Criando e Entendendo o Primeiro Servlet

Por: Raphaela Galhardo Fernandes

Resumo

Neste tutorial serão apresentados conceitos relacionados a *Servlets*. Ele inicial como uma breve introdução do funcionamento de aplicações web, partindo para os conceitos de *servlets*, *container* web, Tomcat. Por fim, é apresentado um exemplo de criação de um *servlet*, utilizando o eclipse como IDE e o *container* Tomcat 6.0.

1. Introdução

Atualmente, diversas aplicações são desenvolvidas para serem acessíveis via Internet. Essas aplicações podem ser denominadas de aplicações ou sistemas web e funcionam com base na comunicação entre um cliente e um servidor web.

Para quem não ainda não conhece, o cliente, nestes casos, geralmente, é uma pessoa interagindo com um navegador web (*browser*), por exemplo, clicando em um link de uma página. Dessa forma, a pessoa espera que, após clicar neste link, alguma ação seja executada, por exemplo, visualizar uma outra página.

O servidor pode ser definido como aquele que possui conteúdos que podem ser enviados ao cliente. Esses conteúdos podem ser, por exemplo, páginas web, figuras, arquivos, etc. O servidor reconhece uma "mensagem" (requisição – *request*) enviada pelo cliente através de um *browser*, trata essa requisição e devolve ao cliente uma resposta (*response*), de acordo com o que foi solicitado.

A comunicação entre o cliente e o servidor pode ser resumida pela Figura 1. (1) Inicialmente, uma pessoa interage com o *browser* (cliente). (2) O *browser* formata a requisição do cliente (que pode ter sido um clique em algum link) e a envia ao servidor. (3) O servidor, por sua vez, interpreta a requisição do cliente e monta uma resposta que será enviada de volta ao cliente (*browser*). (4) Essa resposta pode ser uma página contendo código HTML, (5) que será exibida pelo *browser* para a pessoa de onde partiu a requisição para visualizá-la.

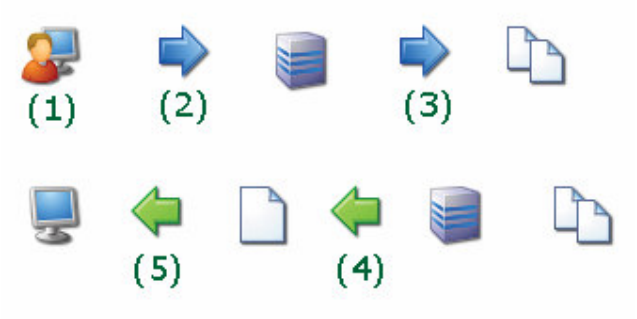

Figura 1 - Comunicação entre cliente e servidor

Essa comunicação entre cliente e servidor web é feita através do uso do protocolo HTTP (*HyperText Transfer Protocol*), que permite a conversação entre requisições enviadas pelo cliente (HTTP *request*) e respostas enviadas pelo servidor (HTTP *response*).

As requisições ao servidor podem ser feitas através URL's informadas ao navegador web e de dois métodos: **get** e **post**, definidos a seguir:

GET –

- A partir dele é possível pedir ao servidor algum recurso: páginas HTML, figuras (JPGE), arquivos PDF, etc.
- Também é possível enviar parâmetros ao servidor, concatenando-os na URL de requisição. Dessa forma, é possível adicionar a URL com os parâmetros aos links de Favoritos do *browser*.
- Como os parâmetros são anexados à URL enviada ao servidor, informações sigilosas (como senhas, por exemplo) ficam expostas a qualquer pessoa.
- Dependendo do servidor, a requisição enviada pode ter um número limitado de caracteres.

POST –

- A partir dele é possível submeter formulários, pedir ao servidor algum recurso: páginas HTML, figuras (JPGE), arquivos PDF, etc.
- Também é possível enviar parâmetros ao servidor, que não são visíveis ao usuário na URL da requisição.
- Os parâmetros são enviados ao servidor no corpo de uma mensagem. Quando o servidor identifica que a requisição veio pelo método **post**, os parâmetros são procurados no corpo da mensagem.

2. Servlets

As aplicações que são servidores web retornam apenas páginas estáticas, ou seja, páginas que não são dinâmicas. As páginas dinâmicas são aquelas montadas somente após o tratamento da requisição pelo servidor, pois dependem de informações que podem variar a cada resposta. Por exemplo, uma página que lista as mensagens de um blog, ou seja, as mensagens podem variar diariamente, dependendo do que o autor do blog submeteu para a aplicação anteriormente.

