas sobre uma determinada tabela de um banco de dados. Essas operações são comumente chamadas de CRUD (*Create*, *Read*, *Update* e *Delete* – Criar, Ler, Atualizar e Excluir). Praticamente cada tabela do nosso banco de dados terá no lado da aplicação uma classe implementada que representa um registro da tabela (tabela “pais”, classe Pais), além de ter uma classe DAO que vai manipular objetos do tipo Pais. Como precisamos definir essas quatro operações básicas que cada DAO vai conter, vamos criar uma classe abstrata que servirá de modelo.

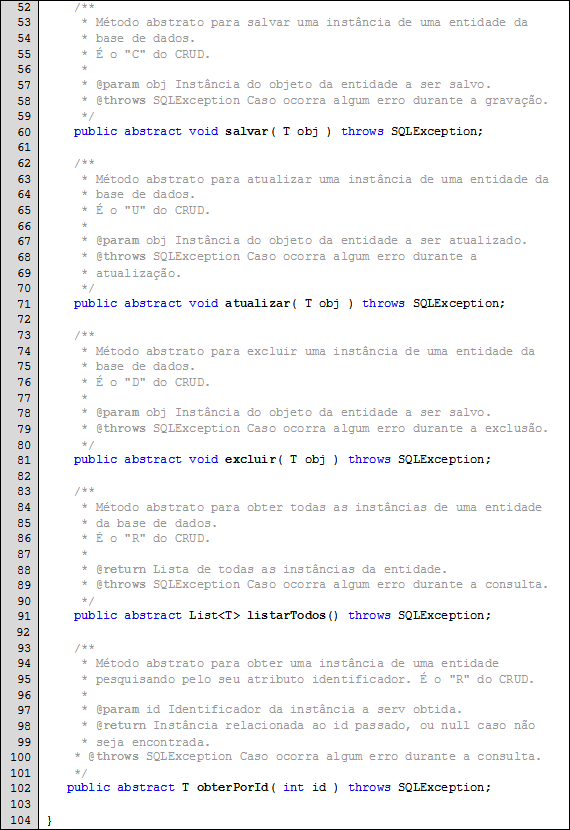
No pacote “padroesempratica”, crie um novo pacote chamado “dao”. Dentro do pacote “padroesempratica.dao”, crie uma classe chamada “DAO” e copie o código apresentado na Listagem 4.3 e na Listagem 4.4. Note que as duas listagens são partes da mesma classe.

Listagem .3: Primeira parte da implementação da classe abstrata DAO



Fonte: do autor

Listagem .4: Segunda parte da implementação da classe abstrata DAO



Fonte: do autor

Copiou o código? Ótimo! Vamos entendê-lo. Na linha 13 da Listagem 4.3 definimos uma classe abstrata (classes abstratas não podem ser instanciadas, são usadas como modelos) chamada DAO que diz que toda classe que for implementá-la, deve definir um tipo “T” que obrigatoriamente estende Object. As construções entre “<” e “>” são chamadas de “Tipos Genéricos” em Java. Note que o tipo “T” é usado em todo o corpo da classe. Está confuso? Acalme-se, logo você vai entender.

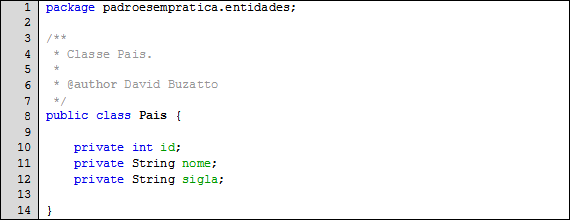
Na linha 16 da Listagem 4.3 é declarada uma conexão, que sempre será obtida usando o método getConnection definido na linha 39. Observe que poderíamos ter declarado essa conexão como *protected*, mas vamos deixá-la como *private* e usar o método getConnection() para obtê-la. Na linha 49 é definido o método para fechar a conexão, afinal, sempre que usarmos uma conexão, precisamos fechá-la depois de usar.

Veja que no construtor da classe (linha 24 da Listagem 4.3), a conexão é obtida usando a fábrica que criamos na seção anterior! Esse construtor será executado quando instanciarmos os objetos das classes que estenderem esse DAO, então, quando criarmos nossos objetos DAO, uma conexão com o banco de dados será estabelecida.

A partir da linha 60 da Listagem 4.4 são definidos todos os métodos CRUD deste DAO genérico. Note que temos dois métodos que correspondem à parte “R” do CRUD, onde um obtém todas as entidades cadastradas e outro obtém apenas uma usando como base seu identificador.

Com o DAO genérico pronto, vamos implementar a classe que vai representar a tabela “pais”. Crie um novo pacote chamado “entidades” dentro do pacote “padroesempratica”. Dentro do pacote “padroesempratica.entidades”, crie uma classe chamada “Pais” (sem as aspas). Os objetos dessa classe representarão registros da tabela “pais”, sendo assim, precisamos que essa classe tenha os mesmos atributos da tabela correspondente. Lembre-se que na tabela “pais” nós definimos três colunas: id (INT), nome (VARCHAR) e sigla (VARCHAR). Na nossa classe, iremos usar o tipo “int” como tipo correspondente ao INT da tabela. Para o tipo VARCHAR, usaremos Strings. Veja na Listagem 4.5 como deve ficar a implementação parcial da classe Pais.

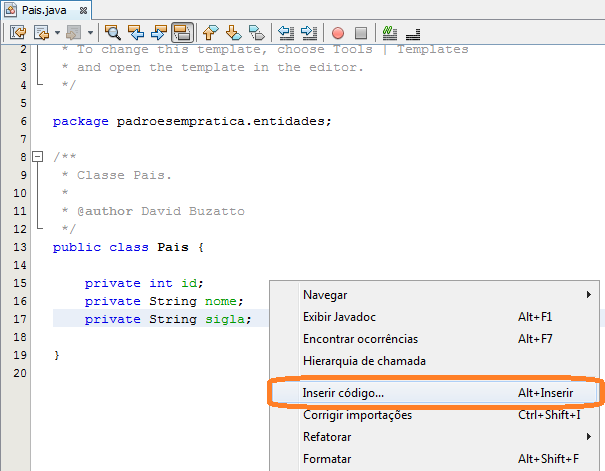
Listagem .5: Implementação parcial da classe Pais



Fonte: do autor

Tenho certeza que você se lembra da discussão sobre o padrão *JavaBeans* não é mesmo? Onde foi dito que devemos expor ao mundo “fora da classe” os atributos que nós queremos que possam ser configurados e obtidos por seus utilizadores? Da primeira vez que falamos sobre isso, nós implementamos manualmente cada método *set* e *get* correspondente a um campo privado não foi? Como essa tarefa é muito corriqueira, o NetBeans tem uma funcionalidade que faz isso automaticamente para nós. Com a classe implementada, como exibido na Listagem 4.5, clique com o botão direito no editor e escolha “Inserir código...”. Veja na Figura 4.4.

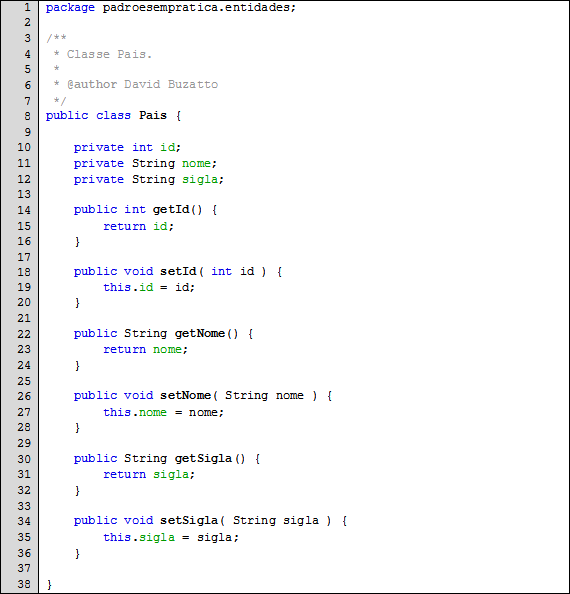
Figura .4: Usando o assistente de inserção de código



Fonte: *Print screen*, NetBeans IDE 6.9.1

Ao clicar nesta opção, uma pequena lista com o nome de “Gerar” será exibida no editor. Nessa lista, escolha a opção “Getter e setter...”. Adivinhe o que vamos fazer? Gerar os métodos *get* e *set* para cada campo da classe! Fazendo isso, um diálogo será exibido, mostrando todos os campos privados da classe. Marque cada um dos campos clicando na caixa de seleção correspondente, ou então, a classe inteira e clique no botão “Gerar”. Veja o que aconteceu! A ferramenta gerou o código dos *gets* e *sets* para nós! Segue na Listagem 4.6 o código completo da classe Pais.

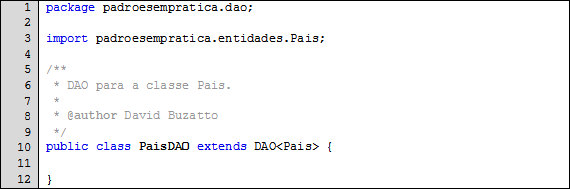
Listagem .6: Implementação completa da classe Pais



Fonte: do autor

Muito legal não é mesmo? Agora que temos a classe que representa a estrutura da tabela “pais” da nossa base de dados, vamos implementar nosso primeiro DAO concreto, ou seja, uma classe que vai estender a classe abstrata DAO. Novamente, no pacote “padroesempratica.dao”, crie uma classe chamada “PaisDAO” (sem as aspas). Essa classe irá lidar com os objetos do tipo Pais, fazendo a ponte entre nossos objetos e o banco de dados. Com a classe criada, copie o código apresentado na Listagem 4.7.

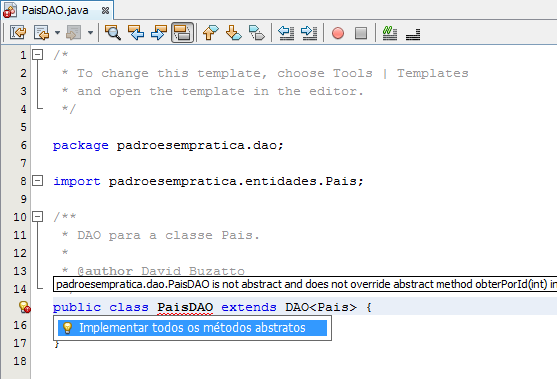
Listagem .7: Implementação parcial do PaisDAO



Fonte: do autor

Veja que nosso PaisDAO vai estender DAO, informando como tipo a classe Pais (DAO<Pais>). Ao copiar o código, você perceberá que o NetBeans vai reclamar, dizendo que tem um erro na classe. O nome da classe ficará destacado em vermelho. Passe o *mouse* por cima do nome e aguarde. Será exibida a causa do erro. No erro, é dito que a classe PaisDAO não implementa todos os métodos abstratos da classe DAO e isso é verdade, visto que como estamos estendendo a classe DAO, precisamos implementar todos os métodos abstratos que foram definidos nela e ainda não fizemos isso. Teríamos então que implementar manualmente todos os métodos marcados como abstratos na classe DAO. Ao invés de fazermos isso manualmente, o NetBeans pode nos ajudar novamente. Veja que na linha do erro, à esquerda, é mostrada uma pequena lâmpada. Clique nela. Ao clicar, o NetBeans vai listar as alternativas que ele pode executar para resolver o erro. No caso, apenas uma opção será listada (“Implementar todos os métodos abstratos”) veja a Figura 4.5.

Figura .5: Implementando automaticamente os métodos abstratos de DAO



Fonte: *Print screen*, NetBeans IDE 6.9.1

Clique nesta opção e, novamente, como num passe de mágica, o NetBeans gera todo o esqueleto da classe para nós, criando uma implementação padrão para cada método abstrato da classe DAO. Mesmo ao fazer isso, o NetBeans continua reclamando que existe um erro. Nesse caso, é dito que é lançada uma exceção no construtor padrão. Você se lembra que lá no DAO genérico nós temos um construtor que cria a conexão e que ele lança uma SQLException? Pois bem, quando criamos um objeto de uma determinada classe, o construtor da superclasse da classe em questão é executado. Como estendemos DAO em PaisDAO, ao tentarmos instanciar um objeto do tipo PaisDAO, o construtor de PaisDAO será executado, além do construtor de DAO, que é sua superclasse. Como o construtor de DAO lança uma exceção caso ocorra algum problema, nós precisamos ou tratar ou dizer que o construtor de PaisDAO também lança esse tipo de exceção. Nós iremos usar a segunda abordagem. Para isso, basta implementar o construtor padrão de PaisDAO e dizer que ele lança esse tipo de exceção. Sendo assim, segue na Listagem 4.8 a implementação que temos até agora da classe PaisDAO.

<Glossário: Construtor Padrão

O construtor padrão é o construtor que não tem nenhum parâmetro.

>

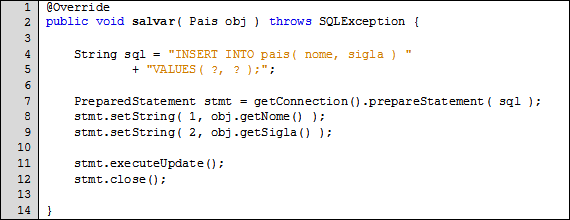
Listagem .8: Implementação parcial de PaisDAO com o construtor padrão



Fonte: do autor

Muito bom. Até agora preparamos toda o esqueleto do nosso PaisDAO, mas SQL que é bom, nada. Vamos agora implementar nosso primeiro método do CRUD, o “salvar”. Para poupar espaço, irei listar apenar o método em questão, ao invés de listar a classe completa, tudo bem? Lembre-se que estamos implementando o método “salvar” da classe PaisDAO. Antes de implementar o método “salvar”, importe a classe “java.sql.PreparedStatement”. Na Listagem 4.9 pode ser vista a implementação do método “salvar” da classe PaisDAO.

