



LIVE SOFTWARE, INC.

# JRun Scripting Guide

Version 2.3

LIVE SOFTWARE, INC.

# JRun Scripting Guide

---

Version 2.3

© 1999 Live Software, Inc.  
20245 Stevens Creek Blvd, Suite 100  
Cupertino, CA 95014  
[info@livesoftware.com](mailto:info@livesoftware.com)  
<http://www.livesoftware.com>

<a href="#">What is the JRun Scripting Toolkit?</a>	2
<a href="#">JRun Scripting Toolkit Hardware Requirements</a>	4
<a href="#">Add-on Development Tools</a>	4
<a href="#">JRun Scripting Toolkit Directory Structure</a>	5
<a href="#">Introduction</a>	7
<a href="#">What are Server-Side Includes (<i>shtml</i> files)?</a>	7
<a href="#">What are Presentation Templates (<i>thtml</i> files)?</a>	8
<a href="#">What is Page Compilation (<i>jsp</i> files)?</a>	8
<a href="#">The SERVLET Tag</a>	9
<a href="#">The INCLUDE Tag</a>	10
<a href="#">The default.template file</a>	11
<a href="#">The default.definitions file</a>	12
<a href="#">The default.template Scope Rules</a>	13
<a href="#">The default.definitions Scope Rules</a>	14
<a href="#">Introduction to JSP Scripting</a>	15
<a href="#">Creating Your First Page</a>	15
<a href="#">From HTML to Java to Servlets</a>	17
<a href="#">Mixing It Up With Multiple HTML/Java Blocks</a>	18
<a href="#">Declaring Variables</a>	18
<a href="#">Conditional HTML</a>	19
<a href="#">Using Expressions</a>	20
<a href="#">Calling Another Page</a>	20
<a href="#">Including Contents of Non-jsp Pages</a>	21
<a href="#">Including Contents of Other jsp Pages</a>	22
<a href="#">Using the global.jsa File</a>	23
<a href="#">What Are Taglets?</a>	24
<a href="#">Loading and Using Custom Taglets</a>	25
<a href="#">Properties</a>	30
<a href="#">Optimizing Page Compilation for Deployment</a>	32
<a href="#">Examples for Writing JSPTaglets</a>	35
<a href="#">Examples for Writing SSITaglets</a>	42
<a href="#">Extensions to Servlet API</a>	46
<a href="#">Compatibility</a>	48
<a href="#">Available objects</a>	48
<a href="#">Syntax</a>	49
<a href="#">The Directive Taglet</a>	49
<a href="#">The Class Wide Declaration Taglet</a>	50
<a href="#">The Scriptlet Taglet</a>	51
<a href="#">The Expression Taglets</a>	51
<a href="#">The JSP BEAN Taglet</a>	52
<a href="#">From ASP to JSP</a>	55
<a href="#">Application Object</a>	55
<a href="#">Request/request Object</a>	57
<a href="#">Response/response Object</a>	59
<a href="#">Server Object</a>	61
<a href="#">Session Object</a>	61

---




## *Introduction to the JRun Scripting Toolkit*

### What is the JRun Scripting Toolkit?

---

#### ICON KEY

---

	Web site
	Example
	Important Note

---

JRun Scripting Toolkit, hereafter referred to as JST, is a collection of components that allows you to develop and deploy server-side scripting for dynamic web content. As a Java<sup>1</sup> solution, JST leverages the power of the Java programming language. Java is a platform-independent programming language developed by Sun Microsystems. A program or application written in Java can be run on many different types of machines, including Windows, Macintosh, and Unix-based machines.

JST is an extension to Live Software's JRun Servlet Engine (JRun). JRun is used to extend your current web server to include additional server-side Java functionality. JST provides the following three main components:

JRun Server Pages (JSP):

Live Software's implementation and extension of Sun's Java Server Pages 0.92 specification.

Dynamic Taglets:

A convenient tool that gives servlet writers the freedom to seamlessly define a tag that maps to a specific servlet.

Presentation Templates:

An easy way to apply a consistent look and feel to html applications.

Each of these three components is discussed in detail in this document.

---

<sup>1</sup> Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

## JRun Scripting Toolkit Features

JST fully supports all of the capabilities of page compilation as defined by the Java Web Server and fully supports the current Java Server Pages specifications. Following is a list of the features of JST:

- JSP Taglet Support
- 'Global.jsa' Support
- Full support for <BEAN> tag
- 100% JWS page compilation compatible
- 100% Java Server Pages compatible
- Contains nearly all ASP-like objects (i.e. Request, Session, Cookie, etc.)
- Supports extending of other JSP pages for true object oriented page design
- Support recursive dependent file compiling
- Includes JSPHttp, JSPFile, and JSPMail JavaBean components
- Includes JSPDBConnection JavaBean for full pooled-connection, database access via JSP
- Includes a full license to InstantDB, a simple yet powerful all-Java relational database
- Presentation Templates
- Support for all Java Virtual Machines and Java Compilers

These features will be explained in more detail throughout the remainder of this document.

JST also comes with many examples and samples to help get you started with developing server-side scripting and Java servlets. In addition, you can visit Live Software's web site at



<http://www.livesoftware.com/>

for more examples and tutorials as they become available.

## *Before You Start*

### **JRun Scripting Toolkit Hardware Requirements**

Because JST is written in Java, it can run on any type of computer platform, including Windows, Macintosh, and Unix. No specific hardware is required to run JST. It is recommended that you have at least 32MB of RAM. The more memory the better your system will perform.

### **Add-on Development Tools**

Several different development tools may be used in conjunction with JST in order to provide additional functionality. These add-on development tools include:

Live Software's Servlet Debugger:

ServletDebugger allows you to thoroughly test and debug your servlets directly within your favorite development environment. It includes an integrated version of Live Software's ServletKiller, a stress test tool that allows you to send many repeated requests to your servlet to ensure its stability.

## *Installing JRun Scripting Toolkit*

### **JRun Scripting Toolkit Directory Structure**

JRun Scripting Toolkit now is part of JRun 2.3. The JRun 2.3 installation wizard should automatically configure JRun to use JST.

# *Server-Side Scripting*

## *Server-Side Documents*

### Introduction

This chapter introduces the concept of server-side documents and touches briefly on the types of server-side documents currently supported by JRun Servlet Engine (JSE) within JST.

Server-side documents are web documents containing instructions that need to be pre-processed on the server end. When a server-side document is requested by a client browser, the web server may pre-process the document before serving it to the client. Server-side documents usually contain instructions that are not normally understood by the client browser, and are interpreted on the server end. Results of the pre-processed documents are eventually sent as readable HTML (HyperText Markup Language) to the client browser. Such server-side instructions may contain a whole set of tags to perform actions and add logic to a web document. Server side documents are a way to apply dynamic content and logic to HTML-based pages, independent of the browser type.

Server-side documents are usually marked with a file extension other than *html*, such as *asp*, *cfm*, *shtml*, *thtml*, or *jsp*. This is a way to let the server know what particular documents need to be pre-processed before being sent out. The file extension also lets the server know what component is responsible for processing the server-side document. Under the JST environment, these components are more than likely Java Servlets that run within JSE. For more information on Java Servlets, see the section *Using Java Servlets*.

The following sections describe the type of server-side documents that are handled by JSE, in particular *shtml*, *thtml*, and *jsp* documents. Users are free to define and map a limitless array of file extensions to any handler (servlet) that they can conjure up.