As páginas estáticas são aquelas que não podem assumir formatos variados após cada requisição, ou seja, sempre possuem o mesmo formato.

A Listagem 1 apresenta um exemplo de página dinâmica. Esta listagem apresenta o código HTML da página dinâmica, onde o conteúdo a ser exibido pelo *browser* depende da data atual, que é fornecida pela máquina em que se encontra o servidor web.

```
<html>
<body>
      Data de Hoje é:
           [Data obtida da máquina onde está hospedado o servidor web]
</body>
</html>
```

Listagem 1 – Exemplo de página dinâmica

Então surge a pergunta, como as páginas dinâmicas são exibidas se não é possível que sejam enviadas como respostas de um servidor web? Na verdade, existe uma aplicação auxiliar que permite a construção de páginas estáticas a partir de páginas dinâmicas, em que dados são preenchidos em tempo de execução.

Quando uma requisição é enviada ao servidor pelo cliente, a aplicação servidora identifica que esta requisição deve ser primeiramente tratada pela aplicação auxiliar, para identificar os valores das informações que variam. Essa aplicação auxiliar identifica esses valores, constrói uma nova página com os dados definidos e a envia à aplicação servidor web. Assim, o servidor web recebe uma página estática no formato HTML, que é enviada como resposta ao cliente (*browser*) e exibida corretamente.

Utilizando a linguagem Java, essa aplicação auxiliar é definida como uma classe que é um *servlet*. Após o recebimento de uma requisição por um *servlet*, ele pode capturar parâmetros enviados na requisição, fazer algum processamento e devolver ao cliente, por exemplo, uma página HTML.

Um *servlet*, por exemplo, pode receber dados em um formulário HTML por meio de uma requisição HTTP, processar estes dados, atualizar uma determinada base de

dados da aplicação, e gerar alguma resposta dinamicamente para o cliente que fez a requisição.

Para que uma classe Java seja um *servlet*, ela deve herdar da classe **javax.servlet.HttpServlet**, disponível na API Java. De acordo com a requisição enviada ao servidor pelo *browser*, um dos métodos **doGet** ou **doPost** é chamado, automaticamente, da classe que o programador definiu como sendo a aplicação auxiliar (classe *servlet*). Dessa forma, a classe *servlet* criada deve ter a implementação de um desses dois métodos para o tratamento da requisição do cliente. Esses métodos devem ter as assinaturas presentes na Listagem 2.

```
Assinatura do doGet:

public void doGet(HttpServletRequest request,
                  HttpServletResponse response);

Assinatura do doPost:

public void doPost(HttpServletRequest request,
                  HttpServletResponse response);
```

Listagem 2 – Assinaturas dos métodos doGet e doPost de um Servlet

A Listagem 3 apresenta um exemplo de uma classe que é um *servlet*, em que a resposta a ser enviada ao cliente será uma página HTML. Observa-se que a mesma herda da classe **javax.servlet.HttpServlet**. Neste exemplo, é implementado o método **doGet**, que recebe como argumento objetos **HttpServletRequest request** e **HttpServletResponse response**, que representam a requisição do cliente e a resposta ao cliente, respectivamente. Esses dois objetos são criados pelo *container* web e passados à classe *servlet*. O mesmo acontece com a implementação do método **doPost**. O *container* web será descrito mais adiante.

Dentro da implementação do método **doGet**, um objeto **PrintWriter out** é criado. Esse objeto é obtido a partir do objeto **response** e a partir dele é possível criar o conteúdo HTML a ser enviado como resposta ao cliente.

```
package br.com.jeebrasil.servlet;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloWorldServlet extends HttpServlet{

   //Os métodos doGet e doPost recebem como argumento objetos
   //HttpServletRequest request e HttpServletResponse response.
   //Esses objetos são criados pelo container web (explicado adiante)
   protected void doGet( HttpServletRequest request,
                         HttpServletResponse response)
                              throws ServletException, IOException {

      //Objeto Date para exibição da data atual
      Date hoje = new Date();

      //Objeto criado para enviar uma resposta ao cliente
      //no formato de código HTML
      PrintWriter out = response.getWriter();

      //Montando HTML a ser exibido a partir do objeto out
      out.println("<HTML>");
      out.println("<BODY>");
      out.println("<h1 style = \"text-align: center; \">");
      out.println("HELLO WORLD!!! <BR/>");
      out.println("DATA: " + hoje);
      out.println("</h1>");
      out.println("</BODY>");
      out.println("</HTML>");
   }
}
```

Listagem 3 - Exemplo de uma classe Servlet

3. Container Web

Quando uma requisição web é enviada ao servidor, se ela for para exibição de conteúdos estáticos (páginas HTML estáticas, por exemplo), este conteúdo é devolvido como resposta ao navegador web diretamente. No caso, de uma requisição que solicita conteúdos dinâmicos, ela deve ser tratada por uma classe *servlet*. Porém, essa requisição não é manuseada diretamente pelo *servlet*. Quem encaminha uma requisição ao *servlet* para que ela seja tratada, é um componente denominado *container* Web, responsável por invocar o método **doGet** ou **doPost** do *servlet* de acordo com o que foi requisitado. Todo esse processo pode ser resumido pela Figura 2.