Listagem .9: Implementação do método “salvar” da classe PaisDAO

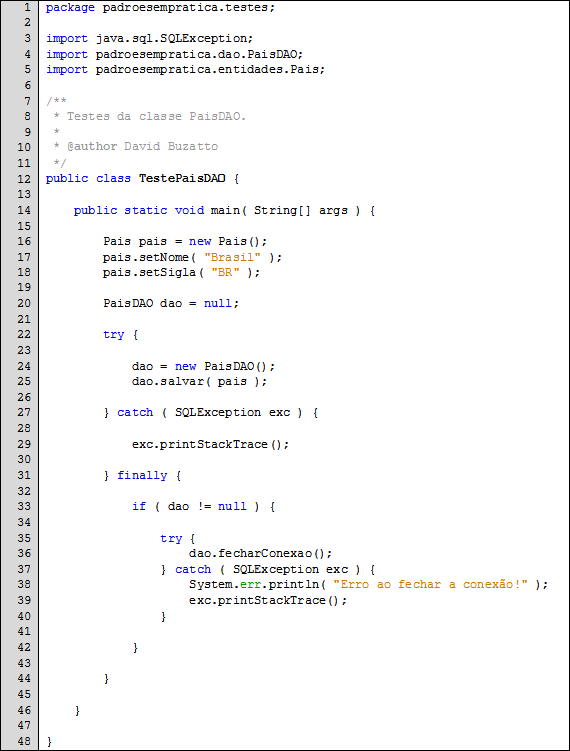


Fonte: do autor

Vamos analisar o código para ver o que está acontecendo. Na linha 2 está definida a assinatura do método. O método “salvar” recebe um parâmetro do tipo Pais, sendo que os dados do objeto passado nesse parâmetro serão persistidos no banco de dados. Nas linhas 4 e 5 é definida a cláusula INSERT do banco de dados. Como a coluna id de pais foi definida como auto incremento, nós não precisamos fornecê-la. Ao invés de definirmos manualmente os valores das colunas nome e sigla, perceba que utilizamos sinais de interrogação, indicando que no lugar de cada ponto de interrogação será trocado pelo valor correto. Na linha 7 é criado um PreparedStatement a partir da conexão do DAO usando o código SQL que foi definido nas linhas 4 e 5. Na linha 8, o primeiro parâmetro do código SQL (primeiro ponto de interrogação) é trocado pelo valor retornado pelo método getNome() do objeto “obj” do tipo Pais. A mesma coisa acontece na linha 9, onde o segundo parâmetro do código SQL (segundo ponto de interrogação) é trocado pelo valor retornado pelo método getSigla(). Com o PreparedStatement configurado, na linha 11 mandamos que seja executado o PreparedStatement. Por fim, na linha 12, fechamos o PreparedStatement. Fácil não é?

Vamos testar? No pacote “padroesempratica.testes”, crie uma classe chamada TestePaisDAO e copie o código da Listagem 4.10.

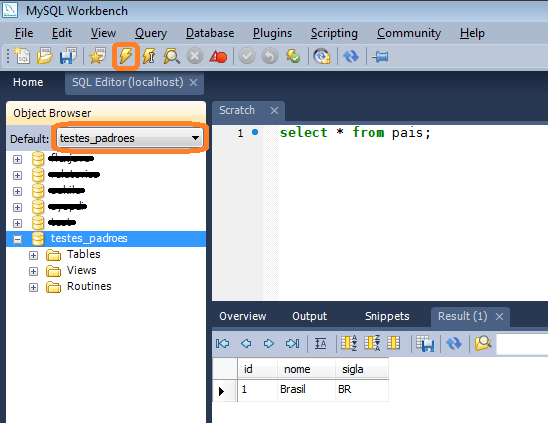
Listagem .10: Código de teste para o PaisDAO



Fonte: do autor

Copiou o código? Execute a classe (botão direito no arquivo, “Executar arquivo” ou Shift+F6). Se tudo estiver correto, a classe será compilada e executada e nenhum erro será emitido. Fazendo isso, um novo registro na tabela “pais” será gerado. Vamos confirmar isso? No MySQL Workbench, abra o editor de SQL clicando duas vezes na conexão definida na seção “SQL Development”. Provavelmente o editor já deve estar aberto. Em “Object Browser” selecione a base “testes\_padroes” em “Default”. Em “Scratch”, entre digite “select \* from pais;” (sem as aspas) e clique no botão que tem um desenho de um raio na barra de tarefas. O código SQL digitado será executado e o resultado será exibido logo abaixo. Veja a Figura 4.6.

Figura .6: Fazendo uma consulta do MySQL *Workbench*

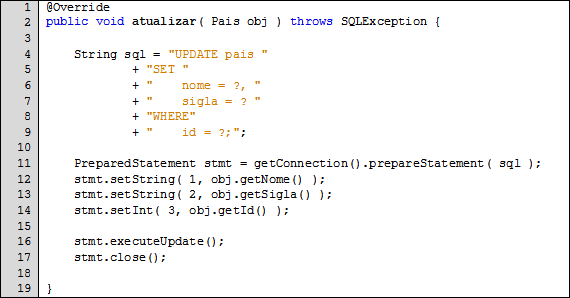


Fonte: *Print screen*, MySQL Workbench

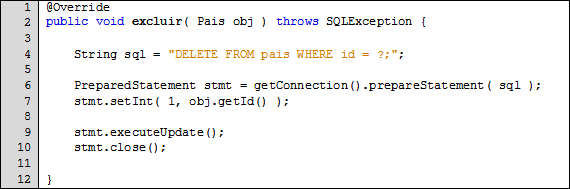
Muito bem! Nosso método salvar está funcionando corretamente. Vamos analisar o código da Listagem 4.10. Entre as linhas 16 e 18 instanciamos e configuramos um objeto do tipo Pais. O nome desse país é Brasil e a sigla é BR. Na linha 20, declaramos uma referência do tipo PaisDAO que foi inicializada com null. Como todo o código dos métodos do DAO podem lançar uma exceção do tipo SQLException, temos que usar um bloco *try*. Na linha 24 instanciamos o PaisDAO e na linha 25 passamos o objeto do tipo Pais que criamos para o método “salvar” do DAO, que por sua vez vai executar o código SQL que definimos. Caso ocorra algum problema durante a execução de uma dessas duas linhas, o *catch* que ouve exceções do tipo SQLException captura o erro e manda mostrar esses problemas dando um printStackTrace() no objeto que representa a exceção. Por fim, temos um bloco *finally* que trata do fechamento da conexão. Sempre quando usarmos um DAO, precisamos fechar sua conexão quando terminarmos de usar. Na linha 33 verifica-se se o “dão” aponta para null. Se não apontar, tenta fechar a conexão, que também pode lançar uma exceção do tipo SQLException e que é tratada dentro do bloco *try* que está aninhado no *finally*.

Agora que já criamos e testamos nosso primeiro método do PaisDAO, vamos implementar o restante dos métodos. Cada um deles será apresentado em uma listagem diferente. Os métodos de pesquisa (“R” do CRUD) serão um pouco diferente, sendo assim, eu os discutirei depois. Vamos lá!

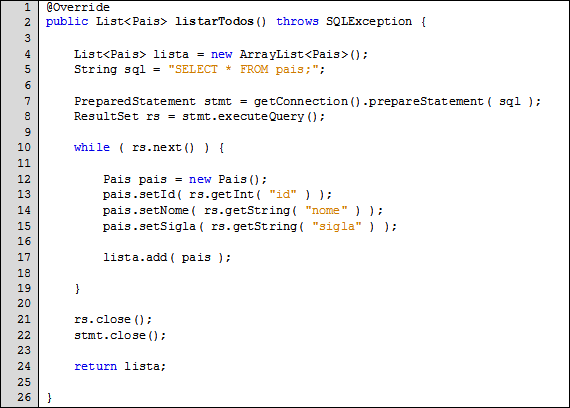
Listagem .11: Implementação do método “atualizar” da classe PaisDAO

Fonte: do autor

Listagem .12: Implementação do método “excluir” da classe PaisDAO

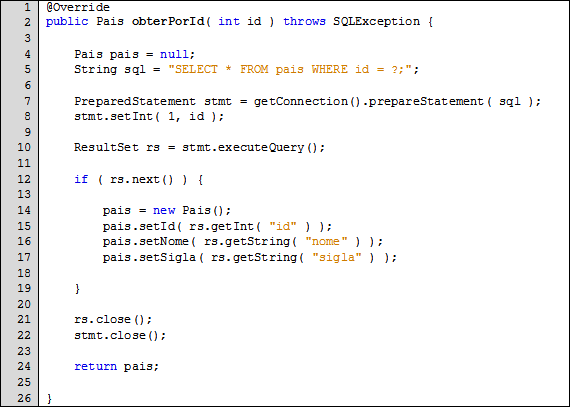
Fonte: do autor

Listagem .13: Implementação do método “listarTodos” da classe PaisDAO



Fonte: do autor

Listagem .14: Implementação do método “obterPorId” da classe PaisDAO



Fonte: do autor

Não se esqueça de importar as classes java.sql.ResultSet, java.util.ArrayList e import java.util.List. Para finalizar essa seção, vamos discutir o código da Listagem 4.13. O método listarTodos retorna uma lista que pode conter apenar objetos do tipo Pais. Essa lista que será retornada é instanciada e inicializada na linha 4. Neste momento, temos a lista de objetos do tipo Pais, só que ela ainda está vazia. Na linha 7, cria-se o PreparedStatement com o SQL que foi definido na linha 5 e, na linha 8, executamos o PreparedStatement usando o método executeQuery(), que retorna um objeto do tipo ResultSet. Os objetos do tipo ResultSet representam os resultados que são retornados em consultas SQL (comando SELECT). Na linha 10, criamos um laço de repetição *while*, que é executado enquanto rs.next() retornar *true*. Na primeira vez que rs.next() é invocado, o ponteiro de registros do ResultSet passa a apontar para o primeiro resultado obtido na consulta. Dentro do *while*, entre as linhas 12 e 15, nós criamos um objeto do tipo Pais e configuramos seus dados a partir dos dados obtidos no ResultSet, usando o método apropriado para cada tipo de cada coluna. Na linha 17, o objeto Pais que acabou de ser criado e configurado, é inserido na lista. O corpo do *while* é repetido até que rs.next() retorne *false*, ou seja, quando todos os resultados da consulta foram obtidos. Nesse ponto, temos a lista de objetos totalmente preenchida. Nas linhas 21 e 22, são fechados o ResultSet e o PreparedStatement. Por fim, na linha 24, a lista com os objetos do tipo Pais é retornada.

Note que o método “obterPorId”, definido na Listagem 4.14, tem uma implementação muito parecida com o método “listarTodos”, só que no caso do “obterPorId”, apenas um objeto pode ser retornado, visto que cada registro da tabela pais tem um identificador específico. Caso o id passado como parâmetro não represente um registro da tabela pais, o método retorna null.

Agora, como exercício, modifique a classe TestePaisDAO para testar todos os métodos que foram implementados. Verifique se todos estão corretos usando o MySQL Workbench para comparar os resultados obtidos. Tente entender a implementação de cada método do CRUD.

Não fez o exercício? Então faça! Ele não é opcional.

Viu só que legal? Com tudo isso que fizemos, conseguimos isolar toda a camada de persistência da nossa aplicação. Para cada tabela nova que tivermos na base de dados, precisamos implementar a classe correspondente à tabela (que chamamos de entidade) e a classe DAO que vai manipular os objetos da classe criada e fazer o intercâmbio entre o mundo orientado a objetos com o mundo relacional. Essa comunicação entre objetos e o modelo relacional, é chamada de ORM (*Object-Relational Mapping* – Mapeamento Objeto-Relacional). Existem diversos *frameworks* que automatizam esse processo, gerando todo o código SQL automaticamente para nós. Infelizmente não teremos tempo para aprendê-los. Um desses *frameworks* é o Hibernate.

<Saiba mais:

Quer saber mais sobre ORM? Acesse os links a seguir:

http://pt.wikipedia.org/wiki/Mapeamento\_objeto-relacional

http://en.wikipedia.org/wiki/Object-relational\_mapping (em inglês)

http://www.hibernate.org/

>

Estamos quase acabando! Vamos para o último padrão de projeto.

## Padrão de Projeto MVC – *Model-View-Controller*

O padrão MVC (*Model-View-Controller* – Modelo-Visualização-Controle) é um padrão muito utilizado no desenvolvimento de aplicações que utilizam linguagens orientadas a objetos. Este padrão ajuda os desenvolvedores a separar as regras de negócio da aplicação, ou seja, as regras de como os dados devem se armazenados, qual a ordem que devem ser gravados, etc., da lógica de apresentação desses dados, ou seja, como eles serão exibidos aos usuários do sistema. Essa separação se dá utilizando três camadas:

* *Model* (Modelo): A camada *Model* é usada para organizar como a informação é armazenada e gerenciada dentro da aplicação. No exemplo que construímos durante esta aula, as classes Pais e PaisDAO fazem parte do modelo.
* *View* (Visualização): Essa camada organiza os recursos utilizados para exibir ao usuário os dados que são gerenciados pela camada de modelo.
* *Controller* (Controle): A camada *Controller*, como o próprio nome já diz, é responsável por controlar o fluxo de execução da aplicação.

A partir dessas definições, nós podemos ver nitidamente, nos exemplos que estamos construindo desde o início da disciplina, qual recurso faz parte de cada camada. A entidade Pais faz parte do modelo. Uma JSP faz parte da visualização, pois é usada pelo usuário tanto para inserir dados no sistema quanto para visualizá-los. Os Servlets que construímos fazem parte do controle, pois são eles que recebem os dados, os utilizam usando a camada de modelo e decidem para onde o fluxo da aplicação deve ser direcionado, por exemplo, uma página JSP que vai exibir a saída gerada por eles.

Essa foi uma pequena introdução do MVC, pois durante a próxima aula nós iremos usá-lo extensivamente no projeto que iremos criar. Tenho certeza que você vai achar muito legal e útil! Como de costume, pratique o que você aprendeu durante esta aula, com as atividades de aprendizagem.

Resumo

Nesta aula demos um passo muito importante na nossa vida como desenvolvedores. Nós aprendemos que existem os chamados “padrões de projeto” ou “*design patterns*”, que são padrões que guiam os desenvolvedores na solução de problemas recorrentes através do uso de soluções de sucesso. Estudamos os padrões *Factory* e DAO implementando exemplos e aprendemos o básico do funcionamento do padrão MVC. Na próxima aula iremos implementar um projeto completo usando esses três padrões.

Atividades de Aprendizagem

1 – Defina, com suas palavras, o padrão *Factory*.

2 – Defina, com suas palavras, o padrão DAO.

3 – Defina, com suas palavras, o padrão MVC.

4 – Da mesma forma que fizemos para a tabela “pais”, crie uma tabela no MySQL, usando o Workbench, com o nome de “fruta”. Essa tabela deve ter como colunas um campo identificador (id), um campo que armazenará o nome da fruta (VARCHAR) e um campo para armazenar a cor predominante da fruta (VARCHAR). Implemente, no projeto que criamos durante a Aula 4, a entidade Fruta e a classe FrutaDAO. Crie uma classe de testes chamada TesteFrutaDAO para testar os métodos do DAO da fruta.

5 – Repita a atividade 4, só que agora para a tabela “carro”. Um carro deve ter um identificador, um nome (VARCHAR), um modelo (VARCHAR) e um ano de fabricação (INT).

6 – Repita a atividade 4, só que agora para a tabela “produto”. Um produto deve ter um identificador, uma descrição (VARCHAR) e uma quantidade em estoque (INT).

# Aula 5 – Primeiro Projeto – Sistema para Controle de Clientes

**Objetivos**

Entender e realizar a construção de uma aplicação Web em Java completa.