### What are Server-Side Includes (*shtml* files)?

Server-Side Includes (SSI) is one of many ways that JST provides dynamic content to a web page. Server-Side Includes (SSI) are tags defined to be used and processed on the server side rather than by the client browser. Files with a *shtml* extension are text files containing a mixture of SSI tags and HTML. Two tags are defined to be used with *shtml* files. They are the `SERVLET` tag and the `INCLUDE` tag. These tags are used to call a servlet or include contents from another file into the *shtml* file respectively. The component responsible for handling *shtml* files is a servlet itself that runs in the JSE.

See the section “Using Server Side Includes” to become familiar with using and taking advantage of server-side includes.

## What are Presentation Templates (*thtml* files)?

Presentation Templates allow web developers to seamlessly apply a constant look and feel to their HTML applications. Template files have the extension of *thtml*. These files are any HTML-based page with nothing more than just a head and body. When a *thtml* file is requested, the component (servlet) responsible for handling the *thtml* file replaces certain tags within a common template file with the text within the head and body tags contained in the *thtml* file. The name of the template file is “default.template.” The *default.template* file defines and contains the common look and feel for all *thtml* files that use it.

See the section “Using Presentation Templates” to become familiar with using and taking advantage of presentation templates.

## What is Page Compilation (*jsp* files)?

Another component of JRun Scripting Toolkit provides the ability to create and run dynamic, interactive, high-performance, platform-independent web applications through server-side scripting. These web-based applications are created as a collection of files that contain HTML and embedded Java code. These files are compiled on-the-fly into Java Servlets and have the file extension of *jsp*. The component responsible for handling *jsp* files is a servlet itself that runs in the JSE and is referred to as the “Page Compilation Engine.”

Page Compilation documents can contain any combination of the following:

- Text
- HTML tags
- Script commands (A script command instructs your computer to do something, such as assign a value to a variable.)

See the section “Using Page Compilation” to become familiar with using and taking advantage of page compilation.

## Using Server Side Includes

This chapter provides the general syntax of SSITaglets used within *shtml* files, as well as a tutorial on how to use and take advantage of these powerful tags.

### The SERVLET Tag

The general syntax for calling a servlet through the SERVLET tag follows:

```
<servlet name="aliasname" code="classname">
<param name="paramname1" value="paramvalue1">
<param name="paramname1" value="paramvalue1">
<param name="paramname1" value="paramvalue1">
...
</servlet>
```

You would use either the *name* attribute or *code* attribute, but generally not both. The `<PARAM>` tags are optional. They allow you to pass extra parameters to your servlets. The servlet's init parameters are fetched from the alias information using the JRun Admin tool. A Java Servlet is a server-side component analogous to an applet, but on the server end. For more information on Java Servlets see the section *Using Java Servlets*.

A good way to experiment with the SERVLET tag is to use it with SnoopServlet. Since SnoopServlet displays anything passed to it, you can see and change values displayed simply by changing parameters specified in the servlet tag. Follow these steps to get started.

1. Open a new text file. In the text file, type the following:

```
<servlet name=SnoopServlet>
<param name="greeting" value="Hello World">
<param name="anotherParam" value="aParamValue">
</servlet>
```

2. Save the text file as ***greeting.shtml*** into your web server's document root. So, if your web server's document root was, `c:\inetpub\wwwroot`, you would save your file as:

`c:\inetpub\wwwroot\greeting.shtml.`

3. Request the *greeting.shtml* from your web browser. The URL should be something like

[http://<your\\_host\\_machine>/greeting.shtml](http://<your_host_machine>/greeting.shtml).

Feel free to modify the *greeting.shtml* file by changing, adding, or removing parameter tags. Once you have saved the changes in your *greeting.shtml* file, the results upon subsequent requests through your web browser will reflect the changes.

## The INCLUDE Tag

The INCLUDE tag can be used within a *shtml* file to include contents of another file. The general example for using the INCLUDE tag follows:

```
<!--#include virtual|file="filename"-->
```

Where you must type either *virtual* or *file*, which are keywords that indicate the type of path you are using to include the file, and *filename* is the path and file name of the file you want to include.

Included files do not require a special file-name extension.

## Using the Virtual Keyword

Use the *Virtual* keyword to indicate a path beginning with a *virtual directory*. For example, if a file named *footer.inc* resides in a virtual directory named */myapp*, the following line would insert the contents of *footer.inc* into the file containing the line:

```
<!--#include virtual="myapp/footer.inc" →
```

## Using the File Keyword

Use the *File* keyword to indicate a **relative** path. A relative path begins with the directory that contains the including file. For example, if you have a file in the directory *myapp*, and the file *header1.html* is in *myapp\headers*, the following line would insert *header1.inc* in your file:

```
<!--# include file="headers/header1.inc"→
```

Note that the path to the included file, *headers/header1.inc*, is relative to the including file – if the script containing this Include statement is not in the directory */myapp*, the statement would not work.

## Using Presentation Templates

Presentation Templates allow web developers to seamlessly apply a constant look and feel to their html applications. Presentation Templates incorporates the use of two other files; the required file *default.template*, and the optional file *default.definitions*.

### The default.template file

The *default.template* is the file used to apply a common look and feel to a *thtml* file. The *default.template* file contains a mixture of HTML and special SUBST tags (*substitution tags*). The SUBST tags are markers that are replaced by the text assigned to the name that the SUBST tag represents. For example, suppose the *thtml* file *greeting.thtml* contained the following HEAD and BODY tags:

*greeting.thtml*:

```
<head>
<title>Greetings from greeting.thtml</title>
</head>

<body>
Hello World from greeting.thtml
</body>
```

and the *default.template* file contained the following:

*default.template*:

```
<html>

<head>
<subst data="head"/>
</head >

<body>
A greeting from the requested document: <subst data="body"/> <br>
</body>

</html>
```

The browser would get the resulting document:

*resulting document*:

```
<html>
```

```

<head>
<title>Greetings from greeting.shtml</title>
</head >

<body>
A greeting from the requested document: Hello World from greeting.shtml
<br>
</body>

</html>

```

As you see, the occurrences of the `<subst data="head">` and `<subst data="body">` tags defined in the *default.template* file have been replaced with the contents of the HEAD and BODY tags in the *thtml* file respectively.

The general syntax for the substitution tag follows:

```

<subst data="[name]"/>

or

<subst data="[name]"></subst>

```

Where *name* is the name of the data to substituted in. The value of the given name would replace its corresponding substitution tag in the resulting output to the client.

As you may know, in order for a *thtml* file to be served properly, a *default.template* must be accessible. The rules for accessibility of *default.template* files that apply to certain *thtml* files is described in a later section.

Template developers are not restricted to only applying substitutions for the HEAD and BODY tags. Values for additional named substitutions can be defined and used within *default.template* files. This is specified in the *default.definitions* file. See the next section for a description of the *default.definitions* file.

## The default.definitions file

The *default.definitions* file is a property file containing name/value pairs. The property file is used to define values for names that may be used for substitution in the SUBST tag. The following example illustrates how one would assign name/value pairs in the *default.definitions* file:

*default.definitions* file:

```

MyThought=<P>Missing sweet San Diego</P>
TodaysSaying=<B>Anger is a gift...(Rage Against The Machine)</B><BR>
MyLineBreaks=<PRE>A line break follows<BR>\nHeres a new line</PRE><BR>

```

Each name/value pair must exist on the same line unless the newline is escaped out using the backslash “\” character, i.e. the backslash must proceed the newline if values take up more than one line. Newlines that must exist in the value themselves can be specified with the “\n”, as in the third name/value pair. Basically, the rules for specifying name value pairs are the same rules that apply in the definition of the `java.util.Properties` class.