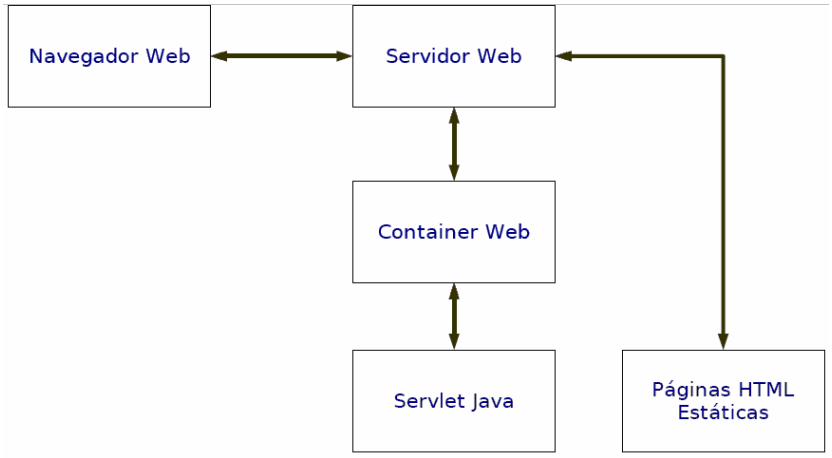

Figura 2 - Tratamento de Conteúdos Dinâmicos e Estáticos

O *container* permite a comunicação entre os *servlets* construídos pelo programador e o servidor web. Para cada *servlet* que recebe uma requisição, automaticamente o *container* cria uma nova *thread* Java e faz o gerenciamento de todas as *threads* criadas.

Um *servlet* possui um ciclo de vida que é gerenciado pelo *container*. Este, por sua vez, controla a vida e morte de todos os *servlets*: carregando as classes, instanciando e inicializando os *servlets*, invocando os métodos dos *servlets* e permitindo que os *servlets* sejam destruídos pelo coletor de lixo quando não mais utilizados.

O manuseamento de uma requisição pelo *container*[1] pode ser ilustrado pela Figura 3 .