## Introdução

Nesta aula iremos colocar em prática tudo o que aprendemos nas aulas anteriores com o objetivo de criar uma aplicação Web em Java completa. Iremos passar por todos os passos do desenvolvimento da aplicação para que na próxima aula, você possa usar um conjunto de requisitos para desenvolver um sistema sozinho. Vamos começar?

## Analisando os Requisitos

Imagine que fomos contratados para criar um sistema para controle de cadastro de clientes. Esse sistema deve manter vários dados de um cliente: nome, sobrenome, data de nascimento, CPF, e-mail, logradouro, número, bairro, cidade e CEP. O contratante também deseja que seja possível manter um cadastro de cidades e de estados, sendo que as cidades devem ter um nome e um estado, enquanto um estado deve ter um nome e uma sigla. Cada um dos cadastros (cliente, cidade e estado), deve conter as funcionalidades de criar, alterar e excluir um determinado registro.

Vamos analisar esses requisitos. Primeiramente, vamos identificar os tipos de entidades que farão parte do sistema, fazendo a seguinte pergunta: Quais são os tipos de “coisas” que o sistema deve gerenciar? O sistema deve manter um cadastro de Clientes, um cadastro de Cidades e um cadastro de Estados. Sendo assim, identificamos três entidades, ou seja, Cliente, Cidade e Estado.

Cada um desses tipos de entidade tem uma determinada lista de características. Vamos organizá-las em uma tabela. Veja essa organização na Tabela 5.1.

|  |  |
| --- | --- |
| Tabela .1: Características dos tipos de entidade | |
| **Entidade** | **Atributos (características)** |
| Cliente | - nome;  - sobrenome;  - data de nascimento;  - CPF;  - e-mail;  - logradouro;  - número;  - bairro;  - Cidade;  - CEP. |
| Cidade | - nome;  - Estado. |
| Estado | - nome;  - sigla. |

Fonte: do autor

Sabemos que esses tipos de entidade que foram identificados se tornarão tabelas na nossa base de dados relacional não é mesmo? Cada característica de cada tipo de entidade se tornará uma coluna na tabela correspondente. Sabemos também que cada registro de uma determinada tabela precisa ser diferenciado dos outros não é mesmo? Para isso analisamos as tabelas até que consigamos identificar as chaves primárias de cada uma delas. Uma chave primária é o conjunto mínimo de um ou mais atributos de um determinado tipo de entidade que garante a unicidade de um registro, sendo assim, precisamos encontrar, na lista de características de cada entidade, uma ou mais características que, usadas em conjunto, garantem que um registro é diferente de outro. Tomemos como exemplo o tipo de entidade Estado. Veja na Tabela 5.2 uma lista de registros da tabela “estado” do nosso provável banco de dados.

|  |  |
| --- | --- |
| Tabela .2: Exemplos de registros da tabela “estado” | |
| **Tabela: estado** | |
| **nome** | **sigla** |
| São Paulo | SP |
| Rio de Janeiro | RJ |
| Minas Gerais | MG |
| ... | ... |

Fonte: do autor

O que diferencia um estado de outro? Se usarmos os atributos “nome” e “sigla”, nós sempre teremos um estado diferente do outro não é mesmo? E se usarmos apenas o “nome”? Ou então somente a “sigla”? Temos então três opções para definir a chave primária dessa tabela. Podemos usar “nome” e “sigla”, somente “nome” ou somente “sigla”. Dentre essas três opções, qual ou quais são as que têm o menor conjunto de atributos? Somente “nome” ou somente “sigla”, correto? Como temos duas opções, podemos escolher qualquer uma delas. Vamos dizer que nós escolhemos a opção de definir a chave primária da tabela usando o atributo “sigla”. Legal, agora sabemos que um estado é diferenciado do outro pela sua sigla, então não pode existir mais de um registro com a mesma sigla.

Agora temos outro problema: desempenho do banco de dados. Quando criarmos a tabela “cidade”, esta vai ter que referenciar a tabela “estado” usando uma coluna que vai ter o mesmo tipo da coluna que representa a chave primária da tabela “estado”. Essa coluna, como você deve se lembrar, é denominada chave estrangeira. Como cada estado tem uma sigla de dois caracteres, sempre que um estado for referenciado em um registro da tabela “cidade”, essa referência vai ter que ter o mesmo valor da tabela sigla. Apesar de essa abordagem funcionar, nós podemos atacar esse problema de outra forma. Podemos definir que a coluna “sigla” tem valor único (*unique*) nos registros da tabela “estado”, e então criar uma chave primária que contém apenas um número. Essa chave primária é chamada de chave artificial, ou *surrogate*, sendo que normalmente é chamada de “id” (identificador). Note que como o próprio nome diz, essa chave é artificial. O que vai garantir a unicidade dos registros é a configuração de cada uma das colunas.

O que ganhamos com isso? Ganhamos desempenho, pois estamos usando números para referenciar colunas de outras tabelas, não Strings. Imagine definir a chave primária de um cliente como CPF. Ao precisarmos referenciar um cliente em outra tabela, digamos uma tabela de “pedidos”, precisaríamos ter uma cópia do CPF do cliente em cada pedido, gastando cerca de 14 caracteres (um CPF é no formato 000.000.000-00), ao passo que poderíamos usar apenas um número! Sabendo de tudo isso, podemos partir para o projeto do banco de dados.

## Projetando o Banco de Dados

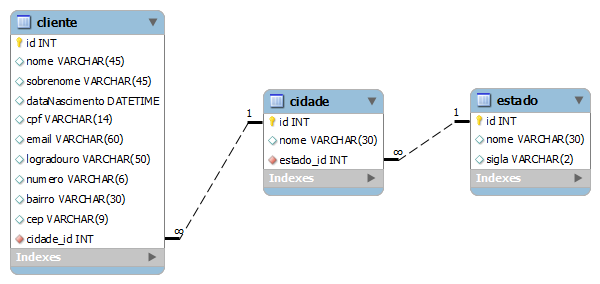
Não irei documentar aqui todo o processo de projeto do banco de dados, que vai desde a análise dos requisitos, passando pela criação do DER (Diagrama Entidade-Relacionamento) até a implementação do modelo físico, pois este não é o objetivo dessa disciplina, mas note que no desenvolvimento de um sistema esses passos são normalmente realizados. Iremos então partir diretamente para a implementação do modelo físico. Antes de criarmos o SQL para a criação da estrutura da nossa base de dados, vamos organizar os atributos das nossas tabelas – que são baseadas nas características dos tipos de entidade – e especificar suas características. Veja a Tabela 5.3.

|  |  |  |  |
| --- | --- | --- | --- |
| Tabela .3: Detalhamento das colunas de cada tabela | | | |
| **Tabela** | **Coluna** | **Tipo** | **É único?** |
| cliente | id\* | INT | sim |
| nome | VARCHAR(45) |  |
| sobrenome | VARCHAR(45) |  |
| dataNascimento | DATETIME |  |
| cpf | VARCHAR(14) | sim |
| email | VARCHAR(60) |  |
| logradouro | VARCHAR(50) |  |
| numero | VARCHAR(6) |  |
| bairro | VARCHAR(30) |  |
| cep | VARCHAR(9) |  |
| cidade\_id\*\* | INT |  |
| cidade | id\* | INT | sim |
| nome | VARCHAR(30) |  |
| estado\_id\*\* | INT |  |
| estado | id\* | INT | sim |
| nome | VARCHAR(30) |  |
| sigla | VARCHAR(2) | Sim |
| \* chave primária | | | |
| \*\* chave estrangeira | | | |

Fonte: do autor

Usei o MySQL Workbench para criar um diagrama do nosso modelo físico. Veja como ficou na Figura 5.1.

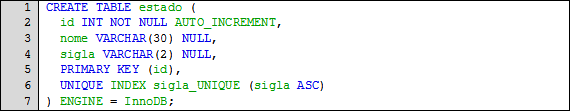
Figura .1: Diagrama do modelo físico da base de dados



Fonte: *Print screen*, MySQL Workbench

Vamos agora implementar o banco de dados. No MySQL Workbench, crie uma nova base de dados com o nome de “cadastro\_clientes” como feito na Seção 4.2. Com a base criada, selecione-a no “*Object Browser*” e copie o código da Listagem 5.1 no editor (*Scratch*) e execute o script.

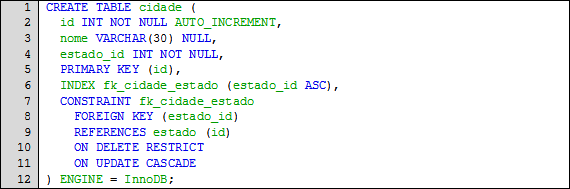
Listagem .1: Script SQL para criação da tabela “estado”



Fonte: do autor

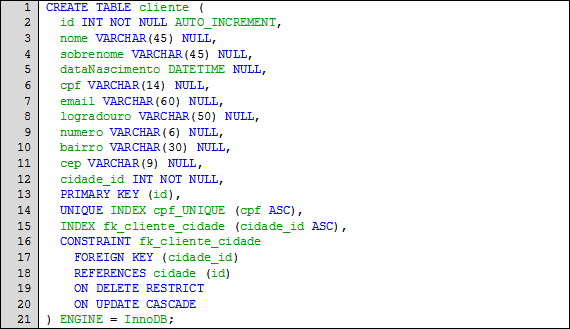
Faça o mesmo processo para a Listagem 5.2 e para a Listagem 5.3.

Listagem .2: Script SQL para criação da tabela “cidade”



Fonte: do autor

Listagem .3: Script SQL para criação da tabela “cliente”



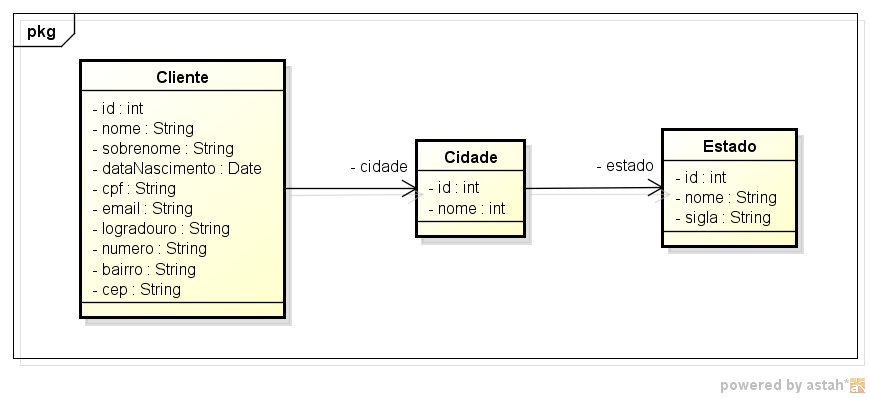
Fonte: do autor

Ao fazer esses três passos, temos nossas três tabelas criadas dentro da base de dados. Clique com o botão direito no “*Object Browser*” e escolhe “*Refresh All*”. Expanda o banco “cadastro\_clientes” e então expanda a pasta “Tables”. Lá dentro estarão as três tabelas criadas. Muito bem, terminamos a implementação do banco. Vamos agora criar um diagrama de classes para representar cada uma das nossas entidades, que serão mapeamentos das nossas tabelas no mundo orientado a objetos.

## Criando o Diagrama de Classes

Para criar nosso diagrama de classes UML, vamos usar a ferramenta astah\* UML (o antigo JUDE UML) que pode ser obtida através do endereço http://astah.change-vision.com/. Usaremos a versão “*Community*” que é gratuita. Como já disse cada tabela do nosso banco de dados vai ter uma representação na nossa aplicação. Essa representação, como você já sabe, é criada usando classes. No diagrama de classes da Figura 5.2, você pode ver as três classes que representam nossas entidades. Note que cada classe contém uma série de atributos privados que representam as colunas da tabela e que o acesso a esses atributos é feito usando os métodos *get* e *set* correspondentes (omitidos no diagrama).

Figura .2: Diagrama de classes das entidades



Fonte: *Print screen*, astah\* UML

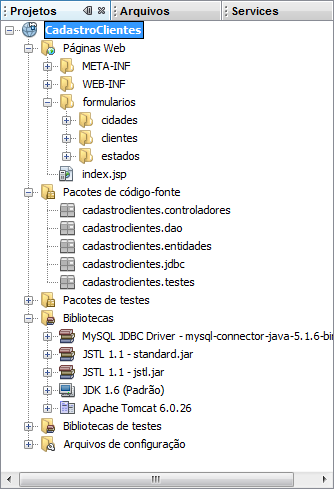
Outro detalhe é que criei apenas o diagrama das classes que são as entidades do sistema, não me preocupando com as outras classes que nosso sistema conterá. Novamente, como no exemplo do nosso banco de dados, em um sistema de verdade, normalmente são desenvolvidos diagramas de classes muito mais completos e complexos, bem como outros tipos de diagramas UML que forem necessários. Com tudo isso pronto, podemos partir para o desenvolvimento do sistema propriamente dito!

## Construindo o Sistema

Nossa primeira tarefa será criar o projeto no NetBeans. Para isso, abra o NetBeans e crie um novo projeto do tipo Java Web com o nome de “CadastroClientes” (sem as aspas). Marque a opção para usar uma pasta dedicada para armazenar as bibliotecas. Com o projeto criado, acesse suas propriedades e desmarque a opção “Implantar ao salvar”. Nas bibliotecas, importe a “JSTL 1.1” e o “MySQL JDBC Driver” e após importar, adicione no projeto.

Em “Pacotes de código-fonte”, crie o pacote “cadastroclientes” (sem as aspas) e os pacotes “controladores”, “dao”, “entidades”, “jdbc” e “testes” dentro do pacote “cadastroclientes”. Em “Páginas Web”, crie os diretórios “formularios” (sem acento) e dentro deste, crie os diretórios “cidades”, “clientes” e “estados”. Veja na Figura 5.3 como deve ficar a estrutura do projeto.

Figura .3: Estrutura do projeto



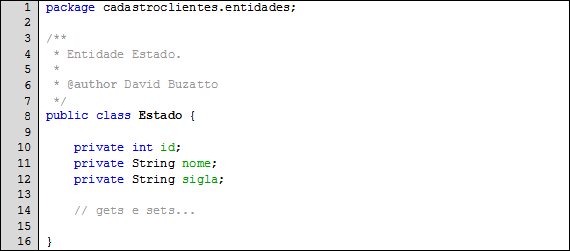
Fonte: *Print screen*, NetBeans IDE 6.9.1

Com a estrutura configurada, vamos agora copiar algumas classes do projeto “PadroesEmPatrica” que criamos na aula anterior. Para isso abra esse projeto, se ainda não estiver aberto, no NetBeans. Expanda o pacote “padroesempratica.jdbc”, clique com o botão direito no arquivo “ConnectionFactory.java” e escolha “Copiar”. Volte ao projeto “CadastroClientes”, clique com o botão direito no pacote “cadastroclientes.jdbc”, escolha “Colar” e então “Refatorar Copiar...”. Um diálogo será aberto. Clique no botão “Refatorar”. O arquivo será copiado para o projeto e as alterações que forem necessárias fazer no arquivo, como mudar a cláusula “*package*”, serão feitas pelo NetBeans. Faça o mesmo processo para o arquivo “DAO.java” contido no pacote “pradoesempratica.dao” do projeto “PadroesEmPratica”, copiando-o no pacote “cadastroclientes.dao” do projeto “CadastroClientes”. O NetBeans vai apontar um erro no arquivo depois da cópia. Para corrigir, abra o arquivo no editor e altere o “*import*” que está com problema.

