

Software development line
IT University of Copenhagen
Copenhagen, Denmark
Supervisor: Kasper Østerbye
Spring 2001

Elastic JavaDoc

An elastic information system for documentation of Java development

**TECHNOLOGY
BACKGROUND**

Written by:
Kasper B. Graversen

© 2001

Preface

This report is a supplement to the EJD report. You should therefore start reading the other report and return here after reading chapter 1 if need be.

This report covers the technoly background:

Description of the current tags in Java API

What the techonologies JDBC, Servlet and JSP are

And how the JavaMiner tool works internally.

Contents

0	Javadoc tags	1
1	The technological platform	3
1.1	JDBC	3
1.2	Generally about dynamic webpages and Java	5
1.3	JavaBeans	5
1.4	Servlet	6
1.4.1	Total encapsulation	7
1.4.2	The webserver environment	8
1.4.3	How to plugin a servlet	9
1.5	JSP	10
1.5.1	What is JSP	10
1.6	Custom tags	11
1.6.1	What are custom tags	12
1.6.2	How to plugin a custom tag	13
1.6.3	Development of type I tags	14
1.6.4	Development of type II tags	15
1.6.5	Development of type III tags	16
2	The EJD skeleton	18
3	JavaMiner	19
3.1	Usage	22

0. Javadoc tags

Below is a list of the tags currently supported by the Javadoc tool. It is an edited version of the material at <http://java.sun.com/j2se/javadoc/writingdoccomments/index.HTML>

Tag	Description
@author	If the author is unknown, use "unascrbed" as the argument to @author.
@version	The Java Software convention for the argument to the @version tag is the SCCS string "%I%, %G%", which converts to something like "1.39, 02/28/97" (mm/dd/yy) when the file is checked out of SCCS.
@param	The @param tag is followed by the name (not data type) of the parameter, followed by a description of the parameter. By convention, the first noun in the description is the data type of the parameter. (Articles like "a", "an", and "the" can precede the noun.) An exception is made for the primitive int, where the data type is usually omitted. Additional spaces can be inserted between the name and description so that the descriptions line up in a block. Dashes or other punctuation should not be inserted before the description, as the Javadoc tool inserts one dash. Parameter names are lowercase by convention. The data type starts with a lowercase letter to indicate an object rather than a class. The description is most usually a phrase, starting with a lowercase letter and ending without a period, unless it contains a complete sentence or is followed by another sentence (as described further below).
@return	Omit @return for methods that return void and for constructors; include it for all other methods, even if its content is entirely redundant with the method description. Having an explicit @return tag makes it easier for someone to find the return value quickly. Whenever possible, supply return values for special cases (such as specifying the value returned when an out-of-bounds argument is supplied).
@deprecated	The @deprecated description in the first sentence should at least tell the user when the API was deprecated and what to use as a replacement. Only the first sentence will appear in the summary section and index. Subsequent sentences can also explain why it has been deprecated. When generating the description for a deprecated API, the Javadoc tool moves the @deprecated text ahead of the description, placing it in italics and preceding it with a bold warning: "Deprecated". An @see tag (for Javadoc 1.1) or @link tag (for Javadoc 1.2 or later) should be included that points to the replacement method:

Tag	Description
@since	<p>Specify the product version when the Java name was added to the API specification (if different from the implementation). For example, if a package, class, interface or member was added to the Java 2 Platform, Standard Edition, API Specification at version 1.2 The Javadoc standard doclet displays a "Since" subheading with the string argument as its text. This subheading appears in the generated text only in the place corresponding to where the @since tag appears in the source doc comments (The Javadoc tool does not proliferate it down the hierarchy). (The convention once was "@since JDK1.2" but because this is a specification of the Java Platform, not particular to the Sun JDK or SDK, we have dropped "JDK".)</p> <p>When a package is introduced, specify an @since tag in its package description and each of its classes. (Adding @since tags to each class is technically not needed, but is our convention, as enables greater visibility in the source code.) In the absence of overriding tags, the value of the @since tag applies to each of the package's classes and members.</p> <p>When a class (or interface) is introduced, specify one @since tag in its class description and no @since tags in the members. Add an @since tag only to members added in a later version than the class. This minimizes the number of @since tags.</p> <p>If a member changes from protected to public in a later release, the @since tag would not change, even though it is now usable by any caller, not just subclasses.</p>
@exception	<p>An @exception tag should be included for any checked exceptions (declared in the throws clause), as illustrated below, and also for any unchecked exceptions that the caller might reasonably want to catch, with the exception of NullPointerException. Errors should not be documented as they are unpredictable. For more details, please see Documenting Exceptions with the @throws Tag.</p>
@throws	is a synonym added in Javadoc 1.2
@serial	For object Serialization
@serialField	For object Serialization
@serialData	For object Serialization
@link	For conventions, see Use In-Line Links Economically.

1. The technological platform

Many people know Java or at least an Object-Oriented language, fewer may know how to communicate with a database, or how to create dynamic webpages in Java. To the extent the technologies are used in the EJD, we present the technologies and relate them to where they are used.

Basically the project uses two different kinds of technology: Database connectivity and dynamic web pages. The database connectivity is in Java implemented under the framework named JDBC, and is found in the API hierarchy under the package name `java.sql.*`.

The dynamic webpages implementation is an even bigger framework, which are subdivided into three categories: JavaBeans, Servlets and JSP. These are found in the API hierarchy under `javax.servlet.*`, `javax.servlet.http.*`, `javax.servlet.jsp.*` and `javax.servlet.jsp.tagext.*` respectively.

For both technologies an important observation is that much of the API are interfaces specifying behaviour, leaving it up to the database- and webserver-vendors to implement these behaviours with whatever approach they find suitable.

The chapter is written on basis of [Hall: 2000; Hougland and Tavistock: 2001; Pelegrí-Llopart and Cable: 1999] and personal knowledge accumulated over time.

1.1 JDBC

JDBC is an object-oriented approach to connecting and communicating with a database. JDBC can be translated to ODBC which is a widely used database communication protocol established by Microsoft. As most databases are ODBC compliant, using a specific database in Java yields few, if any, problems. On figure 1.1 page 4 shows the typical classes used when communicating with a database. The figure also shows the relations between the classes.