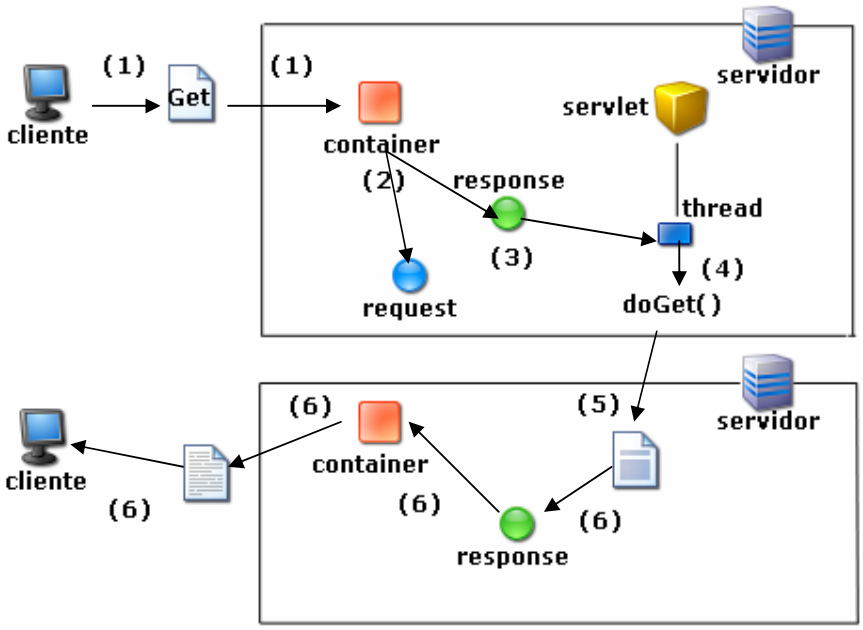

Figura 3 - Manuseamento da requisição pelo container web

1. Inicialmente, uma requisição HTTP é enviada ao servidor pelo cliente, por exemplo, a partir de um método **get**;
2. O *container* identifica que a requisição deve ser tratada por um *servlet* e cria dois objetos, um do tipo **javax.servlet.HttpServletRequest request** e outro do tipo **javax.servlet.HttpServletResponse response**;
3. O *container* identifica qual o *servlet* deve tratar a requisição, de acordo com a URL informada. Se esta for a primeira requisição ao *servlet*, a classe que o representa é carregada em memória. Vale salientar, que um *servlet* só é carregado uma única vez para a memória. Se outras requisições surgirem para um *servlet* já carregado, o mesmo é reaproveitado. O *container* criará uma *thread* do *servlet* para a requisição e passará para ela os objetos **request** e **response** criados.
4. O *container* invoca o método **doGet** ou **doPost** do *servlet*, dependendo do que foi requisitado;
5. O método **doGet** ou **doPost** cria uma página dinâmica que é encapsulada no objeto **response** criado anteriormente;
6. Por fim, o *container* converte o objeto **response** em uma resposta http que é enviada de volta ao cliente, liberando os objetos **request** e **response** criados para a conclusão do processo.

Algumas vantagens fornecidas por um *container* web são listadas abaixo [2]:

- **Suporte de comunicações**. O *container* passa todo código necessário para a *servlet* para se comunicar com o servidor WEB. Sem o *container*, desenvolvedores precisariam escrever o código que iria criar uma conexão *socket* do servidor para a *servlet* e vice-versa e ainda deveria gerenciar como eles falam um ao outro a cada momento.
- **Gerenciamento do Ciclo de Vida**. O *container* conhece o que se passa na vida de suas classes *servlets* desde seu carregamento, instalação, inicialização e recolhimento pelo coletor de lixo.
- **Suporte a múltiplas tarefas**. O *container* controla a função de criar uma nova *thread* a cada momento que uma requisição para a classe *servlet* é realizada.
- **Declaração de Segurança**. Um *container* suporta o uso de um arquivo de configuração XML que pode repassar diretivas de segurança para sua aplicação web sem precisar de um código complexo qualquer dentro do *servlet*.

3.1 Tomcat

O Tomcat [3] é um *container* web robusto e com maturidade, que pode ser usado livremente, tanto para fins comerciais como não comerciais. Tem como principal característica técnica ser centrado na linguagem de programação Java, mais especificamente nas tecnologias de *Java Servlet* e de *Java Server Pages* (JSP).

O Tomcat é um software livre e de código aberto. Através do projeto *Apache Jakarta*, ele teve o apoio oficial da *Sun Microsystems* como Implementação de Referência (RI) para as tecnologias *Java Servlet* e JSP [3].

Devido ao fato de ser escrito em Java, para o seu funcionamento, é necessária a instalação de uma Maquina Virtual Java (JVM) para que possa ser executado. Dessa forma, precisa-se ter uma plataforma java padrão (JavaSE - *Java Standard Edition*) instalada.

Para instalar o Tomcat, basta seguir instruções descritas no site do Apache Tomcat (http://tomcat.apache.org). Após o processo de instalação, uma estrutura de diretórios é criada, como visto na Figura 4 [4].

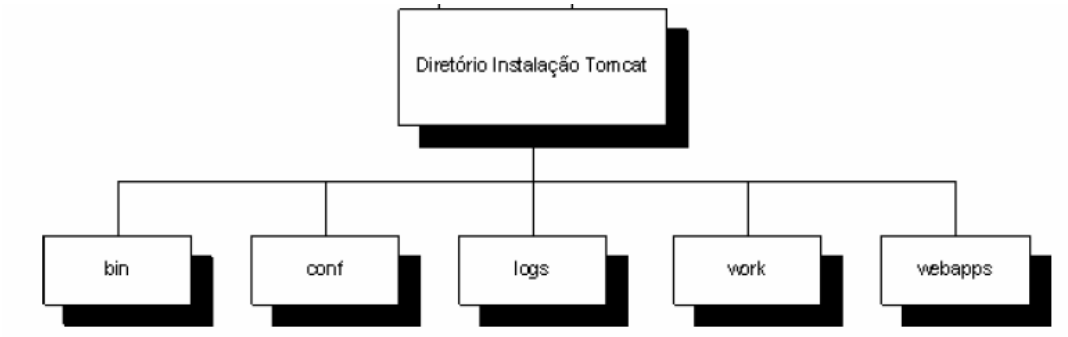

Figura 4 - Estrutura de diretórios do Tomcat [4]

Cada um destes diretórios possui um significado como explicado a seguir:

- **bin**: onde são armazenados alguns executáveis, dentre eles, os aplicativos que iniciam e encerram a execução do servidor.
- **conf**: onde são armazenados os arquivos de configuração do Tomcat.
- **logs**: onde são armazenados os arquivos de log do servidor.
- **work:** é um diretório temporário do Tomcat. É neste diretório que é realizada a recompilação automática de páginas *Java Server Pages* (JSP) [1].
- **webapps**: onde são instaladas as diversas aplicações web pelo programador.