Once name/value pairs have been set in the *default.properties* file. The *default.template* file may use these within the SUBST tags. For example:

*default.template* file:

```
<html>

<head>
<subst data="head"/>
</head >

<body>
A greeting from the requested document: <subst data="body"/><br>
Here's a thought: <subst data="MyThought"/>
Here's a saying: <subst data="TodaysSaying"/>
Line Breaks: <subst data="MyLineBreaks"/>
</body>

</html>
```

If *greeting.shtml* was requested, and the above *default.template* and *default.definitions* were available, the browser would get the resulting document.

*resulting document*:

```
<html>

<head>
<title>Greetings from greeting.shtml</title>
</head >

<body>
A greeting from the requested document: Hello World from greeting.shtml <br>
Here's a thought: <P>Missing sweet San Diego</P>
Here's a saying: <B>Anger is a gift...(Rage Against The Machine)</B><BR>
Line Breaks: <PRE>A line break follows<BR>
Heres a new line</PRE><BR>
</body>

</html>
```

If the *default.template* file contained SUBST tags for which there were no name/value pairs available, then the SUBST tag would be replaced with a blank string in the resulting page. The *default.definitions* file is not required when requesting a *.shtml* file. Based on the scoping rules defined for the template servlet, more than one *default.definitions* file may exist for a *default.template* file. The next section makes this clear.

## The default.template Scope Rules

The available scope of the *default.template* applies to all *.shtml* files existing in the same directory and its subdirectories. If more than one *default.template* file exists in a directory hierarchy, the *.shtml* file will be used with the *default.template* closest to it and up the directory hierarchy. So, if the following three files exist,

```
c:/foo/default.template
c:/foo/bar/doo/default.template
c:/foo/bar/a.shtml
```

then the Template servlet would use `c:/foo/default.template` with `c:/foo/bar/a.shtml` to produce the resulting page to the client.

## The default.definitions Scope Rules

The name/value pairs defined in the *default.definitions* file apply to all *.shtml* files in the same directory and its subdirectories. If more than one *default.definitions* exists in a directory hierarchy, the *.shtml* file will use the *default.definitions* file closest to it and up the directory hierarchy. The name/value pairs of default.definitions files are cumulative relative to the location of the *.shtml* file. This means that name/value pairs can be added to the current set of name/value pairs, but additions are only available to *.shtml* files in the same directory and its subdirectories.

The name/value pairs defined by one default.definitions file can be overwritten by name/value pairs in a default.definitions file in its subdirectory. The overwriting of name/value pairs only affect *.shtml* files in the same directory and in subdirectories of where the default.definitions file that does the overwriting reside. So, if **drinks=cola** was defined in a default.definitions file in `c:/foo/`, and later down the directory structure `c:/foo/bar/` another default.definitions file defined **drinks=milk**, only *.shtml* files under `c:/foo/bar/` and its subdirectories would know **drinks=milk**. The *.shtml* files in `c:/foo/` would still have **drinks=cola**.

## *Using JRun Server Pages (JSP)*

This chapter is a quick introduction to Live Software's Page Compilation Engine, Live Software's advanced implementation of Sun's Java Server Pages specification. See the Developer's Guide section to get a more in-depth description of features and functionality of Live Software's advanced implementation of JSP.

### Introduction to JSP Scripting

A script is a series of script commands. A script can, for example:

- Assign a value to a **variable**. A **variable** is a named storage location that can contain data, such as a value.
- Instruct the Web server to send something, such as the value of a variable, to a browser. An instruction that sends a value to a browser is an **output expression**.
- Combine commands into **procedures**. A procedure is a named sequence of commands and statements that acts as a unit

Page compilation does not define a scripting language, but rather allows you to write documents with Java code and HTML, therefore enabling you to fully use the readily defined Java Programming Language.

The page compilation engine processes the special documents to generate Java Servlets, one of the latest server-side application technologies. Servlets have been popularized due to their high performance over conventional CGI applications. A benefit of page compilation is that you do not have to worry about learning to write actual Java servlets, the page compilation engine virtually hides this complexity from you. This allows first time users that have only a working knowledge of HTML to get started writing applications using page compilation scripting. However, for writers who demand the flexibility of Java Servlets, they can still make full use of the SERVLET API from within a page compilation document.

Because generation of HTML is all handled on the server end, you need not worry whether a web browser can process your page. In addition, response time of servlets and deployed page compilation documents over conventional server-side scripting is improved by an order of magnitude.

### Creating Your First Page

Let's start your first page compilation file with the ubiquitous "hello world" document:

1. Create a new text file and put the following lines in your file:

```
<html>

<head>
<title>Greetings</title>
</head>

<body>
<h1>Hello World!</h1>
</body>

</html>
```

2. Save the file into the Web server's document root, and name it *greeting.jsp*. Files with the *.jsp* extension are recognized, and processed by the page compilation engine. If your Web server's document root was `c:\inetpub\wwwroot`, you would save the file as `c:\inetpub\wwwroot\greeting.jsp`.
3. The next step is to ask for the document through your browser using the appropriate URL, you would type something like the following

<http://<your-host>/greeting.jsp>.

As you have noticed, the document you just created was just a simple HTML document with no Java code. What made this document special was its file extension of *.jsp*. The page compilation servlet recognized the extension of the file and compiled it into a Java servlet. However, this is totally transparent to the person requesting the page.

The first example was intended to illustrate that HTML is naturally handled correctly by the page compilation servlet. We will now modify *greeting.jsp* to do some simple looping using embedded Java code. Re-edit your *greeting.jsp* file to look like the following:

```
<html>
<head>
<title>Greetings</title>
</head>
<body>

<%
for (int i=0; i < 5; i++) out.println("<h1>Hello World!</h1>");
%>

</body>
</html>
```

Save the file and request it again through your browser. The page compilation engine recognizes Java code as being in between the `<% %>` tags. It uses anything in between these tags and executes it as Java. The above loop prints out "Hello World!" five times to the browser through the "out" object. The *out* object is one of many objects available within a page compilation document. For a description of all objects readily available within a page compilation document, see the section "Page Compilation Object Reference."

## From HTML to Java to Servlets

The process of interpreting a page compilation file into a Java servlet class file is called **page compilation**. When a page compilation file is first requested, the page compile engine is invoked to parse the file into a Java source file. The Java source file is then compiled into a servlet class file. The servlet class file is then loaded and run. The output of the servlet is sent to the client-browser. A *jsp* file is parsed and compiled if it is newer than its class file, or if its class file does not exist. Subsequent requests of an unmodified page compilation file will result in the direct invocation of its corresponding servlet, thus skipping its parse and compilation phase.

The following is an expanded explanation of the steps that the page compile engine performs when a page compilation file is requested:

1. The page compilation file is parsed.

This step performs the task of creating a Java servlet source file based on the contents of a page compilation file. Parsing of the file is made possible by tag components called **JSPTaglets**. **JSPTaglets** define tag-based syntax into a page compilation file such as the scriptlet taglet `<% %>` in the above example. For more information of all defined page compilation taglets, see the section “Page Compilation Scripting Guide.”

2. The Java servlet source file is compiled into a servlet class file from the results of parsing the page compilation file.

A Java servlet source file is a result of the combined work done by all JSPTaglets encountered while parsing a page compilation document. Once the Java servlet source file has been generated, it is then compiled into a Java servlet class file.