Connecting to the database happens through a Database Driver, which returns a connection. Usually the driver is loaded manually using the Java classloader. Connecting to a MySQL database yields the following code

```
String driver = "org.gjt.mm.mysql.Driver";
String logininfo = "jdbc:mysql://localhost:3306/test?
user=root&password=god";

Class.forName(driver).newInstance();           // load driver
conn = DriverManager.getConnection(logininfo); // make connection
```

Having a connection to the database a `Statement` object must be created in which ordinary SQL statements appear and are executed. When creating a `Statement` one can specify how the result will be treated, i.e. if the result will be iterated many times or only read once. After executing a SQL statement the result is retrieved in the form of a `ResultSet` object. The following code shows an iteration of all the usernames in a user table in the database.

```
Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                     ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT name FROM user");
```

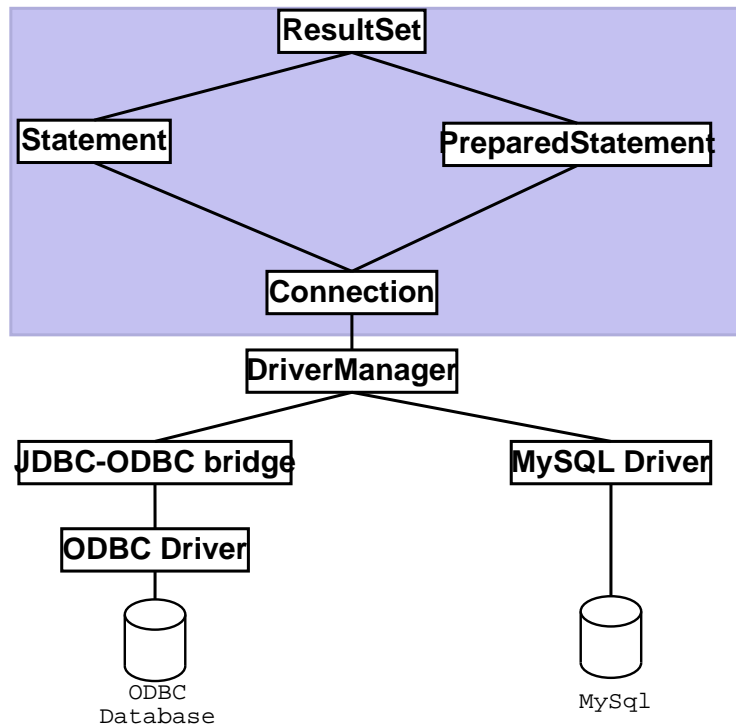


Figure 1.1: The hierarchy involved when connecting to a database. The java application is denoted with a blueish background, while the outside denotes classes or programs outside the application.

Another type of statement is the “prepared statements”, which are precompiled by the database for faster execution, and more importantly, are used for statements which can not be put on a textual form, i.e. pictures or soundclips. Inserting a picture in a user table is shown below:

```

PreparedStatement pstmt = conn.prepareStatement(
    "INSERT INTO User (name, pic) VALUES (?, ?)");
pstmt.setString(1, "John Doe");
pstmt.setBytes(2, serialize(picture));
pstmt.executeQuery();
  
```

Where `serialize()` is a custom method converting the image to a byte array.

Results from requests are places in a `ResultSet` object, which can be traversed a row at a time. The following code is an extension of the previous example where all usernames were retrieved:

```

while(rs.next())
    System.out.println("name: " + rs.getString(1));
  
```

Finally there exists `MetaData` classes for retrieving metainformation about the result, the names and types of the columns.

Summary:

We have here shown the basics of the JDBC framework which is sufficient for this project. Database connectivity is used in the JavaMiner doctlet for inserting raw information into the database. In the EJD the class `MrFetcher` makes queries to the database before pages are generated.

1.2 Generally about dynamic webpages and Java

Dynamic webpages can be created in Java, but needs be run on a webserver. To make the technology as generic as possible the dynamic web elements (servlets and jsp) are to be understood as buildingblocks which is plugged into the webserver. The webserver interfaces the buildingblocks which “by contract” behaves a certain way. The following sections will describe the basic elements.

In short: JavaBeans can be used by both Servlets and JSP pages. Servlets are compiled classes which handles and processes a (Http) request and returns a response – typically the result the browser receives. JSP pages are HTML-looking pages where Java code can be imbedded. The concept is that programmers code the beans and the servlet, whilst the web maintainer produces and maintains the JSP files. While JSP can be completely shielded from programming statements and as the all the business logic is placed in beans and servlets, the non-programmers can focus on the layout of the webpage.

1.3 JavaBeans

Beans is an abstraction for data. This data could be a shopping basket or the user logged in, where the properties could be articles in the basket, user name, user address etc. The basic beans simply exposes a number of properties (accesable through methods), by following a simple rule regarding the method names. The bean provides accessor and mutator methods which are named after the property they modify — so the property named “foobar” would have the methods `getFoobar()` and `setFoobar()` associated with it. The naming convention elegantly defines the link between the property and the accessor/mutator methods. Internally a bean is just an instance of a Java class and is called a bean when the naming convention is respected.

When using a bean within a servlet, the bean is in all respects treated the same way a normal Java class. Using the bean in JSP is a completely different matter. To make things easy for the non-technician using the bean object instantiation and method invocation are hidden in a HTML-like syntax. The following example shows the creation of a bean followed by initializing it by calling its `setName()` method and finally retrieving the property name by calling `getName()`.

Example 1.1:

```
<jsp:useBean id="handle" class="NameBean"/>
<jsp:setProperty name="handle" property="name" value="John Doe"/>

Hello John, you live at:

<jsp:getProperty name="handle" property="address"/>
<jsp:getProperty name="handle" property="city"/>
```

□

Basically the above code is equivalent to

```
NameBean handle = new NameBean();
handle.setName("John Doe");
handle.getAdress();
handle.getCity();
```

While there really is no magic going on, setting and retrieving data for the non-technician has become a matter of just using some new tags looking like the HTML he is already familiar with.

Apart from the syntactical sugar the example clearly shows, the separation of presentation and logic. The bean could be a proxy to an underlying database, looking up “John Doe” and then servicing information about the person.