3.2 Estrutura de diretórios de uma aplicação web

A Figura 5 apresenta a estrutura de diretórios padrão de uma aplicação web. O diretório raiz, representado por **JEEBrasil**, neste exemplo, possui páginas HTMLs, figuras, arquivos, páginas JSP, etc. Dentro do diretório raiz, podem ser criados pacotes para organização de classes ou outros diretórios para a organização de arquivos.

A pasta **META-INF** é opcional e pode ser utilizada para a inclusão de meta-informações sobre a aplicação. Já a pasta **WEB-INF** é aquela que todo conteúdo inserido dentro dela não será acessado diretamente via o navegador. O *container* faz o gerenciamento disso, de maneira que o conteúdo dessa pasta só possa ser acessado dentro da aplicação. Ou seja, é como se essa pasta não existisse para o navegador.

A pasta **classes** dentro da **WEB-INF** é onde ficam armazenados os arquivos **.class** das classes compiladas da aplicação. Já n pasta **lib**, devem ser inseridos arquivos JAR de bibliotecas que podem ser utilizadas pela aplicação.

Por fim, o arquivo **web.xml** é um arquivo obrigatório que deve estar na pasta **WEB-INF**. Neste arquivo, são armazenadas configurações da aplicação web.

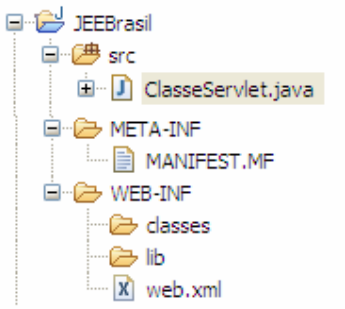

Figura 5 - Estrutura de diretórios de uma aplicação web

4. Primeiro Exemplo com Servlet

Para exemplificar na prática a execução do conteúdo de um *servlet*, considere o exemplo presente na Listagem 3 e que o *Tomcat* já está instalado em sua máquina de trabalho.

Para facilitar, pode-se utilizar a IDE Eclipse (para a elaboração deste material, utilizou-se a IDE Eclipse na versão 3.3.1.1, que pode ser baixada através do site http://www.eclipse.org).

4.1 Criando uma aplicação web no Eclipse

Inicialmente, como mostrado na Figura 6, pode-se criar um projeto **WEB->Dynamic Web Project**. Neste exemplo, o projeto tem nome **Servlet_JSP**. Observa-se que o Tomcat_6.0 foi escolhido para a execução da aplicação.