Outro detalhe é que precisamos mudar a URL da nossa fábrica de conexões para fazer com que as conexões criadas sejam relativas à base de dados “cadastro\_clientes”. Para isso, abra o arquivo “ConnectionFactory.java” do pacote “cadastroclientes.jdbc” e mude a URL de “jdbc:mysql://localhost/testes\_padroes” para “jdbc:mysql://localhost/cadastro\_clientes”.

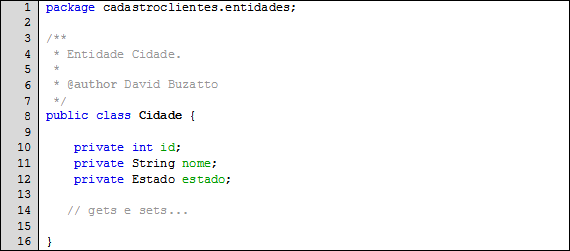
Agora vamos preparar toda a camada de persistência. No pacote “cadastroclientes.entidades”, crie três classes: “Estado”, “Cidade” e “Cliente” (sem as aspas). Nas três listagens a seguir estão listados os códigos-fonte das três classes. Note que estou omitindo os *gets* e os *sets*, mas isso não significa que eles não devam existir. Fica por sua conta criar os *gets* e *sets* ok? Não se esqueça de fazer isso!

Listagem .4: Entidade “Estado”



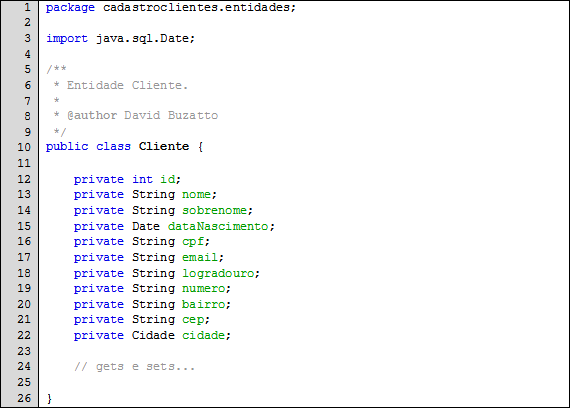
Fonte: do autor

Listagem .5: Entidade “Cidade”



Fonte: do autor

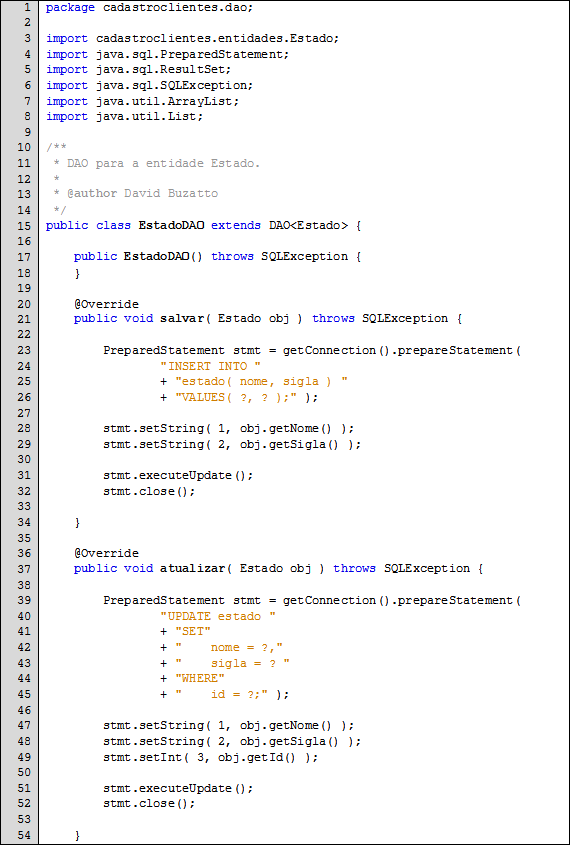
Listagem .6: Entidade “Cliente”



Fonte: do autor

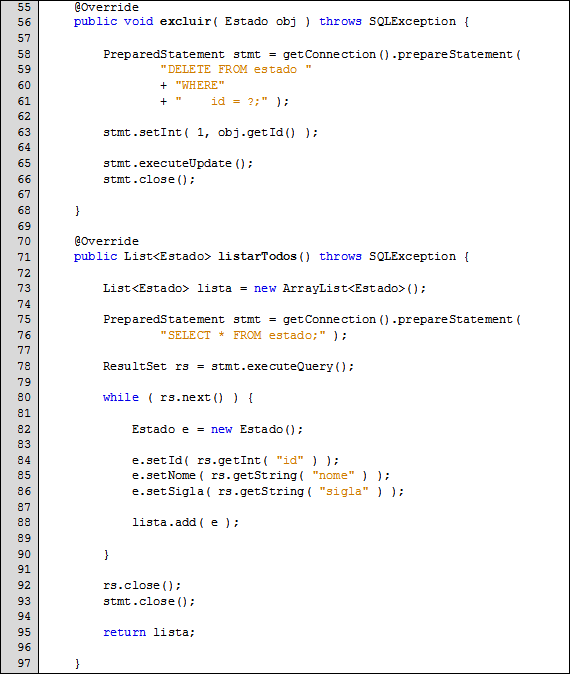
Com as entidades prontas, vamos criar os DAOs. No pacote “cadastroclientes.dao”, crie três classes: “EstadoDAO”, “CidadeDAO” e “ClienteDAO”. O código-fonte de cada uma dessas classes é apresentado nas listagens a seguir. Atenção! As listagens foram divididas.

Listagem .7: Código da classe “EstadoDAO” (parte 1)



Fonte: do autor

Listagem .8: Código da classe “EstadoDAO” (parte 2)

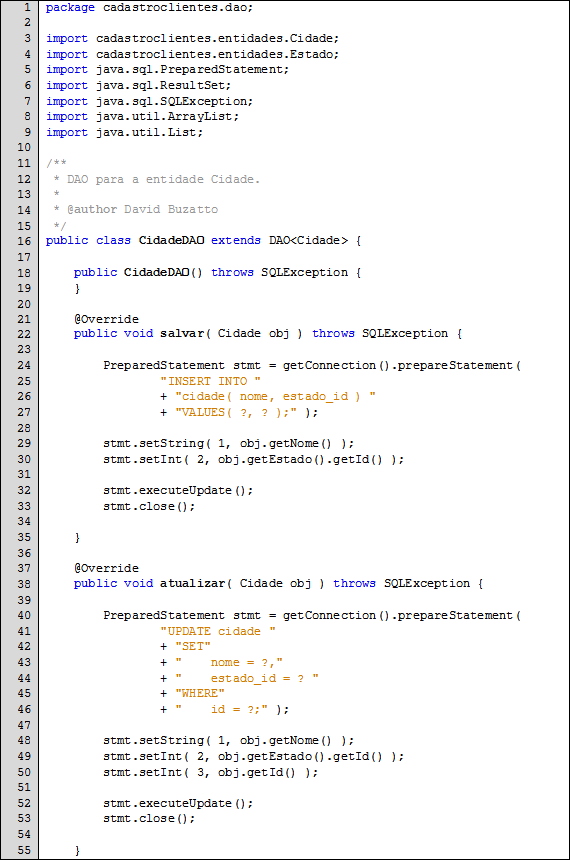
Fonte: do autor

Listagem .9: Código da classe “EstadoDAO” (parte 3)



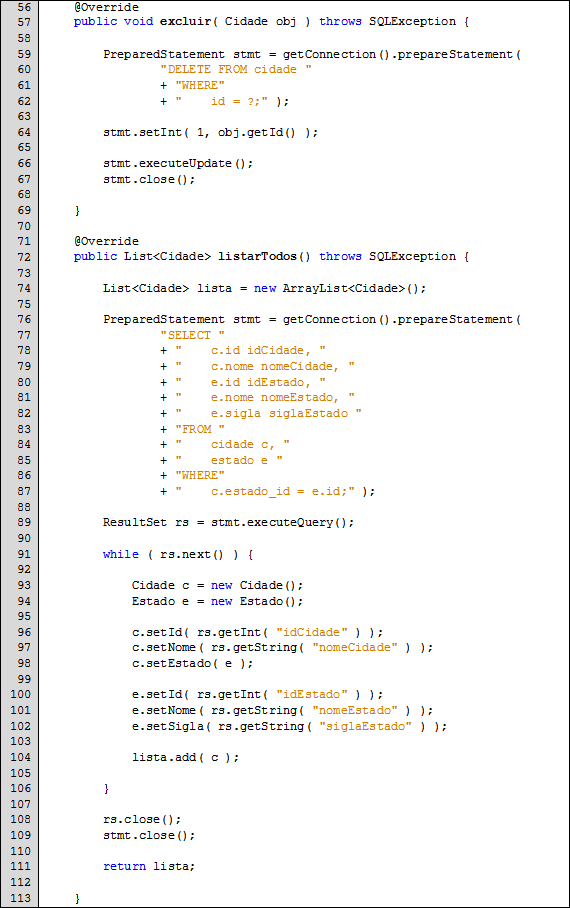
Fonte: do autor

Listagem .10: Código da classe “CidadeDAO” (parte 1)

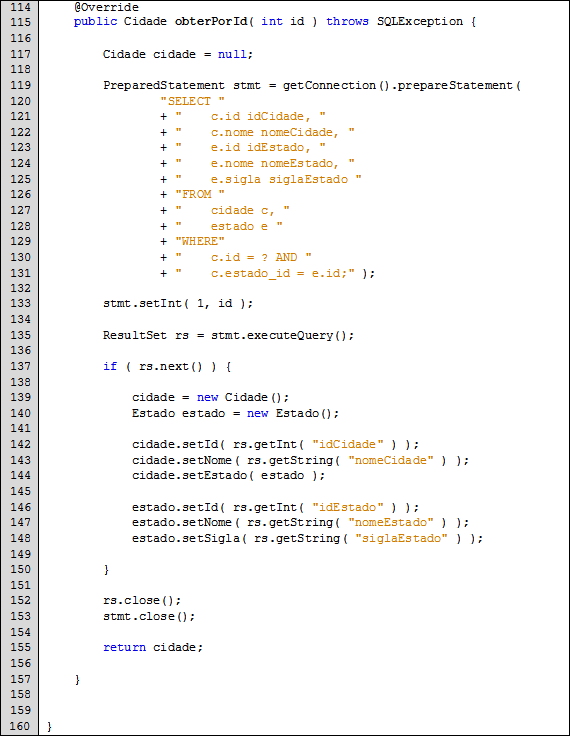


Fonte: do autor

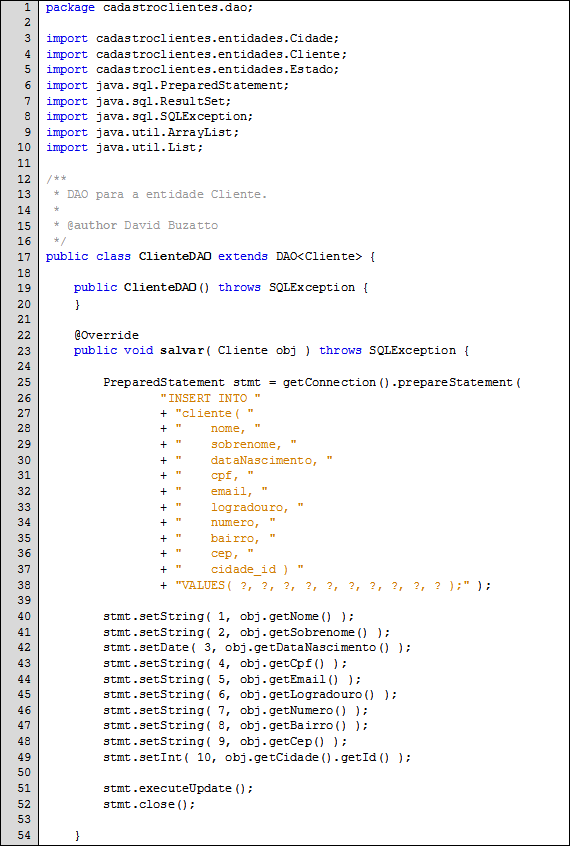
Listagem .11: Código da classe “CidadeDAO” (parte 2)

Fonte: do autor

Listagem .12: Código da classe “CidadeDAO” (parte 3)

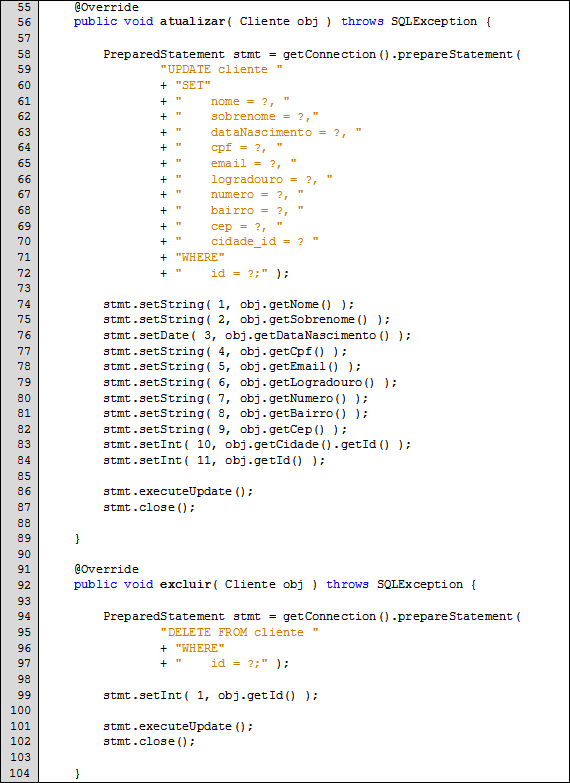
Fonte: do autor

Listagem .13: Código da classe “ClienteDAO” (parte 1)