1.4 Servlet

Servlets are pluggable “chunks” that extends the functionality of the webserver. Servlets are Java classes implementing typically the `HttpServlet` interface. As the EJD project operates by means of `Http`, the focus is put here. The `HttpServlet` interface has four interesting methods

void init() Is called when the servlet is being initialized by the webserver, is comparable to an ordinary class’ constructor.

void destroy() The destructor of the servlet, here cleanup code is placed.

void doGet()/doPost() Is called whenever a webpage is requested matching the specific servlet. The method produces (parts of) the webpage.

To develop a servlet one must at least implement the `doGet()` or `doPost()` method. The lifecycle of a servlet is shown on figure 1.2. The servlet is entirely controlled by the webserver it is “plugged into”. The servlet has no ordinary constructor, instead the method `init()` is called on instantiating the servlet. Whenever the webserver gets a request for a page represented by a servlet, the `doGet()` or `doPost()` method of that servlet is invoked. As the invocation is done by threads in the webserver, several threads could potentially reside in the servlet simultaneously. If the servlet contains thread critical code, implementing the `SingleThreadModel` interface or using the `synchronize(this)` construct must be utilized.

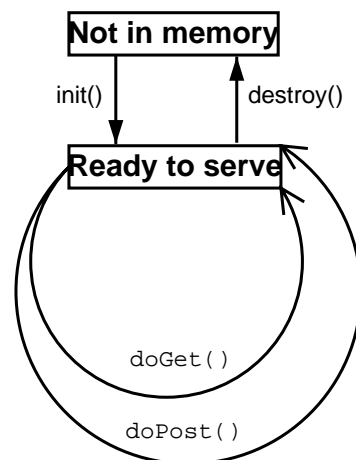


Figure 1.2: *The lifecycle of a servlet. Between initialization and destruction the webserver's threads execute the code of the servlet on behalf of request.*

The reason for having both `doPost()` and `doGet()` methods for handling request ascribes the fact that the `Http`-protocol contains two methods for including data in the request.

1.4.1 Total encapsulation

It is of substantial importance to understand the architecture of the servlet and the webserver. As figure 1.3 page 7 shows, the servlets are completely secluded, *all* communication takes place by dint of the server. This is corroborated in the definition of the `do*` methods — the methods are declared as void. The request and response objects are given as parameters to the methods as they are setup by the server before handled to a servlet. This design enables several servlets to contribute to the same response, as the first servlets response object is given to the next servlet which then writes its response, etc.

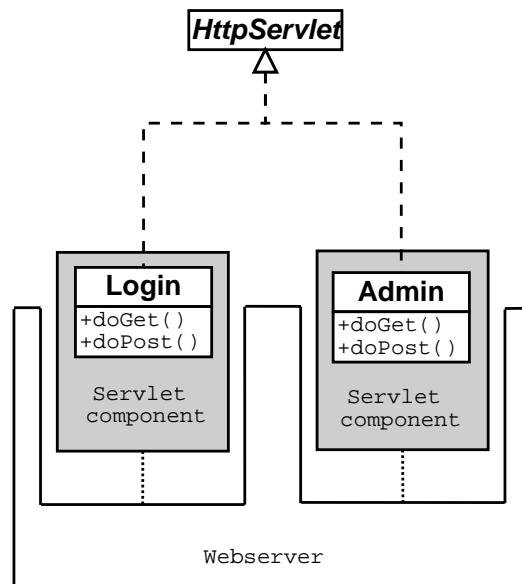


Figure 1.3: Two servlets residing in containers in the webserver. Notice all communication (be it other servlets or "the outside world") takes place by dint of the server.

There are two salient points for this architecture, strict communication enables the internals of the webserver to be freely designed. Existing webserver can therefore with less difficulty become servlet compliant. Secondly the control enforces services disposal to the servlet. The webserver has a huge framework of utility classes, and ordinary things such as fetching and parsing parameters in the request, or forwarding the request to other servlets are automated.

To make things tangible we now show a simple servlet—"hello world" example; at the request `http://localhost/servlet/HelloWorld?name=kasper` the servlet produces a page stating "Hello kasper".

Example 1.2:

```
1 public class HelloWorld extends HttpServlet
2 {
3     public void init(){}
4     public void destroy(){}
5     public void doPost(HttpServletRequest req, HttpServletResponse res)
6     {     doGet(req, res);}
7
8     public void doGet(HttpServletRequest req, HttpServletResponse res)
9     {
10         res.setContentType("text/HTML");
11         PrintWriter out = res.getWriter();
12         out.println("<HTML><head>
13         out.println("<title>Helloworld"</title>");
14         out.println("</head><body>");
15         out.println("<h1>Hello ");
16         out.println(req.getParameterValue("name"));
17         out.println("</h1>");
18         out.println("</body></HTML>");
19     }}
```

The `init()` and `destroy()` methods are empty since the servlet uses nothing that needs instantiation or cleaning up. The `doPost()` calls `doGet()` so the servlet are compliant with both types of Http request. The `doGet` first gets a `Writer` from the response object, and prints away. □

1.4.2 The webserver environment

The servlet has access to several objects offered and maintained by the webserver. Figure 1.4 page 9 shows a reduced view of the environment of a servlet (it only contains the elements used in the development of EJD).

The three important classes on the figure are

HttpSession Http is a “connection less” protocol, which means that when the users result has been generated and completely transmitted, the user is disconnected. In the old days of web programming this obstacle was solved by keeping the persistent information in hidden fields in the response to the user. The information would then be transferred to the server again at the next request. The solution in modern web languages is to place a session object on the webserver. As there is no telling whether the user has closed his browser or is visiting other web sites, a timer is placed in the object, which is renewed when the user re-connects. If the timer runs out the session object is invalidated. To keep track of users, the user automatically receives a cookie containing a key which can identify the session object on the server. To the programmer `HttpSession` object acts as any other storage datastructure in Java with `get-`, `set` and `remove` methods. An additional `invalidate()` is added to force a new session upon the user.

ServletRequest As seen in operation in example 1.2, the request object allows easy manipulation and information retrieval of the request (the object contains a total of 25 methods).