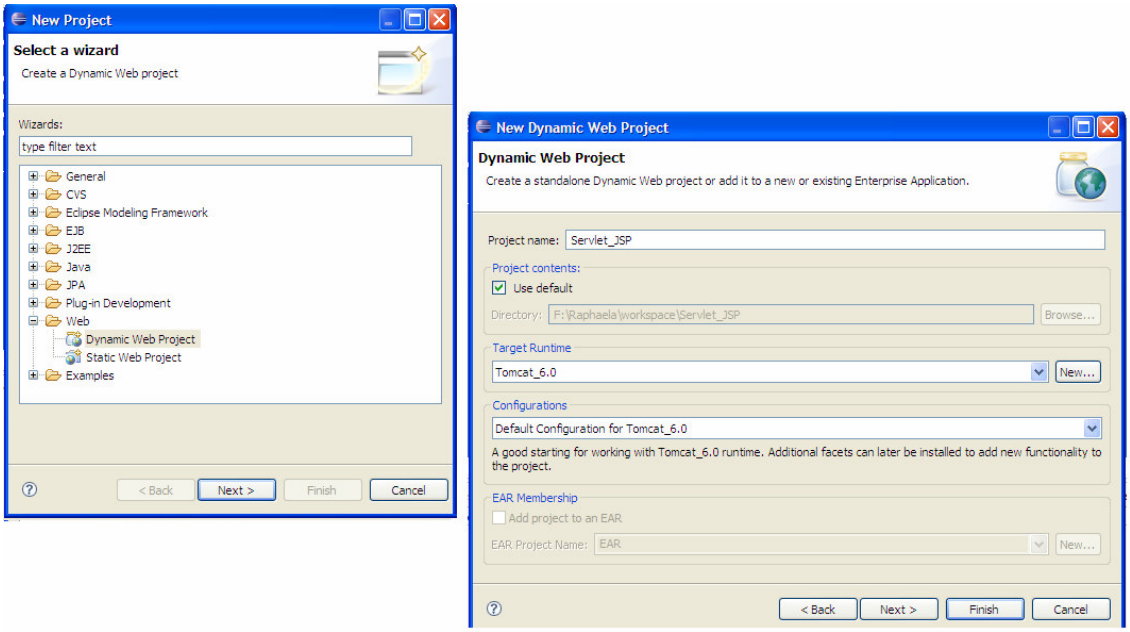

Figura 6 - Criando um projeto no eclipse para executar um servlet

Caso o *Tomcat* não tenha aparecido para ser selecionado na opção **Target Runtime**, adicione-o na aba **Server** do eclipse, clicando com o botão direito e selecionando a opção **New**, como mostrado na Figura 7. Em seguida, utilize a opção **Installed Runtime** para adicionar e realizar as configurações de onde o *Tomcat* foi instalado. Neste exemplo, utiliza-se a versão 6.0 do *Tomcat*.

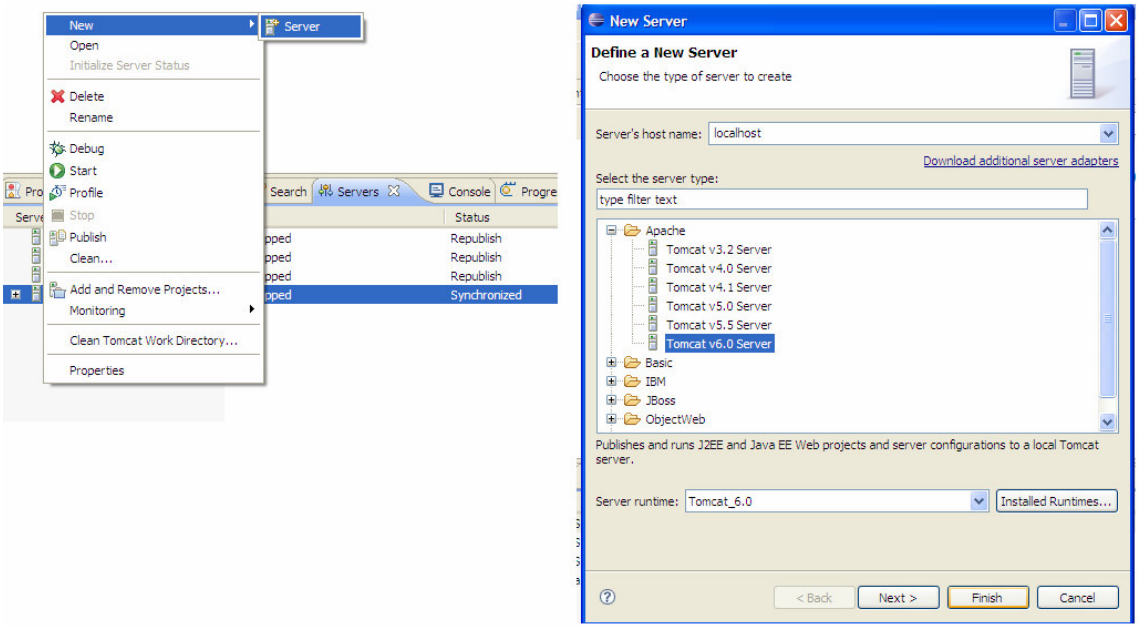

Figura 7 - Adicionando o Tomcat a partir da aba Server do eclipse

A Figura 8 mostra a estrutura do projeto criado no eclipse, com os diretórios essenciais de uma aplicação web. A classe *servlet* da Listagem 3 está contida no pacote **br.com.jeebrasil.servlet**.