3. The servlet class file is loaded into the JRun Servlet Engine (JSE)

4. The servlet is run.

At this point, output going to the client no longer contains JSPTaglets. All taglets have been processed in step one.

Based on the content-type of the servlet, output of the running servlet may be post-processed before being sent to the browser.

By default, the output of a compiled page is post-processed. The post-processing of a compiled page is enabled by specifying its default mime type to be **jsp-internal/taglet**. If this mime type is set, the output will be sent through the **JSPSSIFilter**. The JSPSSIFilter is responsible for handling the SSITaglets `<servlet>` taglet, `<!--#include>` tag, and possibly user defined SSITaglets or DynamicTaglets. Read the “Taglets” chapter for an explanation of taglets.

Upon subsequent requests, if the file has not been modified since its last compilation, the page compile engine only performs step 4 on the servlet corresponding to the requested page-compilation file.

## Mixing It Up With Multiple HTML/Java Blocks

When creating a page compilation file, you can feel comfortable mixing blocks of Java code and HTML. The page compilation engine handles everything as they come along, therefore completely keeping the output of your page compilation file in order. The following example is how one would mix up HTML and Java blocks:

```
<html>
<head><title>Mix me up</title></head>
<body>

<p>

<%
out.println ("This block is generated by java");
%>

<h1> a list generated by java</h1>
<ul>

<%
for (int i = 1; i < 10; i++) {
out.println ("<li>" + i);
}
%>

</ul>

</body>
</html>
```

## Declaring Variables

As with any scripting functionality, page compilation allows Java-style variable declarations. Variables can be defined once, and re-assigned between Java blocks. The following example illustrates this.

```
<html>
<head><title>Using variables</title></head>
<body>

<p>

<% int myVar = 5; %>

<b> <% out.println ("Value of myVar: "+myVar); %> </b>

<p>

<%
myVar = 2;

out.println("Value of myVar again: "+myVar);
%>

</body>
</html>
```

The variable *myVar* is **only** accessible within the page compilation file it was declared in, and is not accessible anywhere else. Although the page compilation file may have separated Java in different `<%%>` tags, the page compilation servlet would still view everything under the same scope, exactly as if all the code was in one function. This means the variable may be re-assigned within the page, but the name itself may not be declared more than once. For this reason, the following example illustrates the **incorrect** use of a variable.

```
<html>
<head><title>Using variables</title></head>
<body>

<p>

<% int myVar = 5; %>

<b> <% out.println ("Value of myVar: "+myVar); %> </b>

<p>

<%
int myVar = 2; //error here

out.println("Value of myVar again: "+myVar);
%>

</body>
</html>
```

The above example declared *myVar* as an `int` in the first Java block, and then redeclared it as an `int` again in the third Java block. This incorrect use would cause a compile error by the page compilation engine since variables cannot be redeclared in the same scope.

## Conditional HTML

The following example illustrates the use of conditional statements used within a page compilation file:

```
<html>
<head><title>Account Balance</title></head>
<body>

<p>

<% double accountBalance =1.00; %>

Your Current Balance: <% out.println( accountBalance ); %> <br>

<% if(accountBalance <= 1.00) { %>
    Get a Job <br>
<% } %>

</body>
</html>
```

The above example simply prints out the value of the variable *accountBalance*, and if *accountBalance* is less than or equal to one dollar, the next statement suggests that the user “get a job.” You can play with the statement by increasing or decreasing the *accountBalance*.

Also notice that the conditional statement uses another block `<% } %>` to close the if-statement – this is to fit in the HTML “`<b> Get a job. </b>`” to be displayed if the condition was true. If you prefer the condition to stay in one Java block, then the following would give you the same results:

```
<% if(accountBalance <= 1.00) out.println("<b> Get a job.</b>"); %>
```

As you can see, the statement “`<b> Get a job. </b>`” is explicitly sent through the stream, rather than being specified within the document as HTML. By using either method, the browser gets and displays the message the same way.

## Using Expressions

So far, previous examples have illustrated the use of displaying variable values by using the `out.println()` method. However, there will be situations where this may clutter your document whenever you would like to display these values. The expression taglet allows you to display evaluated expressions without having to use `out.println()`, thus keeping your page compilation document more readable. The following example illustrates the use of the expression taglet:

```
<html>
<head><title>Account Balance</title></head>
<body>

<p>

<% double accountBalance =1.00; %>

Your Current Balance: <%= accountBalance ) %> <br>

<% if(accountBalance <= 1.00) { %>
    Get a job <br>
<% } %>

</body>
</html>
```

The above example is a modification of the previous conditional example. As you can see in the third Java block, the `<% out.println(accountBalance); %>` has been replaced by `<%= accountBalance %>`. This taglet allows you to display the value of an expression without having to use the `out.println()`.

## Calling Another Page

The `CallPage()` method is a defined method for page compilation files to allow the page compilation writer to call another page compilation document. The method definition for `CallPage` follows:

```
CallPage(String path)
```

where *path* is the virtual path to the page being called.

The following example illustrates how one *jsp* document would call another *jsp* document. Furthermore, it illustrates how one would pass parameters, to the *jsp* document that is called:



Create a text file, and save it into your web document root as *a.jsp*. Type the following into *a.jsp*.

```
<%
    request.setAttribute("Greeting", "Hello World");
    CallPage("b.jsp");
%>
```

The above example uses the *request* object to set a name “Greeting” with a value of “Hello World.” The `CallPage()` method is used to call *b.jsp* from within *a.jsp*. The message will be retrieved from the request object in *b.jsp*. In order for *b.jsp* to get the message, it has to be set in *a.jsp* before *b.jsp* can be called.

Now let’s make the file *b.jsp*. This file should also be saved into your web server’s document root. Type the following into *b.jsp*:

```
Hello from b the greeting is: <%= (String)request.getAttribute("Greeting") %>
```

A very trivial example, but it illustrates passing data from one page to another. Better yet, the value of your named attribute is not restricted to just Strings, you can just as easily pass in any named object you wish. For this reason, Java requires that you at least specify the **type** of what is being retrieved. The above example does this by “casting” the retrieved attribute to String. In Java, you do this by putting the object type in between parenthesis in front of the request’s method call to retrieve it. The following illustrates this:

```
(String)request.getAttribute("Greeting")
```

By using `CallPage` in conjunction with the BEAN taglet, form data can very easily be passed from one *jsp* page to another. For more detailed descriptions of all taglets that JRun Scripting Toolkit has to offer, see the *Page Compilation Scripting Guide* chapter under **Section VI: Developer’s Guide**.

## Including Contents of Non-jsp Pages

The SSI Include tag is a mechanism you can use to insert contents from other files prior to being sent to the client browser. The SSI Include Tag is considered a SSITaglet. This taglet is evaluated from within the SSIFilter’s parser. A JSP document’s output with a content type of **jrun-internal/taglet** is sent throughout the SSIFilter for post-processing before being sent to the browser. Any occurrence of the SSI Include Taglet found in the filtered data is replaced with the contents of the file being included. The syntax of the SSI Include Taglet follows.

```
<!--#INCLUDE VIRTUAL|FILE="filename"-->
```

You must type either *VIRTUAL* or *FILE*, which are keywords that indicate the type of path you are using to include the file, and *filename* is the path and file name of the file you want to include.

Included files do not require a special file-name extension.