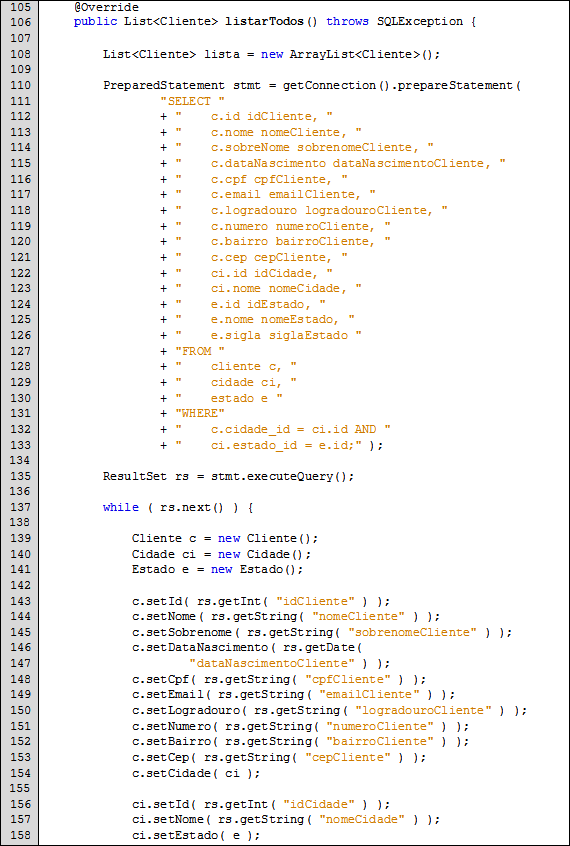
Fonte: do autor

Listagem .14: Código da classe “ClienteDAO” (parte 2)



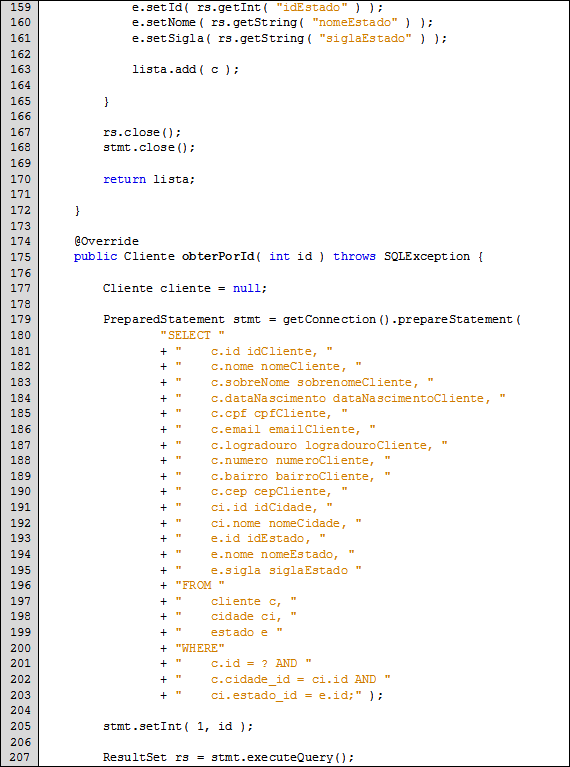
Fonte: do autor

Listagem .15: Código da classe “ClienteDAO” (parte 3)



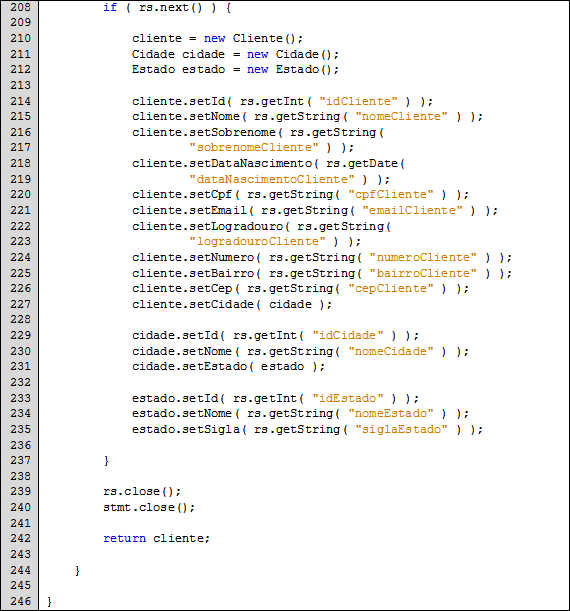
Fonte: do autor

Listagem .16: Código da classe “ClienteDAO” (parte 4)



Fonte: do autor

Listagem .17: Código da classe “ClienteDAO” (parte 5)



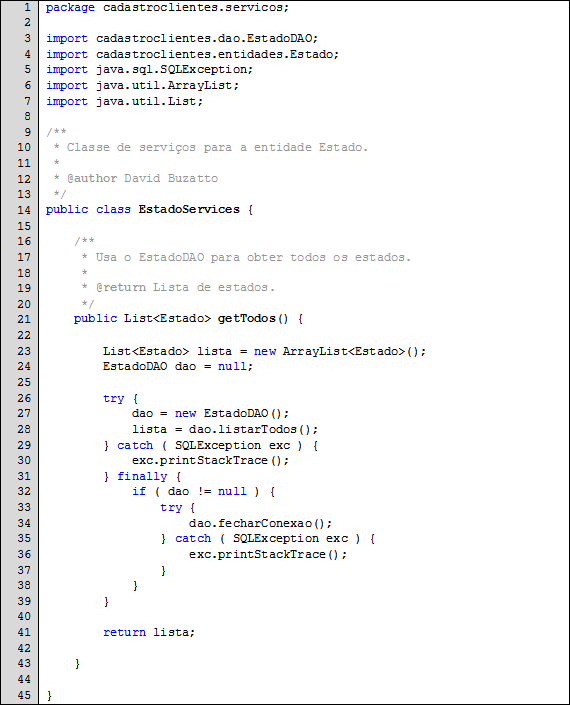
Fonte: do autor

Quantos códigos hein? Copiou tudo? Crie algumas classes de teste no pacote “cadastroclientes.teste” e teste a persistência de cada entidade. Com isso, terminamos a parte da persistência do nosso projeto.

Agora nós vamos começar a implementar as visualizações e os controladores. Nossa aplicação terá três links no index.jsp, sendo que cada link levará a um determinado cadastro. Cada cadastro vai conter uma página principal onde todos os itens desse cadastro serão exibidos e onde poderão ser alterados, excluídos ou então poderemos cadastrar um novo item.

Vamos começar pelo cadastro de estados. Na pasta “estados” dentro da pasta “formularios”, crie um arquivo JSP com o nome de “listagem” (sem as aspas). Nesse arquivo vão ser listados todos os estados, portanto precisamos obter esses estados de alguma forma. Você se lembra que nos nossos DAOs existe um método chamado listarTodos que retorna todos os registros de uma determinada tabela? Nós não vamos usar esse método diretamente no JSP, então vamos criar uma classe de serviços que vai instanciar o DAO, gerar a lista e fechar a conexão para nós. Nos pacotes de código-fonte, crie um novo pacote dentro do “cadastroclientes” chamado “servicos” (sem o “ç”). Nesse pacote, crie uma classe chamada “EstadoServices” (sem as aspas). Essa classe vai ter apenas um método, chamado getTodos, que retornará uma lista de estados. Veja o código-fonte dela na Listagem 5.18.

Listagem .18: Código-fonte da classe EstadoServices

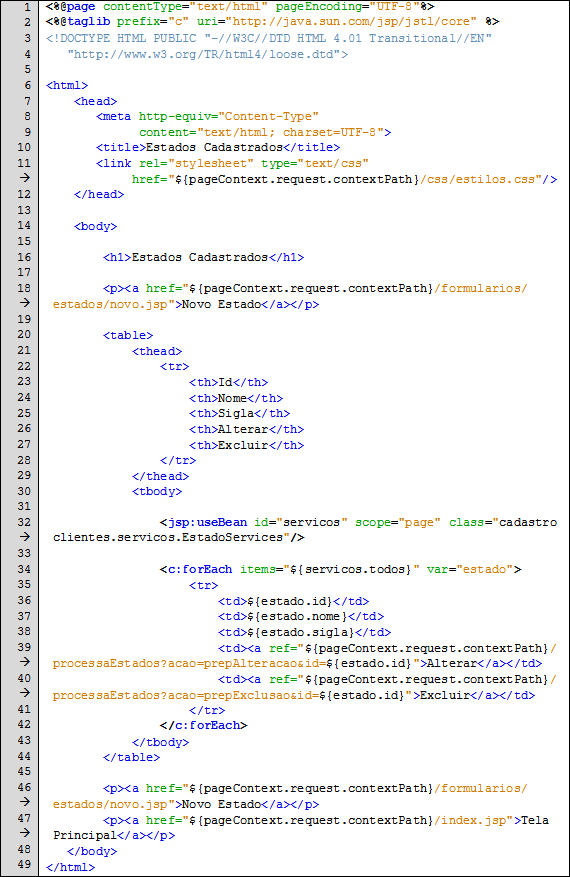


Fonte: do autor

Criamos essa classe para que ela encapsule todo o processo de obtenção da lista de estados. Note que é no método getTodos que o EstadoDAO vai ser instanciado e gerenciado.

Agora que temos a classe que vai obter a lista de estados para nós, além de gerenciar o DAO, nós podemos implementar o nosso arquivo de listagem de estados. Abra o arquivo “/formulários/estados/listagem.jsp” e copie o código da Listagem 5.19.

Listagem .19: Código-fonte do arquivo “/formularios/estados/listagem.jsp”

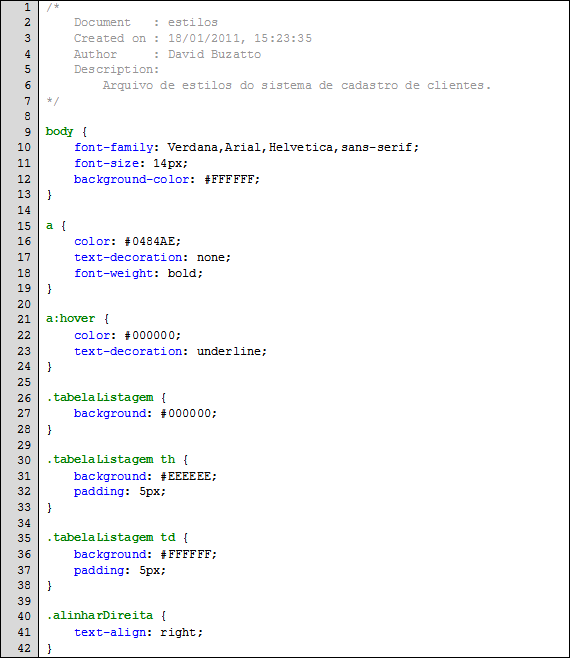


Fonte: do autor

Vamos analisar o código. Na linha 18, usamos a EL “${pageContext.request.contextPath}” que vai gerar o caminho completo dos recursos da aplicação. Nesta linha, criamos um link que aponta para o arquivo “/formularios/estado/novo.jsp” – que ainda não implementamos – e que será o formulário responsável em criar um novo estado. Este mesmo código é repetido na linha 46. Na linha 20, abrimos a *tag* de uma tabela e, até a linha 29, criamos seu cabeçalho. Na linha 32, usamos a *tag* *jsp:useBean* para instanciar um objeto do tipo EstadoServices, que contém o método que vamos usar para obter a lista de estados. Demos o nome de “servicos” para essa instância. Na linha 34, usamos um *c:forEach* para iterar sobre a lista retornada pelo método getTodos da instância “servicos”. Note que a chamada do método getTodos é somente “todos”, pois seguimos o padrão *JavaBeans* nas ELs como você deve se lembrar. Essa chamada é feita usando EL na propriedade “*items*”. Na propriedade “*var*”, damos o nome da variável que vai armazenar a instância atual durante a iteração, no caso, “estado”. Entre as linhas 35 e 41 nós definimos o código que será gerado a cada iteração do *c:forEach*. As três primeiras colunas da tabela são fáceis de entender, entretanto a quarta e a quinta mudam um pouco. Na linha 39, a coluna é formada por um link que aponta para “/processaEstados?acao=prepAlteracao&id=${estado.id}”. Note que estamos codificando na URL duas variáveis. A primeira, chamada “acao”, vai informar para o Servlet que vai estar mapeado para /processaEstados o que queremos fazer, no caso, “prepAlteracao” (preparar alteração). A segunda variável, chamada “id” vai conter o id do estado daquela linha da tabela, ou seja, queremos alterar um estado que tem um determinado id. O mesmo acontece na linha 40, mudando a ação para “prepExclusao” (preparar exclusão). Sei que pode estar um pouco confuso, mas na hora que terminarmos todos os arquivos do cadastro de estados tudo isso vai ficar fácil de entender, não se preocupe.

Note que deixei a linha 11 por último. Nela usamos a *tag* *link* para referenciar um arquivo de folhas de estilos. Nos exemplos das aulas anteriores nós usamos estilos declarados dentro do arquivo JSP usando a *tag* *style*. A partir de agora iremos separar os estilos em arquivos e é por isso que foi usada a *tag* *link* para a pontar para “/css/estilos.css”. Vamos criar esse arquivo? Na pasta “Páginas Web”, crie um diretório chamado “css” (sem as aspas), clique com o botão direito sobre ele e escolha “Novo”. Provavelmente o item que queremos não estará visível, então escolha “Outro...”. Escolha “Web” na categoria e em “Tipo de arquivos” escolha “Folha de estilos em cascata”. Clique em próximo e dê o nome de “estilos” ao arquivo. Copie o código da Listagem 5.20 neste arquivo.

Listagem .20: Arquivo de estilos da aplicação



Fonte: do autor

Execute o projeto e aponte o navegador para a página de listagem para ver como está ficando. Se você já tiver alguns estados cadastrados, vai ver que eles aparecerão na tabela. Vamos agora criar nosso formulário para criar estados. Na pasta “/formularios/estados”, crie um arquivo JSP chamado “novo” (sem as aspas). O código-fonte do arquivo pode ser visto na Listagem 5.21.

Listagem .21: Arquivo “/formularios/estados/novo.jsp”

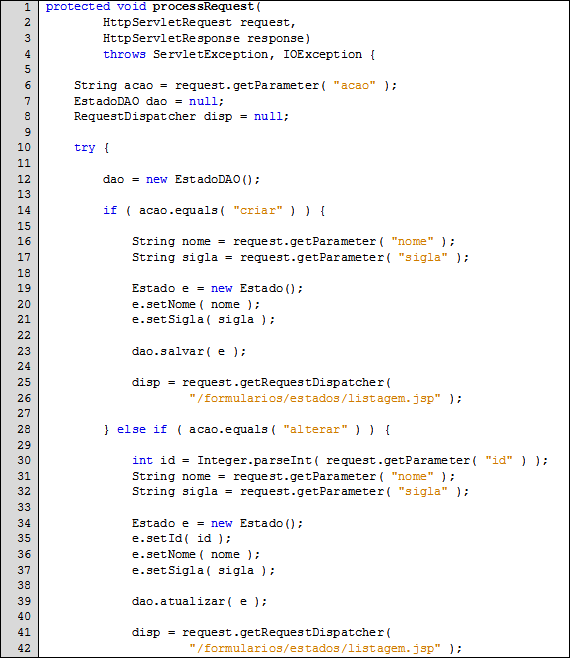


Fonte: do autor

Salve o arquivo e veja se ele está sendo exibido corretamente no navegador. Acesse-o pelo link “Novo Estado” da página de listagem de estados. Vamos analisar o código. Na linha 10 referenciamos o nosso arquivo de estilos. Nas linhas 18 e 19 declaramos a *tag* do formulário. Note que a *action* está apontando para “/processaEstados” que será a URL que mapearemos o Servlet que tratará os estados. A novidade nesse formulário é o uso de um *input* do tipo *hidden* (escondido). Esses tipos de *input* são usados para guardar valores que o usuário do sistema não tem acesso diretamente. No nosso caso, esse *input*, que tem o nome de “acao” e valor “criar”, vai indicar ao Servlet que queremos criar um novo estado. Nós precisamos tratar isso na implementação do nosso Servlet ok? Como já temos nossa página de listagem e o nosso formulário de criação de estados, está na hora de criarmos o Servlet que vai gerenciar isso. No pacote “cadastroclientes.controladores”, crie um Servlet com o nome de “EstadosServlet” (sem as aspas) e configure o mapeamento dele para “/processaEstados” (sem as aspas).