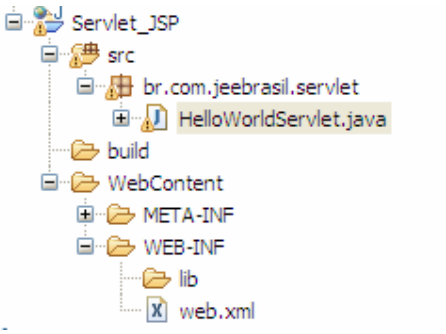

Figura 8 - Projeto Servlet_JSP

4.2 Arquivo web.xml

A invocação da execução do conteúdo da classe servlet é feita através de uma requisição do cliente (navegador web) e é a URL. A URL possui uma descrição e para que o *container* identifique a que classe *servlet* ela se refere, deve haver um passo de mapeamento. Esse mapeamento é feito no arquivo **web.xml**, denominado de *deployment descriptor* (DD).

O arquivo **web.xml** criado pelo Eclipse (Figura 8) possui o conteúdo presente na Listagem 4.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID"
version="2.5">

  <display-name>Servlet_JSP</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

Listagem 4 - Arquivo web.xml criado pelo Eclipse

Dentro da tag **<web-app>** ficarão as configurações da aplicação web. A tag **<display-name>** é utilizada para definir o nome correspondente à aplicação que será exibido no navegador web. Dentro da tag **<welcome-file-list>** são definidos arquivos, usando a tag **<welcome-file>**, que são pontos de entrada padrão para a aplicação quando a URL se refere apenas ao diretório da aplicação.

O mapeamento das classes *servlet* é feito dentro da tag **<web-app>**. Para cada *servlet* existente na aplicação, deve-se ter as tags **<servlet-mapping>** e **<servlet>**, fazendo o seu mapeamento. Dentro dessas tags são mapeados três elementos que representam o *servlet*.

O *servlet* é invocado a partir da URL do navegador através de um nome. Este nome é mapeado a partir da tag **<url-pattern>** da tag **<servlet-mapping>**, ou seja, para este exemplo, o que o cliente conhece é apenas **/urlServletHelloWorld**. Internamente à aplicação, o *servlet* tem um nome diferente do nome de sua classe. Esse nome é mapeado a partir da tag **<servlet-name>** presente nas tags **<servlet-mapping>** e **<servlet>**. Para as duas tags **<servlet-name>**, se elas se referem à mesma classe *servlet*, devem ter o mesmo nome. A tag **<servlet-class**> dentro da tag

<servlet> é utilizada para informar o caminho (pacote) e o nome da classe que representa o *servlet*, neste caso, **br.com.jeebrasil.servlet.HelloWorldServlet**.

```
...

<servlet-mapping>
      <servlet-name>servletHelloWorld</servlet-name>
      <url-pattern>/urlServletHelloWorld</url-pattern>
</servlet-mapping>

<servlet>
      <servlet-name>servletHelloWorld</servlet-name>
      <servlet-class>
            br.com.jeebrasil.servlet.HelloWorldServlet
      </servlet-class>
      <load-on-startup>0</load-on-startup>
</servlet>

...
```

Listagem 5 - Configurando o servlet no arquivo web.xml

4.3 Iniciando o Tomcat

Para iniciar o *Tomcat*, como apresentado na Figura 9, uma das opções é clicar com o botão direito do mouse em cima do nome do projeto e selecionar a opção **Run As -> Run on Server**. Em seguida, deve-se selecionar o *Tomcat* e selecionar a opção **Finish**.

A partir do log exibido na aba **Console** do eclipse, é possível verificar se o procedimento de inicialização do Tomcat ocorreu corretamente. Caso afirmativo, basta invocar o servlet criado, a partir da URL correspondente informada no navegador web.