## Using the Virtual Keyword

Use the **Virtual** keyword to indicate a path beginning with a **virtual** directory. For example, if a file named *footer.inc* resides in a virtual directory named */myapp*, the following line would insert the contents of *footer.inc* into the file containing the line:

```
<!--#INCLUDE VIRTUAL="/myapp/footer.inc"-->
```

## Using the File Keyword

Use the **File** keyword to indicate a **relative** path. A relative path begins with the directory that contains the including file. For example, if you have a file in the directory *myapp*, and the file *header1.html* is in *myapp\headers*, the following line would insert *header1.inc* in your file:

```
<!--#INCLUDE FILE="headers/header1.inc"-->
```

Note that the path to the included file, *headers/header1.inc*, is relative to the including file; if the script containing this Include statement is not in the directory */myapp*, the statement would not work.

Because the SSI Include Taglet is a post-processing taglet, *jsp* documents whose output is being sent through SSIFilter can use the SSI Include Taglet with embedded *jsp* expression tag `<%= %>`. The following example illustrates the use of the SSI Include Taglet from within a *jsp* document.

```
<% myIncludeFile="headers/header1.inc"; %>
```

...

```
<!--#include="<%=myIncludeFile%>" -->
```

The above example is legal within a *jsp* document. Because the contents of the file are sent directly to the client, server-side files such as *jsp* files that need to be interpreted will be sent as is.

## Including Contents of Other jsp Pages

JSP documents that need to “include” contents of other *jsp* documents, and want the included *jsp* files to be pre-processed along with the including *jsp* page, should use the following JSPTaglets. For more information on these JSPTaglets and others, see the “Page Compilation Scripting Guide” chapter.

```
<%@ include=fileName%>
```

or

```
<%@ vinclude=fileName%>.
```

## Using the global.jsa File

Each *jsp* application can have one *global.jsa* file. (The file name extension *jsa* stands for JRun Server Application) This file must be stored in the root directory of the *jsp* application. The page compilation engine reads a *global.jsa* file when:

- The Web server receives the first post-startup request for any *jsp* file in a given application; that is, after the Web server starts, the first request for any *jsp* file in an application causes page compilation engine to read the *global.jsa* file for that application.
- A user who does not have a session requests a *jsp* file in an application.

You can include the following in a *global.jsa* file: application-start events, session-start events, application-end events, and session-end events.

The following example illustrates how to specify each event from within the *global.jsa* file.



```
<script runat="server" event="Session_OnStart">
Response.write("Hello This is a new Session<BR>");
System.out.println("Session_OnStart called for: "+Session.getId());
</script>

<script runat="server" event="Session_OnEnd">
System.out.println("Session_OnEnd called for: "+Session.getId());
</script>

<script runat="server" event="Application_OnStart">
System.out.println("Application_OnStart called");
Application.setAttribute("greeting", "hi mom");
</script>

<script runat="server" event="Application_OnEnd">
System.out.println("Application_OnEnd called");
Application.removeAttribute("greeting");
</script>
```

There are **restrictions** as to what objects are available within each event handling method.

- **Session\_OnStart** can access any object a “normal” JSP file can
- **Session\_OnEnd** can only access the **Application**, **Session**, and **Server** objects. The **Server** object cannot call the **MapPath()** method. The **CallPage()** method is also not available within **Session\_OnEnd**.
- **Application\_OnStart** can only access the **Application** and **Server** objects. The **CallPage()** method is not available within **Application\_OnStart**.
- **Application\_OnEnd** can only access the **Application** and **Server** objects. The **CallPage()** method is not available within **Application\_OnEnd**.

## Taglets

JRun Scripting Toolkit' Taglets present developers creating server-side documents using JSP with one more level of separation of presentation from application logic. This distinction may also reflect a company's logistics in distributing web application development workload among various employees. For example, HTML programmers can concentrate on tag-based documents for presentation, and Java programmers can concentrate on application logic. The JRun Scripting Toolkit Taglet API would be used by Java programmers to define helpful tags for the HTML programmers. Simple communication between the HTML programmer and the Java programmer can streamline development time by keeping the HTML programmer and Java programmer in a position where they can do what they do best.

This chapter is a quick introduction for HTML and Java programmers who are learning about Live Software's Taglets. For an in-depth look on using and developing Taglets in the JRun Scripting Toolkit environment, see the appropriate chapters in Section VI, "Developer's Guide."

### What Are Taglets?

Taglets are user defined server-side tags that are implemented in Java using Live Software's Taglet API. They provide flexibility for application developers to define and implement their own tags that can be used within *shtml* and/or *jsp* files. All of the defined tags defined for page compilation are taglets called *JSPTaglets*. Taglets such as the SERVLET tag and the INCLUDE tag used in *shtml* files are also Taglets called *SSITaglets*. Taglets that provide a one-to-one mapping to call a servlet are called *DynamicTaglets*.

Taglets are broken down into three main types: JSPTaglet, SSITaglet, and DynamicTaglets. JSPTaglets and SSITaglets are also called *codetaglets*. The taglets are split up based on their use and responsibilities in the JRun Scripting Toolkit system.

The following explains the distinction between the three types of taglets.

### What is a JSPTaglet?

*JSPTaglets* are used by the page compile engine when a page compilation document needs to be parsed and recompiled. Since it is responsible for adding any Java code to the servlet source being built, the JSPTaglet is considered a **pre-processing** taglet. Taglet writers who need a way to add Java code to a compiled page's generated servlet can extend the JSPTaglet. JSPTaglets are implemented via the Taglet API. The JSPTaglet provides taglet writers with an interface to

the servlet source being built via the **ServletBuilder** interface. This gives Java programmers the ability to add consistent logic to a servlet whenever their JSPTaglet is encountered by the page compilation engine's parser. All taglets defined for page compilation are internally defined JSPTaglets. User defined JSPTaglets are loaded by the page compilation engine through the **codetaglet.properties** file. The **codetaglet.properties** file is explained in the next section *Loading Custom Taglets*, and for examples on creating custom JSPTaglets see the chapter "Developing Custom Taglets."

## What is a SSITaglet?

*SSITaglets* are taglet components independent of the page compilation engine. SSITaglets are used by the parser in the SSIFilter to add further logic to web based documents. Output of page compilation documents are sent through the SSIFilter before being sent to the client-browser, therefore SSITaglets are considered **post-processing** taglets for page compilation documents that have the mime type **jsp-internal/taglet**. SSITaglets are usually used to perform further processing, and/or contribute data being sent to the client-browser, and are implemented via the Taglet API. SSITaglets are loaded by the **SSIFilter** through the **codetaglet.properties** file. The **codetaglet.properties** file is explained in a later section, and for examples on creating custom SSITaglets see the chapter *Developing Custom Taglets* under the *Developer's Guide Section*.

## What is a DynamicTaglet?

*DynamicTaglets* provide a way to call a particular servlet through a particular tag. This allows a one-to-one naming convention between a DynamicTaglet and the servlet it represents. The DynamicTaglet is a subclass of SSITaglet and is loaded by the SSIFilter based on the mappings defined in the **taglet.properties** file. The **taglet.properties** file is explained in a later section.

## Loading and Using Custom Taglets

There are two properties files used by the page compilation engine and by the SSIFilter. These two property files define mappings to taglets that are to be loaded and used for the parsing of server-side web documents (i.e. *jsp* files or *html* files). The following describes the purpose of the **taglet.properties** file and the **codetaglet.properties** file respectively.