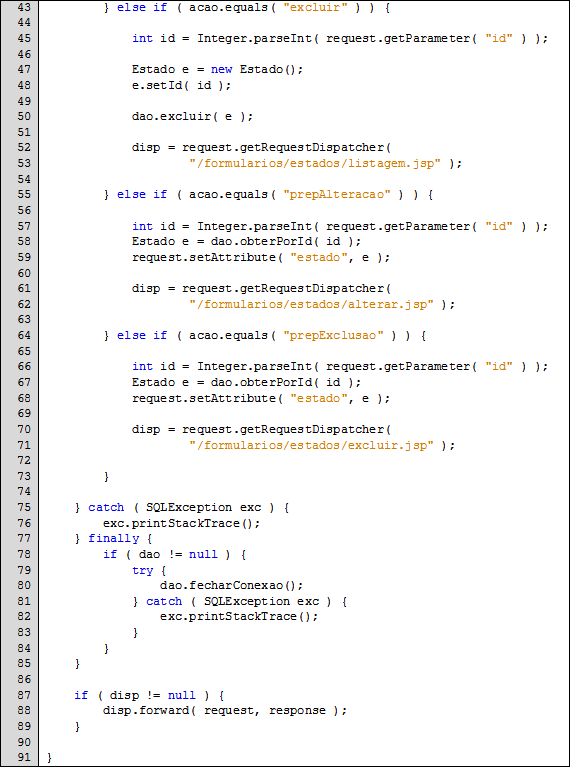
A seguir, na Listagem 5.22 e na Listagem 5.23, é apresentado o código do método processRequest do Servlet EstadosServlet.

Listagem .22: Código-fonte do método processRequest do Servlet “EstadosServlet” (parte 1)



Fonte: do autor

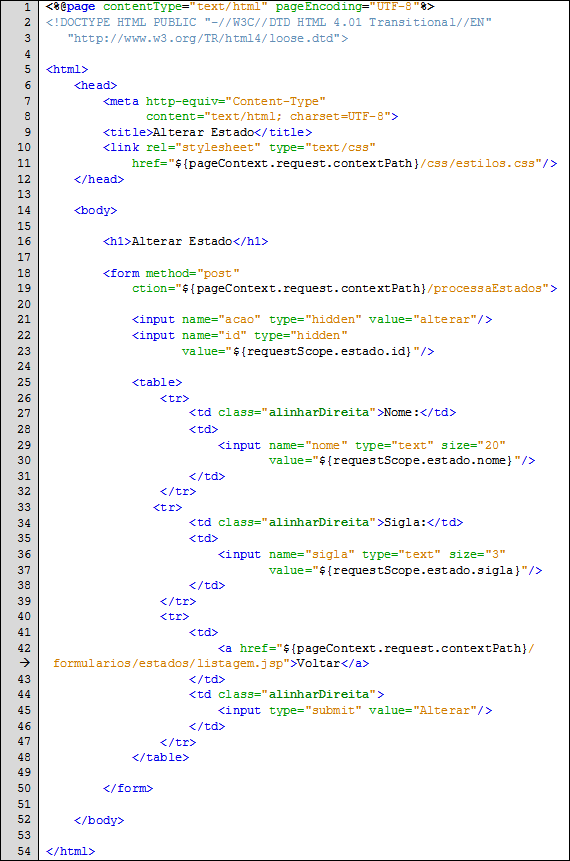
Listagem .23: Código-fonte do método processRequest do Servlet “EstadosServlet” (parte 2)



Fonte: do autor

Copie todo o código e execute o projeto. Acesse a listagem de estados e clique para criar um novo estado. Preencha o formulário e salve. O novo estado será salvo e a listagem aparecerá novamente. Como exercício, tente entender o que está acontecendo no Servlet. Todo o código apresentado já foi estudado nos exemplos anteriores. Perceba que foram tratadas todas as ações possíveis, mas ainda faltam dois arquivos JSP para implementarmos. O “alterar.jsp” e o “excluir.jsp”. Vamos fazer isso? Crie um arquivo na pasta “/formularios/estados” com o nome de “alterar” (sem as aspas) e copie o código da Listagem 5.24.

Listagem .24: Código do arquivo “/formularios/estados/alterar.jsp”



Fonte: do autor

Note que não numerei as linhas dessa listagem, visto que as diferenças importantes em relação ao arquivo novo.jsp são poucas. O que foi alterado é o *input hidden* nomeado como “acao”, que agora tem o valor “alterar”. Foi criado outro *input hidden* para armazenar o id do estado que será alterado. Note que os valores dos campos são preenchidos usando o atributo “estado” que foi configurado no request dentro do Servlet, na seção do código que trata a ação “prepAlteracao”.

O arquivo excluir.jsp é bem parecido, só que neste arquivo não precisamos ter campos de entrada para nome e sigla, pois não vamos alterar esses dados. O importante é o id, que novamente vai ser configurado em um campo escondido. Veja na Listagem 5.25 o código do arquivo “/formularios/estados/excluir.jsp” que você deve criar.

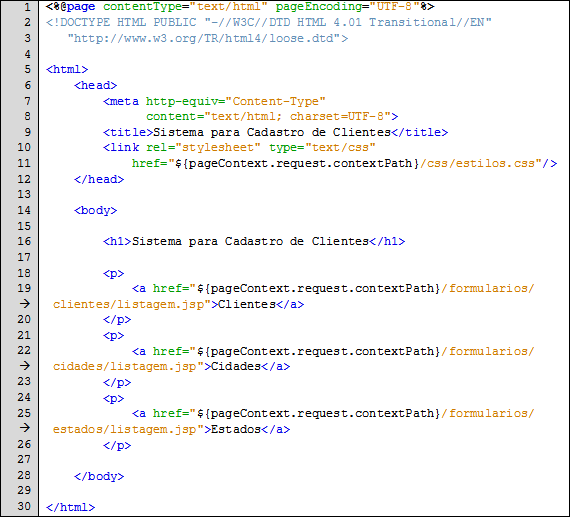
Listagem .25: Código do arquivo “/formularios/estados/excluir.jsp”



Fonte: do autor

Copiou tudo? Teste! Verifique se tudo está funcionando corretamente. Antes de partirmos para os outros cadastros, vamos alterar o index.jsp criando três links, um para cada listagem dos cadastros. O novo código do index.jsp pode ser visto na Listagem 5.26.

Listagem .26: Código-fonte do index.jsp

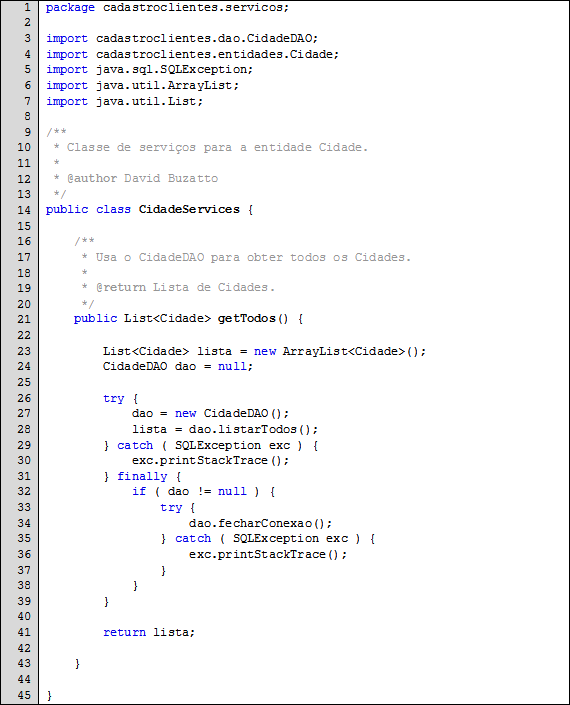


Fonte: do autor

Teste o index.jsp e veja que agora ele tem três links para cada um dos cadastros. O que falta agora para terminarmos nosso projeto é implementar todos os JSPs dos outros cadastros, bem como seus respectivos Servlets e classes de serviço. A seguir, vou listar cada um desses arquivos, agrupando-os por tipo de cadastro, sendo que só irei comentar as linhas que contenham código que ainda não utilizamos ou que precisem de alguma explicação. Não se esqueça de testar o projeto sempre que tiver implementado os arquivos necessários para o funcionamento de uma determinada funcionalidade. Outro detalhe é que algumas linhas de código de algumas listagens foram divididas em duas linhas (nas listagens anteriores eu usei uma flecha para indicar isso), então preste atenção nas linhas que iniciam logo na margem esquerda, pois elas fazem parte da linha anterior. Antes das listagens vou indicar qual arquivo que estamos implementando. Lembre-se de criá-lo antes! Vamos começar?

* Arquivo “cadastroclientes.servicos.CidadeServices.java” (Listagem 5.27).

Listagem .27: Código-fonte da classe CidadeServices



Fonte: do autor

* Arquivo “/formularios/cidades/listagem.jsp” (Listagem 5.28 e Listagem 5.29).

Listagem .28: Código-fonte do arquivo “/formularios/cidades/listagem.jsp” (parte 1)



Fonte: do autor

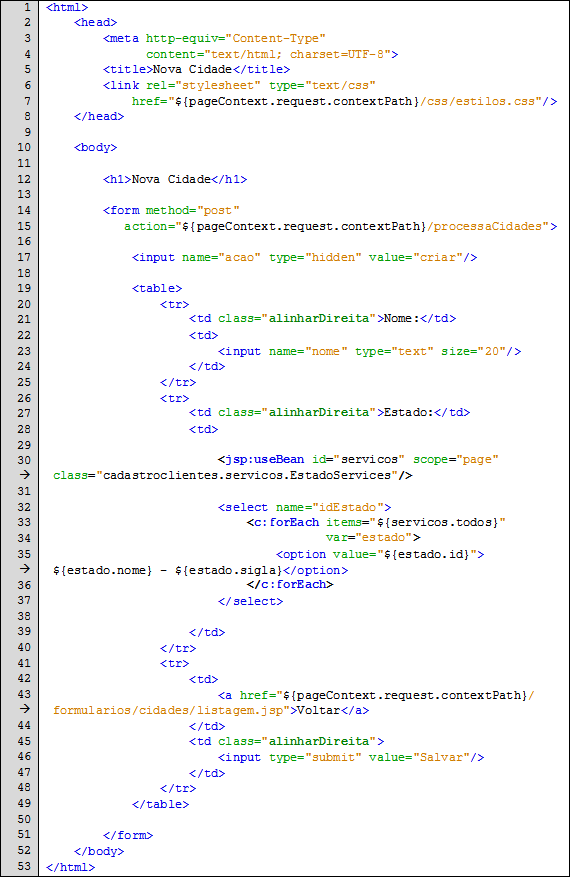
Listagem .29: Código-fonte do arquivo “/formularios/cidades/listagem.jsp” (parte 2)



Fonte: do autor

* Arquivo “/formularios/cidades/novo.jsp” (Listagem 5.30).

Listagem .30: Arquivo “/formularios/cidades/novo.jsp”

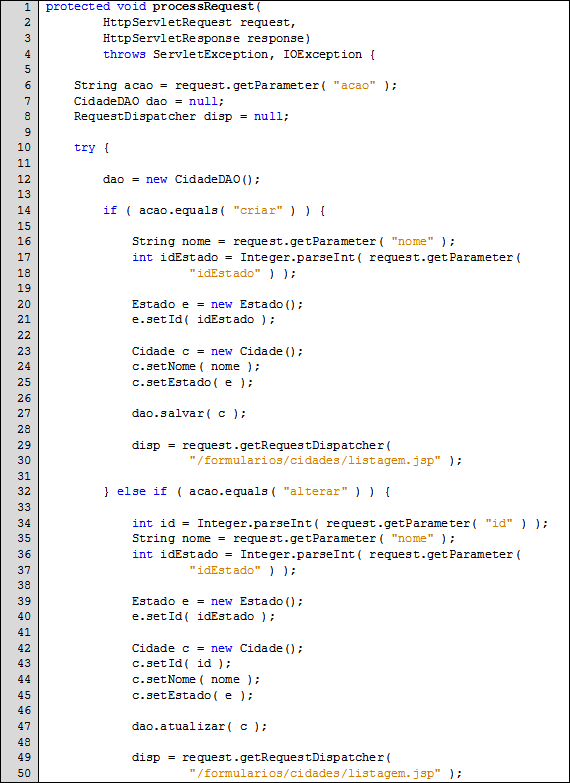


Fonte: do autor

Note que na linha 33 da Listagem 5.30 nós usamos o serviço getTodos() da classe EstadoServices para obter todos os estados cadastrados e com isso gerar as opções (*tag* *option*) do *select* (*combo box*). Note que o valor de cada *option* é o id associado a determinado estado, enquanto o que aparece ao usuário é a junção entre o nome e a sigla do mesmo estado. O id do estado selecionado será enviado ao Servlet por meio da variável “idEstado”, configurada no atributo “*name*” da *tag* *select*.

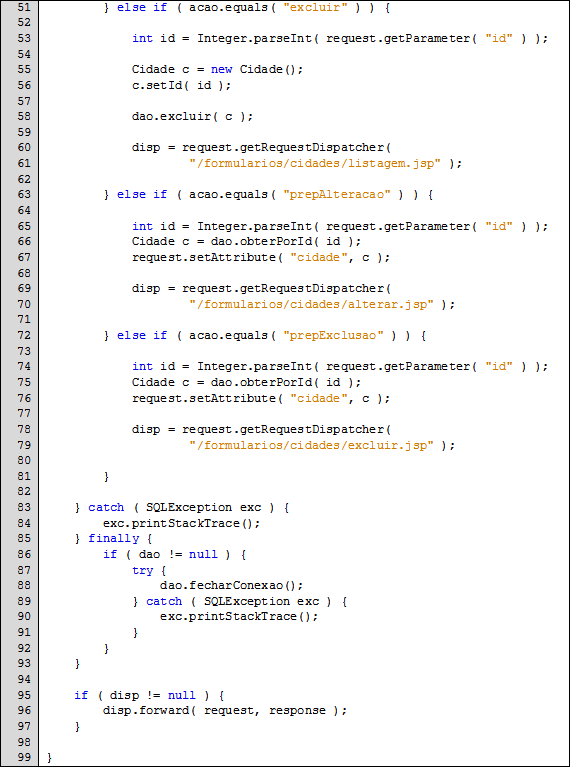
* Método processRequest do arquivo “CidadesServlet.java” contido no pacote “cadastroclientes.controladores”. O padrão de URL do mapeamento desse Servlet deve ser configurado para “/processaCidades” (sem as aspas) (Listagem 5.31 e Listagem 5.32).