RequestDispatcher Is used for redirecting the request to another servlet internally in the server. The user is unaware of this and wont

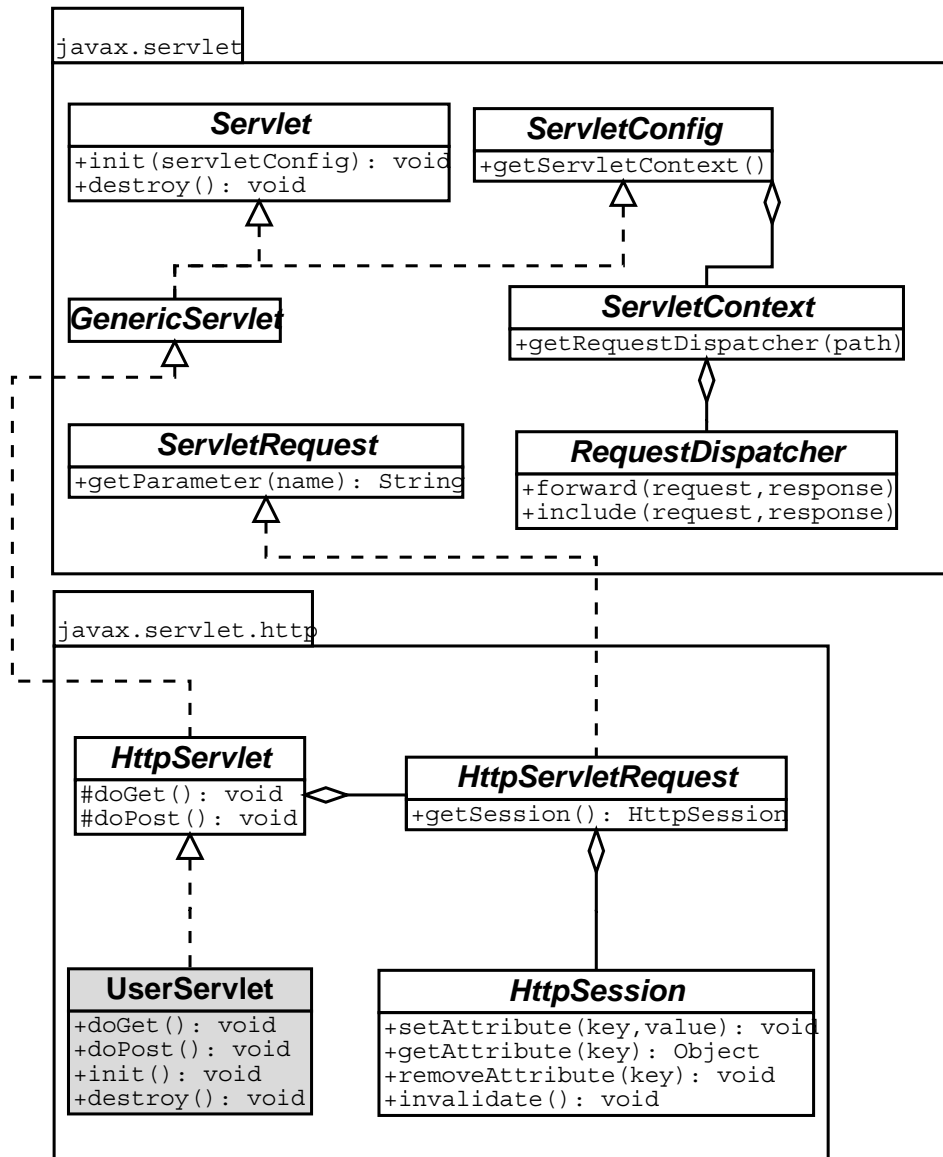


Figure 1.4: A reduced view of the the environment servlets live in. The three important classes are `HttpSession`, `ServletRequest` and `RequestDispatcher`. The `HttpServletResponse` is important but for this project plays a minimum role and thus omitted.

1.4.3 How to plugin a servlet

Servlets are “plugged in” to webserver in a fairly easy way. When the webserver starts running it first reads the configuration file `server.xml`. This file contains information which ties a directory with classes (compiled servlets) with an URL. This means that there is no connection between the URL and the physical placing of servlets on the server. An example configuration line would be

```
<Context path="/foo" docBase="webapps/mycode" debug="9" reloadable="true"
trusted="false"></Context>
```

Which links classes found in the directory `webapps/mycode` (relative to where the webserver is installed) to URL’s starting with `/foo`, i.e. `localhost://foo/servlet/HelloWorld`

1.5 JSP

Looking closer at example 1.2 page 8 we find more than just the principals of servlets syntactics. Even from such a small example it is obvious that there are problems with the distribution of responsibilities. [Gamma et al.: 1995] states good programming practice is to divide applications into a model, a view and a controller part. Not only is the three intermixed in the example, the example also illustrates the substantial amount of typing required. It quickly becomes an onerous and time consuming task to do long HTML pages when an `out.println()` call per HTML line is needed.

Another drawback of servlets are that they require programmers to design the pages be displayed. In efficient web productions, layout and code are developed in parallel, but using servlets alone requires web ;designers to be able to program.

1.5.1 What is JSP

JSP is HTML pages with arbitrary embedded blocks of Javacode inside. Like PHP or other scripting languages the code needs be surrounded by code indicator tags. In JSP there exists three different scripting elements: the scriptlet element, the expression element and the declaration element — presented here in the same order.

`<% %>` Inside these blocks reside the code. This code goes by the name *scriptlets*.

`<%= %>` This is a shortcut replacing `<% out.println() %>` calls.

`<%! %>` Its purpose is to initialize variables and methods and make them available to other scriptlets, expressions and declarations.

The technique used for turning text into webpages is simply to compile the JSP pages into servlets thereby exploiting the servlet framework. All the scripting elements gets added directly into the servlet code that is to be generated. This means that compile time errors can occur as well as runtime errors in your JSP page. Depending on the webserver pages can get precompiled, or compiled runtime the first time a user requests the page.