## Loading and Using DynamicTaglets

The **taglet.properties** file is used specifically by the **SSI filter** for the post-processing of data. Though this property file is editable by hand, the JRunAdmin provides a user-friendly interface for setting these properties. See the documentation for the JRun Admin for more information. The file is used to define name/value pairs describing mappings of **DynamicTaglets** to their corresponding servlet. The **taglet.properties** file is located in your **<JRun Server Pages\_InstallDir>/properties** directory. The general syntax follows:

```
<tagname>=<servlet class name|servlet alias>
```

Where **tagname** is the name of the tag to be used to call the servlet, **servlet class name** is the fully qualified class name of the servlet, and **servlet alias** is the alias of the servlet as defined in the JRun Server Pages Admin under the Aliases tab. Either the **servlet class name** or the **servlet alias** can be used as the value for the tagname. For example:

```
foo = SnoopServlet
```

defines a tag called **foo** to be used to invoke **SnoopServlet**. In this case, SnoopServlet is the class name of the servlet. Defining this mapping allows the web document writer to call SnoopServlet within the document the following way:

```
<foo></foo>
```

Dynamic Taglets allow passing of parameters to the servlet the following way:

```
<foo name1=value1 name2=value2 name3=value3>  
Hello World  
</foo>
```

The servlet writer can then fetch the name/value pair parameters through the **getParameter()** method of the request object from within the servlet. For example, to get the value of the parameter **name1**, you would do the following.

```
request.getParameter("name1")
```

The body of the Dynamic Taglet, in the above case, *Hello World*, would be fetched out using

```
request.getAttribute("taglet.body.foo")
```

Notice how we used the name of the taglet as the third key in the *taglet.body.key* string. If your taglet was named DATETAG, then you would use **getAttribute("taglet.body.datetag")**. In general, this method would return all text in between the opening `<foo>` tag and closing `</foo>` tag.



Note: The attribute name must be in lower case.

## Loading JSPTaglets and SSITaglets

The `codetaglet.properties` file is used by both the **page compilation engine** and the **SSIFilter**. The `codetaglet.properties` file is located in your `<JRun Server Pages_InstallDir>/properties` directory. The file contains name/value pairs for mappings used to specify **codetaglets**, specifically JSPTaglets and SSITaglets. Because the `codetaglet.properties` file is used by both the page compilation engine and the SSIFilter, a distinction must be made to identify which Taglet is used for which system i.e page compilation or ssi. The value on the right side of the equal sign resolves this distinction. The general syntax for defining **codetaglets** follows:

classname={jsp|ssi}

where **classname** is the fully qualified class name of the codetaglet, and the values **jsp** or **ssi** specify the parser of which the taglet is intended for. For example:

```
com.mypackage.MyJSPTaglet=jsp  
com.mypackage.MySSITaglet=ssi
```

where **com.mypackage.MyJSPTaglet** subclasses **JSPTaglet** and is loaded and used in the page compilation engine's parser. The taglet representing **MyJSPTaglet** would be used within a jsp file.

The class **com.mypackage.MySSITaglet** subclasses **SSITaglet** and is loaded and used in the parser of the **SSIFilter**. In the page compilation engine, by default, output of a jsp document is sent through the **SSIFilter** before being sent to the browser. In this case, the **MySSITaglet** would be used not only in shtml files, but within jsp files as well. If output of jsp applications were not sent through the **SSIFilter**, then **MySSITaglet** would only make sense within an shtml file.



# *JSP Administration*

## Setting Page Compilation Properties

This chapter will explain properties for the page compilation engine that apply to the compilation and runtime of compiled pages.

### Properties

The following properties can be set through the *JRun Server Pages Admin Panel*, or manually within the *jsp.properties* file located in your <JRun Servlet Engine\_InstallDir>/properties directory.

#### **checkseconds:**

The *checkseconds* property is the number of seconds the page compilation engine should wait to check the file system for file updates. When the value of the *checkseconds* property is 0, the page compilation engine will query the file system for .jsp file's modtimes on every request. This property is usually set greater than 0 to optimize request/response times of jsp files. This property is initially set to 0 upon JSP installation.

#### **checkjsa:**

The *checkjsa* property, when set to *true*, will allow the page compilation engine to check for the existence of any global.jsa files. When set to *false*, global.jsa files will not be checked for. This property is initially set to *true* upon JSP installation.

#### **checkextends:**

The *checkextends* property, when set to *true*, will allow the page compilation engine to automatically check for modification times and recompile the extended pages of a page compilation file. This is also known as *recursive compilation*. Although this property may be useful during the development phase of page compilation documents, it is relatively expensive since this hierarchy check must be made for every request. This property is initially set to *false* upon JSP installation.

Although recursive compilation of compiled pages is supported, there are situations where changes may not take effect in an extended page.

1. Situations where b.jsp subclasses a.jsp: Suppose a.jsp has been modified and recompiled as a result of asking for a.jsp through a client browser. The newly compiled changes in a.jsp will not be reflected in b.jsp if a.jsp is recompiled and reloaded before b.jsp is requested from a client browser. In order for changes of a.jsp to be reflected in b.jsp, b.jsp has to be “touched” to force reloading of the new a.jsp in b.jsp’s context.
2. Changes in a.jsp will not be apparent through requesting a.jsp in a client browser if a.jsp has been recompiled and reloaded through asking for b.jsp. In order for a.jsp’s changes to be apparent through requesting a.jsp in a client browser, a.jsp has to be “touched”, and then asked for directly through a client browser.

If you want to avoid having to deal with all the “touching”, it is best to stop and restart your web server after the recompilation. This ensures that the changes of the newly recompiled files will fully take place along your compiled page class hierarchy.

### **compiler:**

The *compiler* property is used to specify the compile command used when compiling page compilation files to Java class files. If the compiler property is not specified, the page compilation engine will compile the files in-process using the javac compiler.

When the compiler property is specified, the value of the compiler property must contain required parameter place holders. The parameter place holders, **%c**, **%d**, and **%f** are replaced by the appropriate values by the page compilation engine. A description of each parameter place holder follows:

**%c**: This place holder will be replaced by the value of the classpath variable as known by the JSE. The classpath string will include the path to the servlets directory and the classpath obtained using `System.getProperty("java.class.path")`.

**%d**: This place holder will be replaced by the value of the path to the directory where compiled files will be generated to.

**%f**: This place holder will be replaced by the Java file name corresponding to the requested page compilation document.

The following example illustrates how to specify the Microsoft compiler for documents written for the page compilation engine:

```
compiler=jvc /cp:c %c /dest: %d %f
```

By specifying the Microsoft compiler, page compilation writers may have the freedom to implement code that allows efficient use of the underlying Windows environment. Page compilation documents containing COM objects can be successfully compiled independent of the type of virtual machine the JRun Servlet Engine happens to be running on. However, in order to successfully run the compiled page that utilizes COM objects, the JSE must run on top of the Microsoft Virtual Machine. Since the compilation phase is independent of the run time environment, page compilation writers who just want a faster compiler, but still want to run their servlets on the Java Virtual Machine can still do so by simply specifying the compiler property. So, a page compilation writer that happens to be developing on the Windows platform, and wants to use jview, rather than javac to compile her documents can still have them run on top of the Java Virtual Machine as long as the compiled pages do not reference any COM objects.

The *compiler* property is initially not specified upon JSP installation. Therefore, compilation will occur in-process using the javac compiler.

**encoding:**

The *encoding* property specifies the character encoding used on page compilation documents. When not specified, the system's default encoding will be used on page compilation documents. This property is initially not specified upon JSP installation.