Listagem .31: Código-fonte do método processRequest do Servlet “CidadesServlet” (parte 1)



Fonte: do autor

Listagem .32: Código-fonte do método processRequest do Servlet “CidadesServlet” (parte 2)



Fonte: do autor

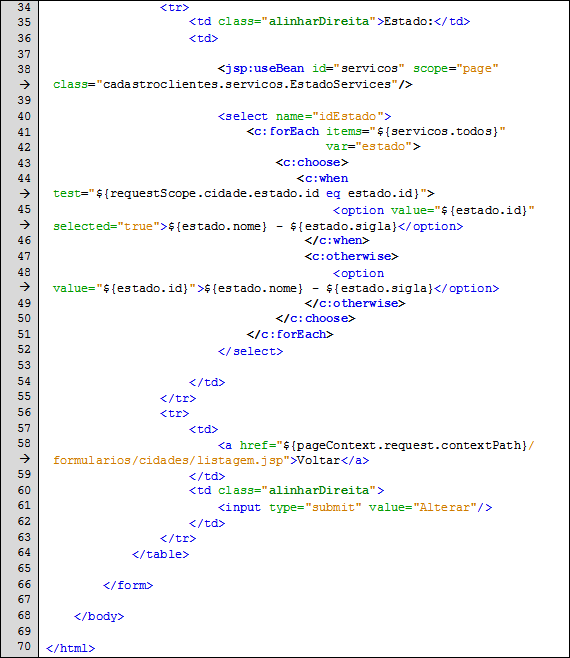
* Arquivo “/formularios/cidades/alterar.jsp” (Listagem 5.33 e Listagem 5.34).

Listagem .33: Código do arquivo “/formularios/cidades/alterar.jsp” (parte 1)



Fonte: do autor

Listagem .34: Código do arquivo “/formularios/cidades/alterar.jsp” (parte 2)



Fonte: do autor

Na Listagem 5.34 temos algo muito interessante. Note que construímos nosso *select* da mesma forma que fizemos na Listagem 5.30, entretanto precisamos saber qual é o estado que é usado na cidade que será alterada. Para isso, ao usarmos o *c:forEach*, nós comparamos o id de cada estado do cadastro com o id do estado associado à cidade que vai ser alterada. Caso os ids sejam iguais, isso quer dizer que é o estado usado na cidade, sendo assim, a *tag* *option* é gerada com um atributo a mais, o “*selected*”, configurado como “*true*”. Usando esse atributo, nós dizemos ao navegador que aquele determinado item deve ser selecionado por padrão.

* Arquivo “/formularios/cidades/excluir.jsp” (Listagem 5.35).

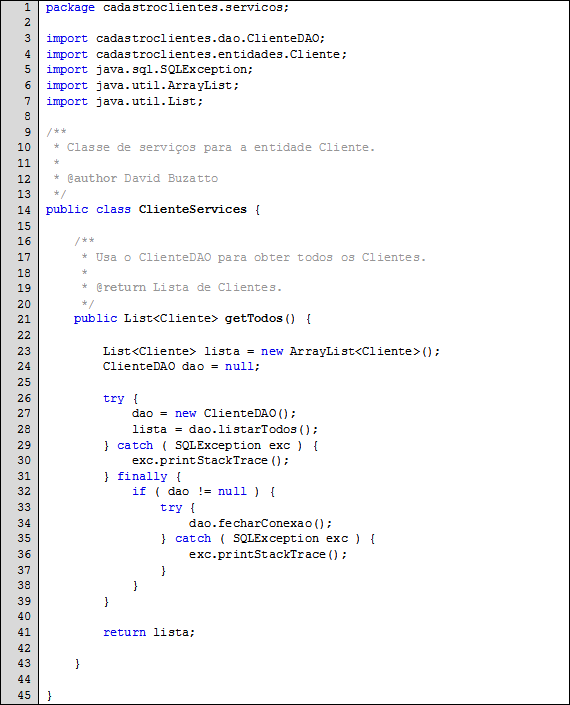
Listagem .35: Código do arquivo “/formularios/cidades/excluir.jsp”



Fonte: do autor

* Arquivo “cadastroclientes.servicos.ClienteServices.java” (Listagem 5.36).

Listagem .36: Código-fonte da classe ClienteServices



Fonte: do autor

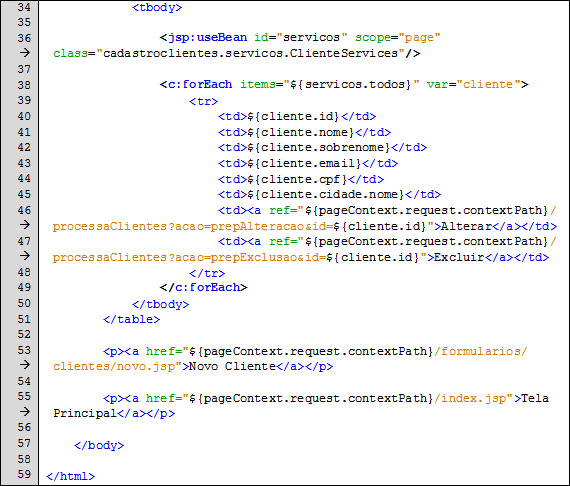
* Arquivo “/formularios/clientes/listagem.jsp” (Listagem 5.37 e Listagem 5.38).

Listagem .37: Código-fonte do arquivo “/formularios/clientes/listagem.jsp” (parte 1)



Fonte: do autor

Listagem .38: Código-fonte do arquivo “/formularios/clientes/listagem.jsp” (parte 2)



Fonte: do autor

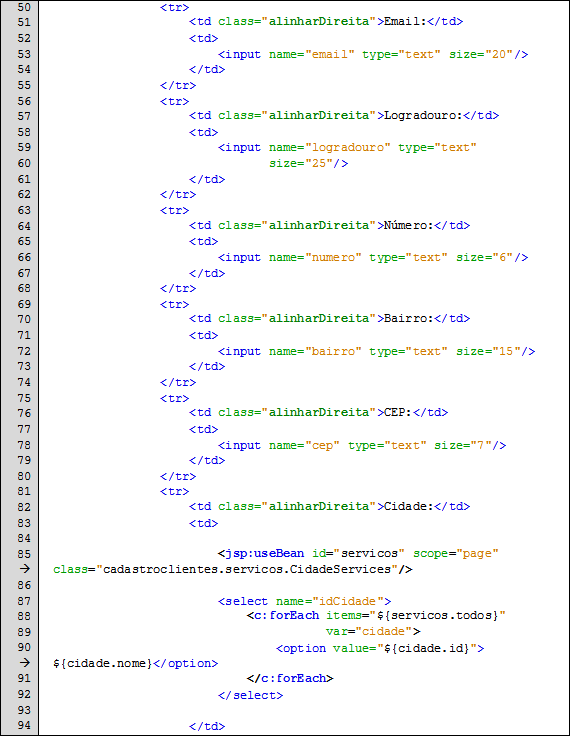
* Arquivo “/formularios/clientes/novo.jsp” (Listagem 5.39, Listagem 5.40 e Listagem 5.41).

Listagem .39: Arquivo “/formularios/clientes/novo.jsp” (parte 1)



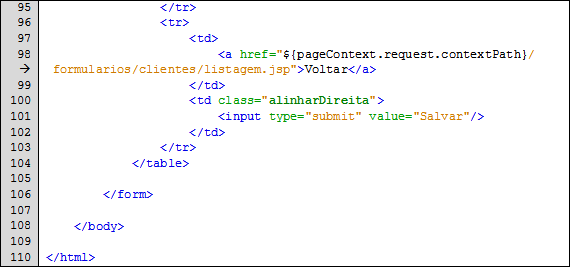
Fonte: do autor

Listagem .40: Arquivo “/formularios/clientes/novo.jsp” (parte 2)



Fonte: do autor

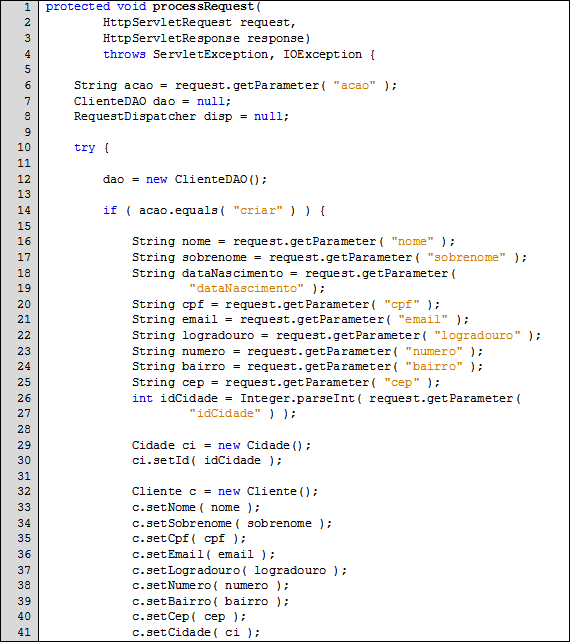
Listagem .41: Arquivo “/formularios/clientes/novo.jsp” (parte 3)



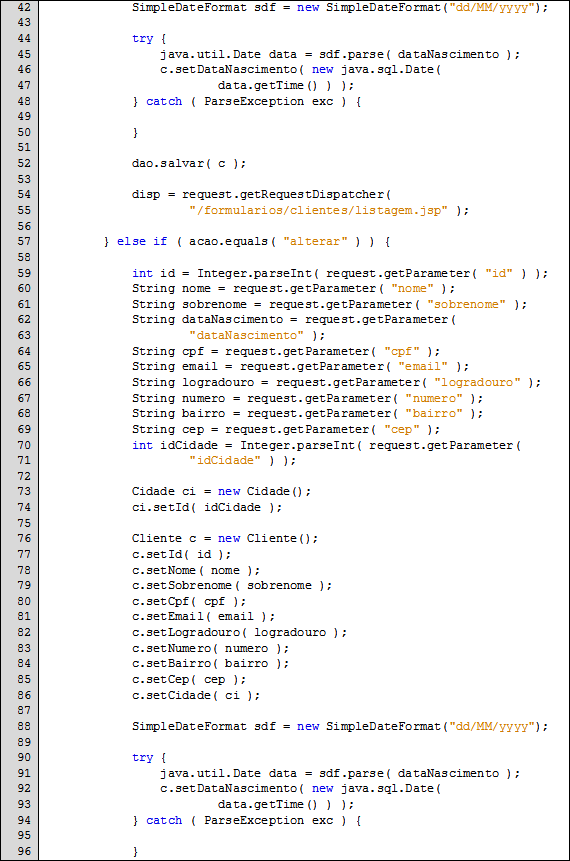
Fonte: do autor

* Método processRequest do arquivo “ClientesServlet.java” contido no pacote “cadastroclientes.controladores”. O padrão de URL do mapeamento desse Servlet deve ser configurado para “/processaClientes” (sem as aspas) (Listagem 5.42, Listagem 5.43, Listagem 5.44 e Listagem 5.45).

Listagem .42: Código-fonte do método processRequest do Servlet “ClientesServlet” (parte 1)

Fonte: do autor

Listagem .43: Código-fonte do método processRequest do Servlet “ClientesServlet” (parte 2)



Fonte: do autor

Note que no início da Listagem 5.43 (linha 42), nós usamos a classe SimpleDateFormat para converter a data inserida pelo usuário no formulário para um objeto do tipo java.util.Date. Para fazer essa conversão, precisamos criar o SimpleDateFormat com o formato que estamos usando para a data, no caso “dia dia / mês mês / ano ano ano ano” (dia com dois dígitos, mês com dois dígitos e ano com quatro dígitos), que deve ser codificado como dd/MM/yyyy. Para fazer a conversão, usamos o método “*parse*” de SimpleDateFormat, sendo que a String com a data que é passada para o método “*parse*” deve estar no formato especificado no construtor do SimpleDateFormat. Caso a String não esteja no formato correto, é lançada uma exceção do tipo ParseException que precisamos tratar.

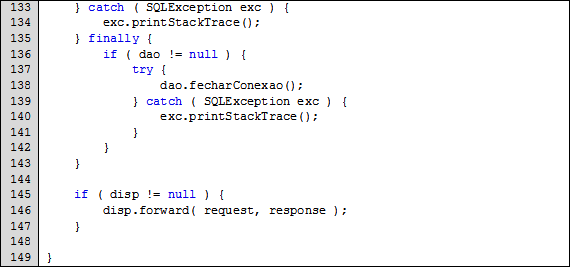
Outro detalhe é que nossa classe Cliente não usa o tipo java.util.Date para representar datas, mas sim o tipo java.sql.Date. Sendo assim, após gerarmos a data com o método “*parse*”, precisamos ainda criar um objeto do tipo java.sql.Date, que recebe como parâmetro o tempo em milissegundos (método getTime() de java.util.Date) da data desejada, ou seja, da data que foi criada a partir da String passada pelo usuário através do formulário.

Listagem .44: Código-fonte do método processRequest do Servlet “ClientesServlet” (parte 3)



Fonte: do autor

Listagem .45: Código-fonte do método processRequest do Servlet “ClientesServlet” (parte 4)



Fonte: do autor

* Arquivo “/formularios/clientes/alterar.jsp” (Listagem 5.46, Listagem 5.47 e Listagem 5.48).