The following example shows the code generation of a simple JSP page to a servlet don by the JRun webserver.

```
1 <HTML>
2 <head>
3 <title>Greetings</title>
4 </head>
5 <body>
6 <% for(int i = 0; i < 5; i++) { %>
7 <h1>Hello World!</h1>
8 <% } %>
9 </body>
10 </HTML>
```

Notice how the lines 6–8 corresponds to the lines 26–28 in the generated servlet below

```

1 import javax.servlet.*;
2 import javax.servlet.http.*;
3 import javax.servlet.jsp.*;
4 import javax.servlet.jsp.tagext.*;
5 import allaire.jrun.jsp.JRunJSPStaticHelpers;
6 public class jrun__MyFirstJspPage2ejspl3 extends allaire.jrun.jsp.HttpJSPServlet
7 implements allaire.jrun.jsp.JRunJspPage
8 {
9     private ServletConfig config;
10    private ServletContext application;
11    private Object page = this;
12    private JspFactory __jspFactory = JspFactory.getDefaultFactory();
13    public void _jspService(HttpServletRequest request, HttpServletResponse response)
14        throws ServletException, java.io.IOException
15    {
16        if(config == null) {
17            config = getServletConfig();
18            application = config.getServletContext();
19        }
20        response.setContentType("text/HTML; charset=ISO-8859-1");
21        PageContext pageContext = __jspFactory.getPageContext(this, request, response, null, true, 8192, true);
22        JspWriter out = pageContext.getOut();
23        HttpSession session = pageContext.getSession();
24        try {
25            out.print("<HTML>\r\n<head>\r\n<title>Greetings</title>\r\n</head>\r\n<body>\r\n");
26            for(int i=0;i<5;i++) {
27                out.print("\r\n<h1>Hello World!</h1>\r\n");
28            }
29            out.print("\r\n</body>\r\n</HTML>\r\n\r\n");
30        }
31        catch(Throwable t) {
32            if(t instanceof ServletException)    throw (ServletException) t;
33            if(t instanceof java.io.IOException) throw (java.io.IOException) t;
34            if(t instanceof RuntimeException)     throw (RuntimeException) t;
35            throw JRunJSPStaticHelpers.handleException(t, pageContext);
36        }
37        finally {
38            __jspFactory.releasePageContext(pageContext);
39        }
40    private static final String[] __dependencies__ = {"/MyFirstJspPage.jsp",null};
41    private static final long[] __times__ = {982776754000L,0L};
42    public String[] __getDependencies() {return __dependencies__;}
43    public long[] __getLastModifiedTimes() { return __times__; }
44    public int __getTranslationVersion() {return 14;}
45 }

```

1.6 Custom tags

Although business logic can be placed in servlets and beans, JSP pages often become cumbersome with hard to read chunks of code. From a tutorial at SUN¹ a webpage displaying the contents of a shopping cart is given

¹at <http://java.sun.com/j2ee/tutorial/doc/JSPIntro11.HTML>

```

<%
  Iterator i = cart.getItems().iterator();
  while (i.hasNext())
  {
    ShoppingCartItem item =
      (ShoppingCartItem)i.next();
    ...
  }
%>
  <tr>
    <td align="right" bgcolor="#ffffff">
      <%= item.getQuantity() %>
    </td>
    ...
  <%
  }
%>

```

By using a custom tag for iteration of the elements the code logic is eliminated and the JSP code is much easier to read:

```

<logic:iterate id="item"
  collection="<%=cart.getItems()%>"
  <tr>
    <td align="right" bgcolor="#ffffff">
      <%=item.getQuantity()%>
    </td>
    ...
  </logic:iterate>

```

1.6.1 What are custom tags

To eliminate confusion we first define the terminology of tags

```

< basket : iterator type="discount" > ... </basket : iterator >
   prefix      tagname      Parameter w. value      body      endtag

```

Custom tags are simply Java classes extending certain interfaces and implementing certain methods. Tags are pluggable entities which can be “plugged into” a webpage, as servlets can be “plugged into” a webserver. When developing a custom tag one needs only concentrate on the functionality of the tag. The parsing of the JSP page is done by the webserver, attributes are found and automatically feed by the server as well. If the JSP page is ill-formatted the webserver throws an exception

In the EJD project we are using three different kinds of custom tags. We give them numbers so we can refer back to them:

Type I Tag with no body. In JSP written as “<tag/>”

Type II Tag with body with or without parameters.

Type III Tag with dynamic parameters, using custom tags as parameter values. This type of tag is not possible to create in JSP!

Figure 1.5 page 13 shows some of the many classes existing in the JSP framework. The two blue boxes are the two interfaces extended by the EJD tags. An important observation is that from the `TagSupport` interface it is possible to reach the session object. This means that servlets and tags can place data in the session object which are reachable for other tags. This is exactly how EJD works.

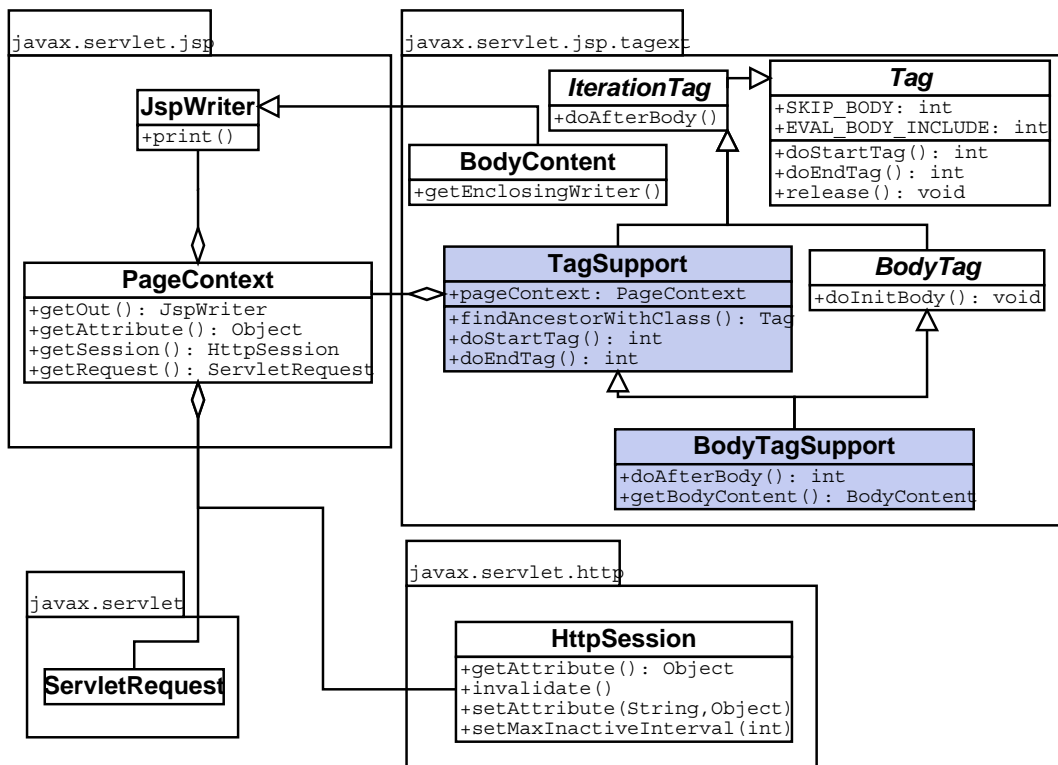


Figure 1.5: Taglibs seen from the programmers point of view... more text here..