## Optimizing Page Compilation for Deployment

The page compilation engine can be optimized to run its pages for deployment. The following settings assume the compiled pages are ready to deploy, i.e. there are to be no future modifications and/or compilation of page compilation documents. Simply put, the application is ready to do without the '.jsp' files and can run just fine on its servlet class files alone. Settings can be made in the *jsp.properties* file manually (located in your <JrunServerPages\_InstallDir>/properties directory) or through the *JRun Server Pages Admin Panel*.

- Set the *checkseconds* property value to any integer **greater than 0**. Setting this value greater than zero allows the page compilation engine to deploy a thread to check for updated files. The thread is responsible for caching requested file dates. This allows the page compilation engine to restrict itself from performing any file I/O for requested documents. It is recommended to set this value to a very high integer in seconds.
- Set the *checkextends* property value to **false**. This prevents the page compilation engine from having to check modification times of extended compiled documents since compilation may no longer be required in deployment.

- If your page compilation application does not use global.jsa files. You can set the *checkjsa* option to **false**. Otherwise it must be set to **true**.



# *Scripting Developer's Guide*

## *Developing Custom Taglets*

### Examples for Writing JSPTaglets

This section walks you through a couple of examples for creating JSPTaglets. *JSPTaglets* are user defined taglets that are used within a page compilation document (see the *Taglets* chapter under *Section II.*). The first example is of a loop taglet. The second example is a set of taglets used for performing conditional constructs within a page compilation document. Both the loop taglet and the conditional taglets can very simply be done using embedded Java, but one of the purposes of taglets is to allow the freedom to define tag based constructs to help keep a uniform and readable page compilation document.

#### The JSPLoopTaglet

The JSPLoop taglet is a simple JSPTaglet that provides a more readable alternative to using embedded Java for looping in a page compilation document. The general syntax for the loop tag follows:

```
<loop index=<string> from<integer> to=<integer> [step=<positive integer>] >
</loop>
```

where the value of

**index:**

represents the variable name that will hold the current increment in the loop.

**from:**

is any **java qualified expression** that evaluates to an integer representing the starting index of the loop.

**to:**

is any **java qualified expression** that evaluates to an integer representing the index limit of the loop.

**step:**

is an **optional** parameter. The value of **step** specifies the increment of the index. The value of **step** will default to “1” when it is not specified. When it is specified, the value **must** be a positive integer.

The following is an example of the loop taglet usage.

```
<loop index="LoopCount" from="1" to="8" step="1">
<font size=<%=LoopCount%>> Hello World</font><br>
</loop>
```

The next example illustrates the use of a Java qualified expression used for the *from* and *to* parameter values.

```
<% int START = 1; %>
<loop index="LoopCount" from="START" to="START+7">
  <font size=<%=LoopCount%> >Hello World</font><BR>
</loop>
```

Example 1-0 is the implementation of the loop taglet called *JSPLoop*. The source file *JSPLoop.java* can be found in the directory under **<JRun\_InstallDir>/src/**.

**Example 1-0: JSPLoop.java**

```
package com.livesoftware.jrun.plugins.jsp.taglets;

import com.livesoftware.jrun.plugins.jsp.JSPTaglet;
import com.livesoftware.jrun.plugins.jsp.ServletBuilder;
import com.livesoftware.jrun.plugins.jsp.JSPTaggableHandler;
import com.livesoftware.jrun.plugins.jsp.JSPParseException;

import com.livesoftware.jrun.plugins.ssi.ITaggableData;
import com.livesoftware.jrun.plugins.ssi.SSIAbortException;
import com.livesoftware.jrun.plugins.ssi.TagletAggregate;
import com.livesoftware.jrun.plugins.ssi.Tagable;

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

import java.io.IOException;
import java.util.Vector;
/**
 * This class adds tag based looping capabilities in an html document.
 *
 * Example usage from a jsp page.
 *
 * <loop index="LoopIndex" from="0" to="5">
 * <B>Hello World</B>
 * </loop>
```

```

*
* An optional parameter is 'step'. The value of
* the step parameters specifies the number of iterations
* to skip in the loop. The 'step' parameter is optional
* and will default to 1 when not specified.
* The following is an example of specifying the "step"
* parameter.
*
* <loop index="LoopIndex" from="0" to="6" step="2">
* <B>Hello World</B>
* </loop>
*/
public class JSPLoop extends JSPTaglet
{
    private static final char[] startPat = {'<','l','o','o','p'};
    private static final char[] endPat = {'<','/','l','o','o','p','>'};

    public char[] getStartPattern() { return startPat; }
    public char[] getEndPattern() { return endPat; }

    public void handleTag(ITaggableData td, Servlet jspServlet,
        HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
    {
        //Get the taglet parameters
        String index = td.getParameter("index");
        String from = JSPEval(td.getParameter("from"));
        String to = JSPEval(td.getParameter("to"));
        String step = JSPEval(td.getParameter("step"));

        // If step is not specified, we'll use '1'. Also, we'll make sure it is
        // not negative number.
        if(step == null) step = "1";
        else if(step.startsWith("-")) step = step.substring(1);

        // Check to see that all required parameters were specified
        if( index == null )
            throw new JSPParseException(
                "'INDEX' parameter not specified in loop tag. ", td);
        if( from == null )
            throw new JSPParseException(
                "'FROM' parameter not specified in loop tag. ", td);
        if( to == null )
            throw new JSPParseException(
                "'TO' parameter not specified in loop tag. ", td);

        //Instantiate a JSP parser
        JSPTaggableHandler th = new JSPTaggableHandler(getServletBuilder(), td);

        //Let the parser know of what taglets to look out for
        Vector vectorOfJSPTaglets =
            ((TagletAggregate)jspServlet).getAllTaglets();
        th.setTagables(vectorOfJSPTaglets);

        //Set the data to be parsed
        char[] dataToParse =
            preprocessData(index, to, from, step, td.getTagBody());
        th.setDataSource(dataToParse);

        //Parse the data
        try {
            th.handleData(jspServlet, req, res);
        }catch(SSIAbortException e){}
    }
}

```

```

/**
 * A method that creates the logic in the jsp page that the JSPLoop tag
 * replaces.
 */
private static char[] preprocessData(String index, String to, String from,
                                     String step, String data)
{
    StringBuffer preprocessed=new StringBuffer();

    preprocessed.append("<% if (" +from+ " <= " +to+) { %>\n");
    preprocessed.append("<% for(int "+index+"="+from+
        "+ "+index+"<="+to+";"+index+"+=" + step + ") { %>\n");
    preprocessed.append(data);
    preprocessed.append("<% } %>\n");
    preprocessed.append("<% } %>\n");
    preprocessed.append("<% else { %>\n");
    preprocessed.append("<% for(int "+index+"="+from+
        "+ "+index+">="+to+";"+index+"-=" + step + ") { %>\n");
    preprocessed.append(data);
    preprocessed.append("<% } %>\n");
    preprocessed.append("<% } %>\n");

    return preprocessed.toString().toCharArray();
}
}

```

The import list below shows the list of classes used by the JSPLoop taglet. Further examination of the **jsp** and **ssi** packages can be accomplished through reviewing the Taglet API javadocs located under the directory `<JSP_InstallDir>/docs/tagletapi`

```

import com.livesoftware.jrun.plugins.JSP.JSPTaglet;
import com.livesoftware.jrun.plugins.JSP.ServletBuilder;
import com.livesoftware.jrun.plugins.JSP.JSPTagableHandler;
import com.livesoftware.jrun.plugins.JSP.JSPParseException;

import com.livesoftware.jrun.plugins.ssi.ITagableData;
import com.livesoftware.jrun.plugins.ssi.SSIAbortException;
import com.livesoftware.jrun.plugins.ssi.TagletAggregate;
import com.livesoftware.jrun.plugins.ssi.Tagable;

```

JSPLoop is an implementation of the abstract class JSPTaglet. JSPLoop overrides the abstract methods `getStartPattern()`, `getEndPattern()`, and `handleTag()` of JSPTaglet.