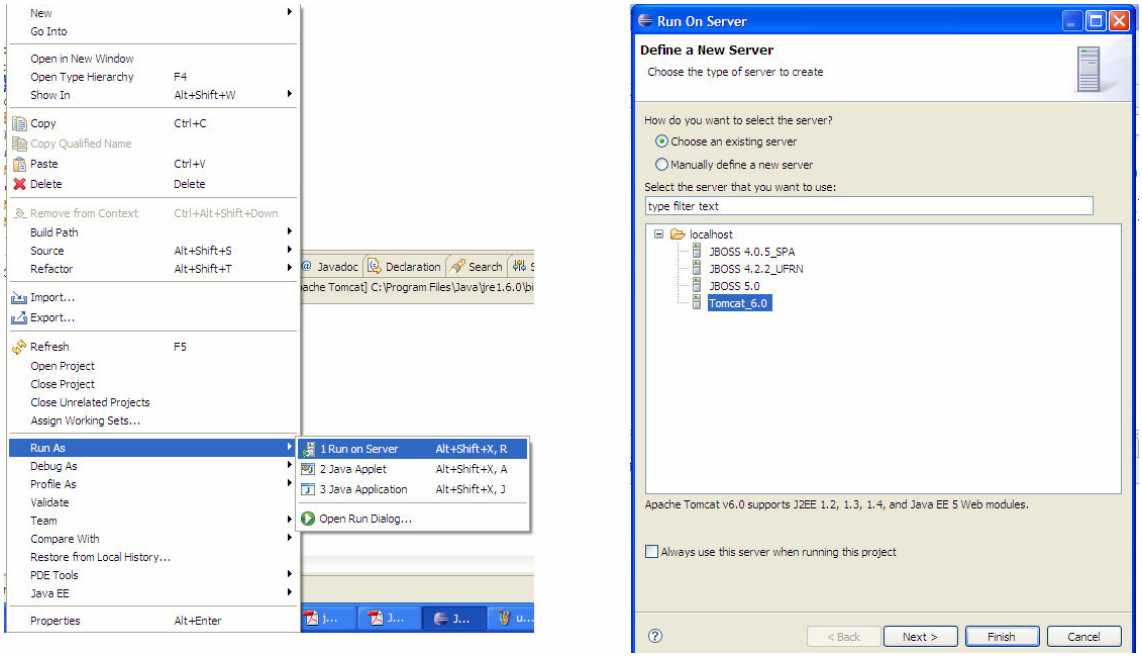

Figura 9 - Iniciando o Tomcat pelo eclipse

4.4 Resultado da aplicação web

A Figura 10 apresenta o resultado da execução do servlet exibido anteriormente. A primeira parte URL informada ao navegador web, http://localhost:8080/Servelt_JSP/urlServletHelloWorld corresponde à localização de onde o servidor foi instalado, no caso a máquina local (localhost). Após os dois pontos, informa-se a porta onde o servidor está instalado (:8080). Em seguida, vem o contexto da aplicação (Servlet_JSP) e o nome que representa o *servlet* criado externamente (urlServletHelloWorld).

Observa-se que o código HTML informado dentro do *servlet* foi exibido no navegador web da forma esperada.

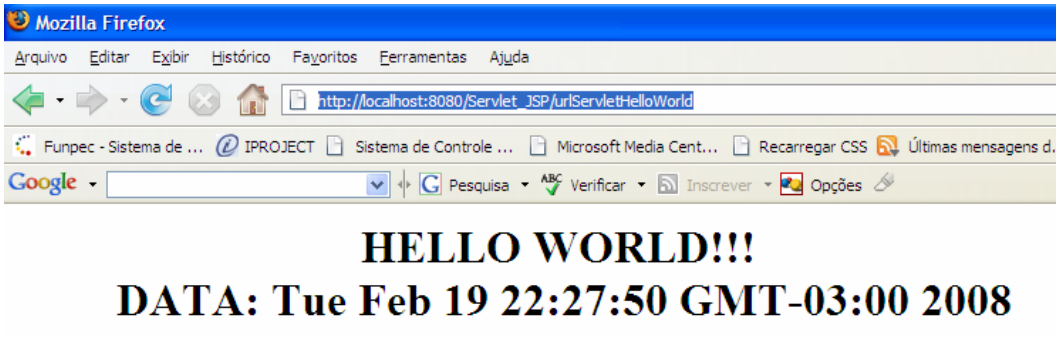

Figura 10 - Resultado da execução do servlet

5. Conclusões

Neste tutorial, procurou-se apresentar um primeiro exemplo utilizando *servlets*, focando no entendimento dos conceitos envolvidos para iniciantes.

A criação de um *servlet* é uma tarefa simples. Uma das coisas que pode parecer confusa é a inserção de código HTML dentro da classe Java (*servlet*). Esse problema foi resolvido com a utilização de *Java Server Pages – JSP* [1].

Com a utilização das JSPs, a estrutura do código HTML não precisa mais ser inserida na classe *servlet*, e sim será inserida em um arquivo denominado de página JSP (extensão .jsp). Dessa forma, a página JSP pode ser alterada sem afetar o restante aplicação.

6. Referências

[1] Head First Servlets e JSP. Bryan Basham, Kathy Sierra e Bert Bates.

[2] Lição 1: Introdução à Programação WEB. Daniel Villafuerte. Material disponível pela Iniciativa JEDI (*Java Education and Development Initiative*). Novembro/2007.

[3] Apache Tomcat - http://tomcat.apache.org/. Acesso em 18 de fevereiro de 2008.

[4] Programação Web com JSP, Servlets e J2EE. André Temple, Rodrigo Fernandes de Mello, Danival Taffarel Calegari, Maurício Schiezaro. 2004.