1.6.2 How to plugin a custom tag

Tags are grouped in tag libraries (also called “taglibs”). The tag library is defined by an XML file whose last name is “.tld”. The tag libraries are made available to a JSP page via a taglib directive:

```
<%@ taglib uri="ejd-taglib.tld" prefix="ejd" %>
```

Where `ejd-taglib.tld` is the XML file defining the tag library. The `prefix` parameter defines the prefix to all tags from that library. This prevents name clashing when two libraries define tags with identical names.

The tld-file

The format of the tld file is simple. Apart from a header defining the tag libraries name and description, each tag is defined with its corresponding Java class file. Such as

```
<tag>
  <name>class</name>
  <tagclass>ejd.displaytag.ClassTag</tagclass>
  <info>Provides information on a given class</info>
  <bodyContent>JSP</bodyContent>
</tag>
```

Where `bodyContents` is set to `EMPTY` instead of `JSP` if the tag has no body. If the tag has parameters, each parameter needs be explicitly specified as well:

```

<attribute>
  <name>target</name>
  <info>Set the target frame the result is to appear in</info>
  <required>false</required>
  <rtexprvalue>false</rtexprvalue>
</attribute>

```

The `<attribute>` needs be defined within the `<tag>` scope, typically placed after `<bodyContent>`. The `<required>` tag defines whether the parameter is optional or not with values false or true. The `<rtexprvalue>` defines if the attribute can have scriptlet expressions as a value or the value needs be static. False means static values. Setting the value to true lets the value be i.e. dynamically calculated, but it does not enable custom tags be used as value.

It is notable that there is no way to define a hierarchy among the tags, one can not define that one tag needs be nested inside another tag. This must be manually coded in the tags which on one hand seems tedious. On the other hand, tags requiring being within the scope of another tag often needs information or functionality of that tag. To access it, it needs a handle to the “outer tag” which needs be coded. The only code saved by having a specification as to where tags are legally placed, is the error handling in the tags. On the other hand such a specification could be highly usable in a tools such as EJD as it would be able to give an overview over the usage of the taglib (note the EJD currently does not support tld files).

Also it is not possible to create valueless parameters, all parameters must supply a value. Nor can parameters be grouped such that (A and B) or C are required. In such a case all three parameters must be specified optional.

1.6.3 Development of type I tags

The type I tags are used in EJD for outputting presentation information, i.e. the classname or code for radiobutton.

Tags with no body must extend class `TagSupport` and at least implement the `doStartTag()` method. The `TagSupport` is chosen over the `Tag` as this enables the tag to consult i.e. configurations objects placed in the session. A `<helloworld>` tag could be defined as:

```

public class HelloWorldTag extends TagSupport
{
  public int doStartTag()
  {
    JspWriter out = pageContext.getOut();
    out.print("Hello world");
    return(SKIP_BODY);
  } }

```

In order to use the tag in a JSP file, an entry in an tld file must be made, and the tld file must be included in one of the JSP page taglib directives.

All tags for documentation presentation in EJD uses data from other tags and must thus placed in the scope of those tags. In order to retrieve the data, a handle to the tag is needed. The following code finds an ancestor and prints some information from it.

```

AnoterTag handle = (AnotherTag) findAncestorWithClass(this,
                                                    Class.forName("AnotherTag"));
out.print(handle.nice_data);

```

If the tag is not to be found an exception is thrown. Caveat: A `getParent()` method exist in `Tag` which is easier to call as it requires no parameters. However in EJD one can not know what the parent tag is, hence the method is tempting but is worthless to use.

1.6.4 Development of type II tags

Type II tags are typically either setting up information for type I tags or creating some “initial data” before the type I tags produce their output. Developing tags with a body is a bit more complicated than bodiless tags. We have split the explanation in three parts to make things easier.

Handling the body

When a tag has a body it is in charge of handling the output of the body. In EJD this meant blindly outputting it, but i.e. a filtering mechanism could be implemented. Outputting the content of the body is a simple matter of retrieving the content and printing it.

```
public class MyIteratorTag extends BodyTagSupport
{
    public int doEndTag() throws JspException
    {
        if(bodyContent != null)
            bodyContent.writeOut(bodyContent.getEnclosingWriter());
        return EVAL_PAGE;
    }
}
```

A if-then-else tag which has the following form

```
<if>
  <condition> some condition </condition>
  <then>Output when true</then>
  <else>Output when false</else>
</if>
```

Could internally work as: `<condition>` sets a boolean in the `<if>` tag. The `<then>` and `<else>` tags checks the value in `<if>` before outputting their body. Idea from [Hall: 2000,p344–350].

Iterating over the body

The `IterationTag` interface makes it possible to iterate over the body, executing the body at each iteration. The `doAfterBody()` is called after each iteration. A simple iterator could be implemented as

```
public class MyIteratorTag extends BodyTagSupport
{
    int i = 0, max = 10;

    public int doAfterBody()
    {
        if(i < max) return EVAL_BODY_TAG;
        else      return SKIP_BODY;
    }

    public int doEndTag()
    { // defined as before
    } }
}
```

Producing 10 'hello world' could done with

```
<iterator>
    Hello world
</iterator>
```

Handling parameters

Handling parameters is easy. There is a direct link between the specification in the tld file and the Java class file. When the `target` parameter is specified in the tld file, the Java class is required to contain the method `public void setTarget(String s)`. A particular nice feature of JSP is, that when the JSP file is being executed calls to the mutator methods of the tag class files are automatically called, hence the only difference between tags with and without parameters are the required mutator methods in the tags with parameters.