Listagem .46: Código do arquivo “/formularios/clientes/alterar.jsp” (parte 1)



Fonte: do autor

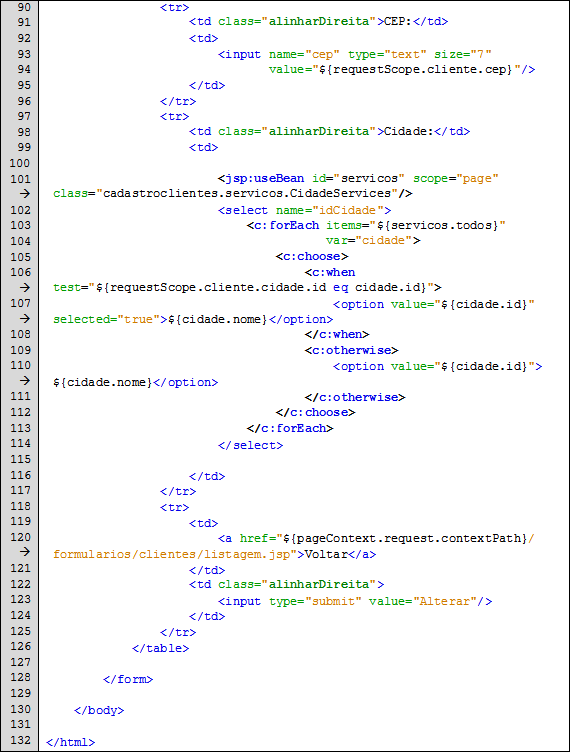
Listagem .47: Código do arquivo “/formularios/clientes/alterar.jsp” (parte 2)



Fonte: do autor

Para que possamos apresentar a data de nascimento (tipo java.sql.Date) do cliente que está passando pelo processo de edição em um formato específico, nós usamos a *tag* *fmt:formatDate* (linha 46 da Listagem 5.47), que faz parte da JSTL. Note que a TagLib que contém essa *tag* é declarada no início da Listagem 5.46 usando o prefixo “fmt”. Na *tag* *fmt:formatDate*, usada na Listagem 5.47, usamos o atributo “*pattern*” (padrão) para configurarmos o formato da String que deve ser gerada a partir da data de nascimento do cliente. O atributo “*value*” é usado para indicar o objeto da data que faz parte do objeto “cliente” contido no “*requestScope*”, ou seja, o objeto que queremos formatar. O atributo “*var*” é usado para indicar o nome da variável usada para armazenar o resultado da formatação, sendo que no caso, configuramos o nome da variável como “data”. Por fim, o atributo “*scope*” é usado para definir o escopo que essa variável vai existir, no caso, cofiguramos para “*page*”, ou seja, a variável só vai existir dentro dessa página. A seguir, como valor do *input* da data de nascimento, usamos a variável “data”, obtida usando EL na forma ${data}.

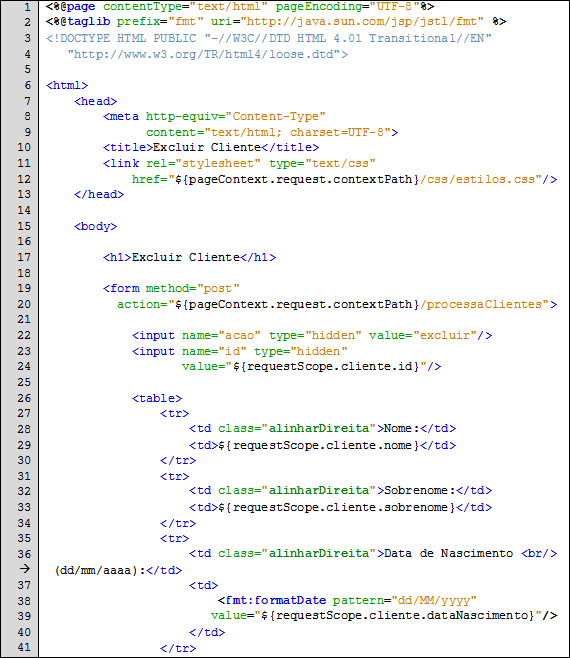
Listagem .48: Código do arquivo “/formularios/clientes/alterar.jsp” (parte 3)



Fonte: do autor

* Arquivo “/formularios/clientes/excluir.jsp” (Listagem 5.49 e Listagem 5.50).

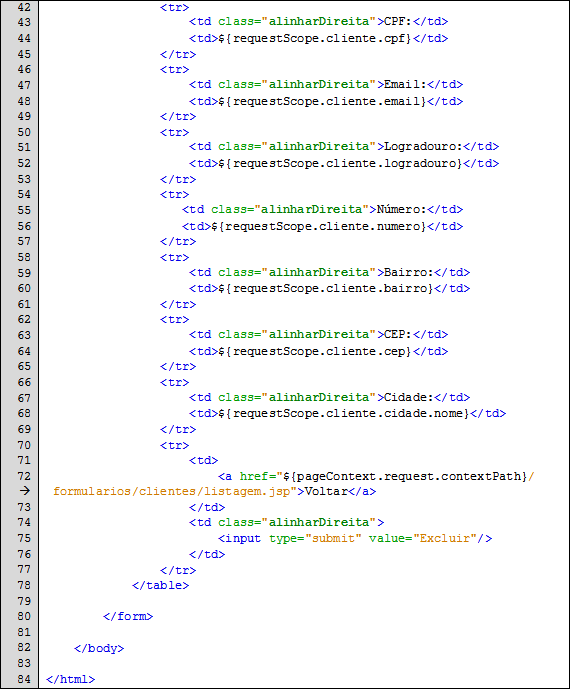
Listagem .49: Código do arquivo “/formularios/clientes/excluir.jsp” (parte 1)



Fonte: do autor

Veja que no final da Listagem 5.49 (linha 38) usamos novamente um formatador de datas (*tag* *fmt:formatDate*) para apresentar a data de nascimento do cliente. Note que desta vez o uso do formatador foi simplificado, visto que agora não precisamos inserir a data formatada dentro de um *input* como fizemos na Listagem 5.47. Quando queremos apenas exibir a data formatada, basta usarmos a *tag* *fmt:formatDate* com os atributos “*pattern*” e “*value*” configurados que a data formatada será gerada onde a *tag* foi usada.

Listagem .50: Código do arquivo “/formularios/clientes/excluir.jsp” (parte 2)



Fonte: do autor

Ufa! Quanta coisa! Copiou todos os arquivos? Testou tudo o que você fez? Que bom! Se tudo funcionou, parabéns! Caso tenha dado algum problema, verifique o que pode ter acontecido, principalmente comparando o código que você copiou com o código das listagens. Agora você é capaz de criar uma aplicação Web em Java que contenha cadastros. Muito legal não é mesmo? Com essa bagagem teórica e prática que tivemos nessa e nas aulas anteriores, nós seremos capazes de trabalhar no projeto que será proposto na próxima aula.

Resumo

Nesta aula construímos uma aplicação Web completa que mantém o cadastro de Clientes, Cidades e Estados. Durante o nosso aprendizado, nós usamos os padrões que aprendemos na Aula 4 para criar a arquitetura da nossa aplicação, dividindo as responsabilidades entre as classes, bem como usando as camadas propostas no padrão MVC, organizando assim o nosso projeto.

Atividades de Aprendizagem

1 – Estude todos os arquivos de código-fonte que foram apresentados nesta aula e, mentalmente, descreva cada uma das linhas desses arquivos.

2 – Na listagem de clientes, insira uma nova coluna na tabela para apresentar a data de nascimento de cada cliente. Use a *tag* *fmt:formatDate* para formatar a data no formato dd/MM/yyyy (dia com dois dígitos, mês com dois dígitos e ano com quatro dígitos). Não se esqueça de usar a diretiva “<%@taglib ...%>” para configurar a TagLib de formatadores da JSTL.

3 – Um detalhe que passou sem ser tratado no nosso projeto é a validação dos campos dos cadastros. Por exemplo, imagine que uma data de nascimento errada foi fornecida pelo usuário do sistema no cadastro de clientes, ou então que foi inserida uma sigla com mais de dois caracteres no cadastro de estados. Se esses dados forem passados de forma errada ao banco de dados, ele gerará um erro e então nosso cadastro não vai funcionar corretamente. Como você resolveria esse problema? Tente criar um mecanismo de validação dos dados fornecidos pelo usuário no cadastro de estados. Caso algum campo não tenha as características necessárias (sigla com mais de dois caracteres, por exemplo), direcione o usuário para uma página de erro, onde deve ser apresentado ao usuário qual erro ocorreu. Nessa página deve existir um botão “Voltar”, que levará o usuário de volta à listagem de estados.

4 – Crie um mecanismo de validação de dados para o cadastro de cidades, da mesma forma que você fez para o cadastro de estados.

5 – Crie um mecanismo de validação de dados para o cadastro de clientes, da mesma forma que você fez para o cadastro de estados.

# Aula 6 – Segundo Projeto – Sistema para Locação de DVDs

**Objetivos**

Aplicar o conhecimento adquirido na construção de uma aplicação Web em Java.

## Introdução

Nesta aula será apresentada uma série de requisitos que devem ser usados para criar uma aplicação Web da mesma forma que fizemos na Aula 5. Tudo que será requisitado estará baseado no que já aprendemos, sendo assim, todas as funcionalidades requeridas poderão ser implementadas com recursos já vistos na Aula 5.

## Apresentação dos Requisitos

Você foi contratado para criar um sistema para controle de cadastro de DVDs. Esse sistema irá manter apenas o cadastro de DVDs e não irá gerenciar a locação dos mesmos. Os dados que deverão ser mantidos para todos os DVDs são: título, ano de lançamento, ator principal, ator coadjuvante, data de lançamento, duração em minutos, gênero e classificação etária. Os dados dos atores, dos gêneros e das classificações etárias deverão ser mantidos através de cadastros específicos. Um ator deve ter um nome, um sobrenome e uma data de estreia (indica quando foi o primeiro filme do ator). Um gênero deve ter uma descrição. Uma classificação etária deve ter também apenas uma descrição. Cada um dos cadastros (DVD, ator, gênero e classificação etária), deve conter as funcionalidades de criar, alterar e excluir um determinado registro. A página principal da aplicação deve conter um link para cada tipo de cadastro.

## Desenvolvimento do Projeto

O projeto Web que deverá ser criado deve ter o nome de “LocacaoDVDs”. Configure o projeto para conter as bibliotecas internamente e para não ser implantado quando algum arquivo for salvo. O pacote de código-fonte base do projeto deve ter o nome de “locacaodvds”. A estrutura do projeto deve ser igual à estrutura do projeto criado na Aula 5, sendo que, obviamente, o nome das classes e suas respectivas implementações serão diferentes do projeto da Aula 5. A base de dados com as tabelas das entidades obtidas a partir da análise dos requisitos na seção anterior deve ter o nome de “locacao\_dvds”. Não se esqueça de que cada entidade deverá conter um identificador. As páginas da aplicação deverão ter sua aparência configurada usando uma folha de estilos, da mesma forma que fizemos no projeto da Aula 5. Tente personalizar os estilos, mudando as cores, etc. Você está livre para usar imagens ou qualquer outro artifício que julgar interessante para mudar a aparência da sua aplicação. Boa sorte!

Resumo

Nesta aula foi requisitado que você implementasse uma aplicação Web em Java para gerenciar o cadastro de DVDs de uma locadora, por isso, não teremos exercícios para serem realizados.

Atividades de Aprendizagem

1 – A partir de tudo o que foi estudado e aprendido durante todas as aulas, faça um sistema para o gerenciamento de um frigorífico. Tente criar um texto que descreva os requisitos do sistema e então execute cada passo do desenvolvimento como sugerido na Aula 6.

2 – A partir de tudo o que foi estudado e aprendido durante todas as aulas, faça um sistema para o gerenciamento de uma escola. Tente criar um texto que descreva os requisitos do sistema e então execute cada passo do desenvolvimento como sugerido na Aula 6.

# Referências

ALEXANDER, CHRISTOPHER; ISHIKAWA, SARA; SILVERSTEIN, MURRAY. **A Pattern Language:** towns, buildings, construction. New York: Oxford University Press, 1977. 1171 p.

BASHAM, BRYAN; SIERRA, KATHY; BATES, BERT. **Use a Cabeça! Servlets & JSP**. 2. ed. Rio de Janeiro: Alta Books, 2008. 912 p.

DEITEL, PAUL J; DEITEL, HARVEY M. **Java Como Programar**. 6. ed. São Paulo: Pearson Prentice Hall, 2005. 1110 p.

ELMASRI, RAMEZ E.; NAVATHE, SHAMKANT. **Sistemas de Banco de Dados**. 4. ed. São Paulo: Addison-Wesley, 2005. 744 p.

GAMMA, ERICH; HELM, RICHARD; JOHNSON, RALPH; VLISSIDES, JOHN. **Padrões de Projeto:** Soluções Reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000. 364 p.

Currículo do Professor-Autor

Meu nome é David Buzatto, sou mestre em Ciência da Computação pela Universidade Federal de São Carlos (UFSCar) e bacharel em Sistemas de Informação pelo Centro Universitário da Fundação de Ensino Octábio Bastos (UniFEOB). Atualmente sou Professor do Ensino Técnico e Tecnológico do Instituto Federal de Educação, Ciência e Tecnologia de São Paulo no Campus de São João da Boa Vista. Atuo nas áreas de Engenharia de Software e Interação Humano-Computador e tenho experiência no desenvolvimento de aplicações tanto Web quanto *desktop* utilizando diversas tecnologias. Tenho interesse em desenvolvimento de software, linguagens de programação, linguagens formais, construção de compiladores e desenvolvimento de jogos eletrônicos. Se quiser saber um pouco mais sobre mim, acesse meu site: <http://www.davidbuzatto.com.br>. O meu currículo lattes pode ser acessado através do link: <http://lattes.cnpq.br/7916716785143122>

Minhas últimas publicações foram:

BUZATTO, D.; ANACLETO, J. C.; DIAS, A. L.; SILVA, M. A. R.; VILLENA, J. M. R.; CARVALHO, A. F. P. de. **Filling out Learning Objects Metadata considering Cultural Contextualization**. In: IEEE International Conference on Systems, Man, and Cybernetics (IEEE SMC 2009), 2009, San Antonio, Texas. Anais IEEE SMC 2009. Red Hook, NY, USA: Curran Associates, 2009. p. 424-429.

DIAS, A. L.; ANACLETO, J. C.; PENTEADO, R. A. D.; SILVEIRA, L. M.; SILVA, M. A. R.; BUZATTO, D.; VILLENA, J. M. R. **Web Collaboration Motivated by Colors Emotionally Based on Common Sense**. In: IEEE International Conference on Systems, Man, and Cybernetics (IEEE SMC 2009), 2009, San Antonio, Texas. Anais IEEE SMC 2009, 2009. Red Hook, NY, USA: Curran Associates, 2009. p. 801-806.

SILVA, M. A. R.; ANACLETO, J. C.; BUZATTO, D. **A Game to Support Childrens' Expression and Socialization Considering their Cultural Background**. In: IEEE International Conference on Systems, Man, and Cybernetics (IEEE SMC 2009), 2009, San Antonio, Texas. Anais IEEE SMC 2009, 2009. Red Hook, NY, USA: Curran Associates, 2009. p. 1230-1235.

BUZATTO, D.; ANACLETO, J. C.; DIAS, A. L. **Providing Culturally Contextualized Metadata to Promote Sharing and Reuse of Learning Objects**. In: ACM International Conference on Design of Communication (SIGDOC 2009), 2009, Bloomington, Indiana. Proceedings of the 27th ACM International Conference on Design of Communication. New York: Association for Computing Machinery, Inc., 2009. p. 163-170.