Extending the previous example to

```
<iterator max="4">
    Hello world
</iterator>
```

is a matter of extending the previous code with a single method (and extend the tag definition in the tld file).

```
public void setMax(String m)
{
    max = Integer.parseInt(m);
}
```

1.6.5 Development of type III tags

Currently the type III tag is only needed in the `<link>` tag. The role of the `<link>` tag is to make elements on the page clickable. Typically the elements are dynamically generated by an iterator so parameters are not known when the JSP file is developed. The usage could be

```
<iterator>
    <link source="<name>"> <name> <otherstuff> </link>
</iterator>
```

In order to circumvent the restrictions in the JSP language, we came up with a solution inspired by the if-then-else tag explained earlier. As tags can have tags as their body a `<parameter>` tag was invented which took its body and called a method in the `<link>` tag.

```
<iterator>
  <link>
    <parameter type="source"><name></parameter>
  </link>
</iterator>
```

When the `</link>` is met, all information gathered from the nested tags are analyzed at output is generated.

2. The EJD skeleton

The following figure shows how the implementation of the system was constructed.

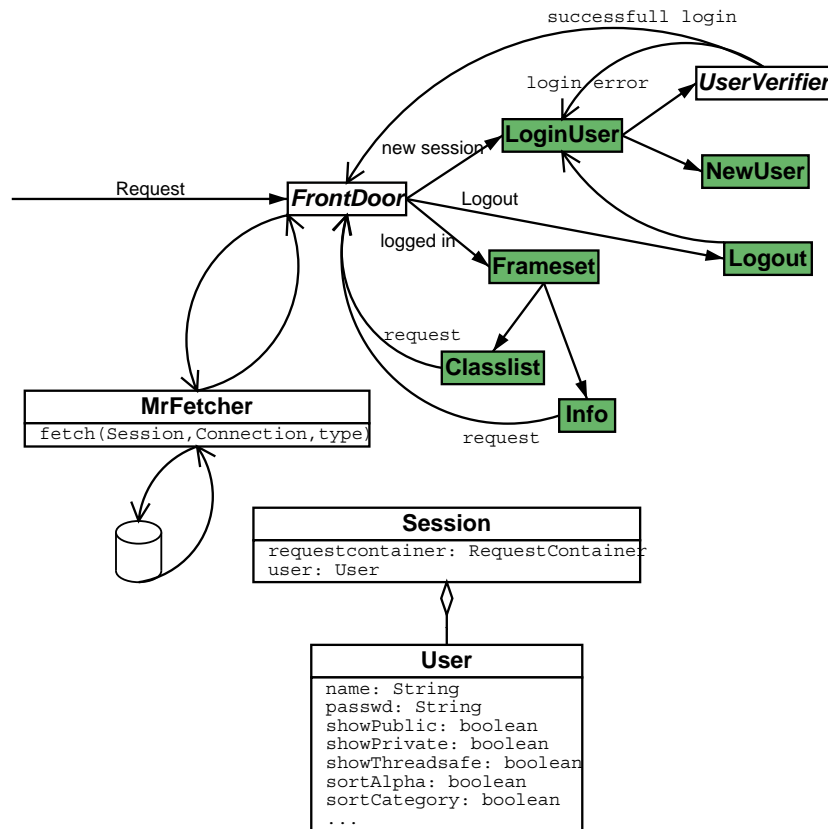


Figure 2.1: The flow of the EJD system. The dark boxes are JSP pages while the white boxes are servlets.

The point of entry is the servlet called `FrontDoor`. The front door ask the webserver of the existence of a `User` object in the current session. If the result is negative, the request is passed on to the JSP page `LoginUser` which contains a welcome message and a HTML `<FORM>` containing name and password fields. When the user submit his data the `UserVerifier` examines the data. If the data are valid the users information is retrieved from the database and stored in a `User` object placed in the users session. The request is then forwarded to the `Frontdoor` which forwards the request to the JSP file `Frameset`. `Frameset` contains `<frameset>` definitions which defines the shapes and how many frames the browser window be divided in.

When the client receives the response the browser will automatically request the pages defined within it. On the figure two pages are retrieved `clacelist.jsp` and `info.jsp`. The first showing all classes and interfaces, while the later show detailed information of classes/interfaces chosen from the first list — just like the normal `JavaDoc`.

However the design leaves the display open. Another useful way of displaying the documentation would be a three window split.

The JSP file generates its output on the basis of `requestContainer` object placed in the users session. The `requestContainer` contains the result from the database query fetched by `MrFetcher`.

3. JavaMiner

Creating a doclet seems at first fairly easy. The following code outputs all the classes found when calling the Javadoc tool.

```
public class JavaMiner extends Doclet
{
    public static boolean start(RootDoc root)
    {
        ClassDoc[] classes = root.classes();
        for(int i = 0; i < classes.length; i++)
            System.out.println(classes[i].name());
    }
}
```

We get a reference to a `ClassDoc` array, traverse it and print the information needed. Things get a bit more complicated when traversing a full fledged Java source code. As figure 3.1 page 20 shows no less than 18 interfaces are needed understood. The figure includes some of the most used methods, many more exists in the framework. Although the many classes, the hardest part was to find the proper method names and understand the class hierarchy (which is not obvious when looking only at the API).

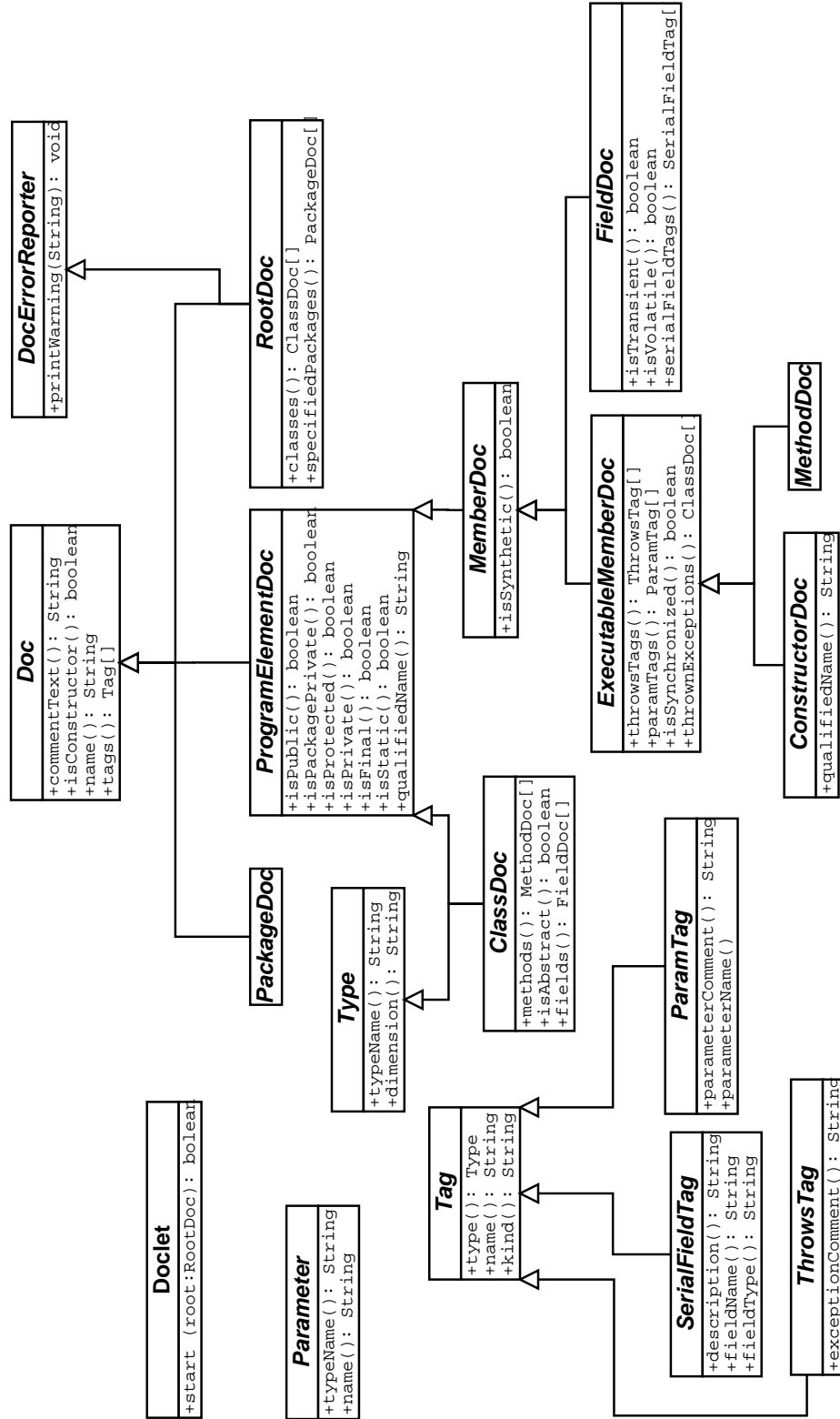


Figure 3.1: The hierarchy of the doclet framework.

The design

Although the figure resembles table 3.1 page 24 the table was developed before we learnt about the hierarchy in figure 3.1. The design of the JavaMiner doclets also conform the structure of table 3.1 rather than the figure. The implementation followed the recipe which also were used for the database design. For every nonterminal a method was created handling only that part of the whole.

Comparing $\langle \text{class} \rangle$ from table 3.1 with the implementation we see the code strictly follows the definition in the table.

```
class  →  [⟨comment⟩] [InPackage] Superclass [(implements interfaces)]
         [abstract] ⟨modifiers⟩ Name DisplayName {⟨constructor⟩}
         {⟨method⟩}{⟨field⟩}[⟨inherited methods⟩ ]

static void CLASS(ClassDoc cd)
{
  StringBuffer sb = new StringBuffer();

  sb.append(COMMENT(cd, cd.qualifiedName())      + ","); // <comment>
  sb.append("'" + cd.containingPackage()         + "',"); // inpackage
  sb.append("'" + cd.superclass()               + "',"); // Superclass
  sb.append(MODIFIERS(cd, cd.isAbstract(), false) + ","); // <modifiers>
  sb.append("'" + cd.qualifiedName()            + "',"); // name
  sb.append("'" + cd.name()                     + "',"); // DisplayName
  sb.append(fetchCode(cd.qualifiedName()));     // sourcecode

  // Insert data into table "Class" in database
  dbconn.insert("Class", sb.toString());

  // generate table Implements Interfaces
  IMPLEMENTSINTERFACES(cd, cd.qualifiedName());

  // Generate CONSTRUCTOR entries
  ConstructorDoc[] cod = cd.constructors();
  for(int i = 0; i < cod.length; i++)
    CONSTRUCTOR(cod[i], cd.qualifiedName());

  // Generate METHOD entries
  MethodDoc[] md = cd.methods();
  for(int i = 0; i < md.length; i++)
    METHOD(md[i], cd.qualifiedName());

  // Generate FIELD entries
  FieldDoc[] fd = cd.fields();
  for(int i = 0; i < fd.length; i++)
    FIELD(fd[i], cd.qualifiedName());

  // Generate INHERITED CLASSES entries
  INHERITEDCLASSES(cd);
}
```

The two are identical. Notice how the primary key for Class is passed on to the other methods in form of `cd.qualifiedName()` Not only did the EBNF notation give a terse overview of what comments in Java was, it also gave a recipe for designing the database and implementing the JavaMiner doclet.

It should be noted however, that method `COMMENT()` was a little more complicated to write, as we needed to parse the content of the tags, splitting into parts separated by the separator characters.

The method `insert()` come from the class `DbConnection` which uses the JDBC as described in section 1.1 page 3.

3.1 Usage

Using the JavaMiner doclet is like using the Javadoc tool except that one more parameter needs be specified (which is to use the JavaMiner doclet to take care of the outputting). To include the file `B.java` in the package `A` into the EJD one has to write:

```
javadoc -private -doclet JavaMiner -docletpath . A\B.java
```

provided the doclet and the package resides in the directory where the call is being made from. Otherwise the `docletpath` and `sourcepath` parameters can be used.

Bibliography

Gamma, E., Helm, R., Johnson, R., Vlissides, J. and Booch, G. (1995). *Design Patterns*, first edn, Addison-Wesley Pub Co. ISBN: 0201633612.

Hall, M. (2000). *Core Servlets and JavaServer Pages*, first edn, Prentice Hall PTR/Sun Microsystems Press, USA. ISBN:0130893404.

Hougland, D. and Tavistock, A. (2001). *Core JSP*, first edn, Prentice Hall, Upper Saddle River, USA. ISBN:0-13-088248-8.

Pelegri-Llopart, E. and Cable, L. (1999). *Java Server PagesTM Specification*, 1.1 edn, Sun Microsystems.