The `getStartPattern()` method returns a `char[]` representing the opening of a tag. The following character array represents the *start pattern* of the JSPLoop class.

```
{ '<', 'l', 'o', 'o', 'p' }
```

Notice that there is no closing `'>'` in the character array. This is because the loop tag needs to take in parameters. Leaving the `'>'` out of the pattern, lets the parser know to look for any name/value pairs that may exist in this taglet.

The `getEndPattern()` method returns a `char[]` representing the tag pattern that closes a tag. The following character array represents the end pattern of the `JSPLoop` class.

```
{ '<', '\', '!', 'o', 'o', 'p' }
```

The `handleTag()` method contains the logic to perform specific operations for the taglet whenever the taglet is encountered by the parser. It is called by the parser whenever the taglet's character pattern is recognized. Its parameter-list is made up of `ITaggableData`, `Servlet`, `HttpServletRequest` and `HttpServletResponse`.

***ITaggableData*** is an interface that is used to get parameters passed to the taglet. `ITaggableData` is also used to get the `String` that represents the text in-between the taglet's start and endpattern.

The ***Servlet*** object is the reference to the page compilation servlet and is also a ***TagletAggregate***. `TagletAggregate` acts as a factory for supplying a `Vector` of taglet objects. The method `getAllTaglets()` of `TagletAggregate` is used to get a `Vector` of `JSPTaglet` instances. The `Vector` contains the jsp internally defined `JSPTaglets` as well as taglets specified in the `codetaglet.properties` file that have the value of `jsp`.

The ***HttpServletRequest*** and ***HttpServletResponse*** objects provide a connection to the client. See the javadocs of these objects for a complete description of their methods. Javadocs for `HttpServletRequest` and `HttpServletResponse` can be found under the directory `<JRun_InstallDir>/docs/jdkapi/>`.

The ***JSPTagableHandler*** is a jsp parser that is used to parse the pre-processed string returned by the `preprocessData()` method. The results of the parse data will be sent to the servlet being built. See the javadocs of `JSPTagableHandler` for a complete description of its methods. Javadocs for `JSPTagableHandler` can be found in the directory `<JRun_InstallDir>/docs/tagletapi`.

`JSPTaglet` class also provides a reference to a ***ServletBuilder*** instance. `ServletBuilder` is an interface to the servlet being built. The reference to the `ServletBuilder` instance can be obtained by the `getServletBuilder()` method. See the javadocs for a detailed list of defined methods in `ServletBuilder` and `JSPTaglet`.

## The `JSPConditionalTaglet`

The `JSPConditional` taglet provides a simple tag based alternative to using embedded Java for conditional statements in a JSP document. The general syntax for the `JSPConditional` taglet follows:

```

<if condition=<boolean expression> >
{ <elseif condition=<boolean expression> /> }*
{ <else> }*
</if>

```

Where the parameter name:

**condition:**

is any Java qualified expression that evaluates to a boolean.

The JSPConditional taglet requires that there **must** be a close `</if>` for every opening `<if>`, and the `<elseif/>`, `<else>` tags always be contained within a `<if></if>` pair. See the following example.

```

<% int NUMBER="5"; %>
...
<if condition="<%= NUMBER == 5 %>">
  <b>The number is 5</b>
<elseif condition="<%= NUMBER == 6 || NUMBER == 7 %>" />
  <b> The number is 6 or 7</b><br>
<elseif condition="<%= NUMBER == 8 %>" />
  <b> The number is 8</b><br>
  <if condition="<%= true %>">
    <b><Nested If></b>
  </if>
  <else>
    <b> The number is not found</b><br>
  </if>

```

Example 1-1 is the implementation of the loop taglet called *JSPConditional*. The source file *JSPConditional.java* can be found in the directory under **<JRun\_InstallDir>/src/**.

***Example 1-1: JSPConditional.java***

```

package com.livesoftware.jrun.plugins.jsp.taglets;

import com.livesoftware.jrun.plugins.jsp.JSPTaglet;
import com.livesoftware.jrun.plugins.jsp.ServletBuilder;
import com.livesoftware.jrun.plugins.jsp.JSPTagableHandler;
import com.livesoftware.jrun.plugins.jsp.JSPParseException;

import com.livesoftware.jrun.plugins.ssi.ITagableData;
import com.livesoftware.jrun.plugins.ssi.SSIAbortException;

```

```

import com.livesoftware.jrun.plugins.ssi.TagletAggregate;
import com.livesoftware.jrun.plugins.ssi.Tagable;

import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Vector;

/**
 * This class adds tag based conditional capabilities in a tag based document.
 */
public class JSPConditional extends JSPTaglet
{
    private static final char[] startPat = {'<', 'i', 'f'};
    private static final char[] endPat = {'<', '/', 'i', 'f', '>'};

    public char[] getStartPattern() { return startPat; }
    public char[] getEndPattern() { return endPat; }

    public void handleTag(ITagableData td, Servlet jspServlet,
        HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
    {
        String condition = JSPEval(td.getParameter("condition"));

        //Throw a parse exception if condition is null
        if( condition == null ) throw new JSPParseException(
            "if tag parameter 'condition' not specified", td);

        //Get a reference to the servlet being built
        ServletBuilder sb = getServletBuilder();

        sb.appendJava("if (" + condition + ") { \n");

        //Instantiate a JSP parser
        JSPTagableHandler th = new JSPTagableHandler(sb, td);

        //Let the parser know of what tags to look out for
        Vector vectorOfJSPTaglets =
            ((TagletAggregate)jspServlet).getAllTaglets();
        th.setTagables(vectorOfJSPTaglets);
        th.addTagable(new JSPElseif());
        th.addTagable(new JSPElse());

        //Set the data to be parsed
        th.setDataSource(td.getTagBody().toCharArray());

        //Parse the data
        try {
            th.handleData(jspServlet, req, res);
        } catch (SSIAbortException e){}

        //Close the if condition
        sb.appendJava("}\n");
    }

    /**
     * A member class that handles the <elseif> tag.
     */
    class JSPElseif extends JSPTaglet
    {
        private char[] startPat = {'<', 'e', 'i', 's', 'e', 'i', 'f'};
        private char[] endPat = {'/', '>'};
    }
}

```

```

public char[] getStartPattern() { return startPat; }
public char[] getEndPattern() { return endPat; }

public void handleTag(ITaggableData td, Servlet jspServlet,
                    HttpServletRequest httpRequest,
                    HttpServletResponse httpResponse)
    throws ServletException, IOException
{
    String condition = JSPEval(td.getParameter("condition"));

    //Throw a parse exception if condition is null
    if( condition == null ) throw new JSPParseException(
        "if tag parameter 'condition' not specified", td);

    getServletBuilder().appendJava(" else if (" + condition + ") { ");
}

}

/**
 * A member class that handles the <else> tag.
 */
class JSPElse extends JSPTaglet
{
    private char[] startPat = {'<', 'e', 'l', 's', 'e'};
    private char[] endPat = {'>'};

    public char[] getStartPattern() { return startPat; }
    public char[] getEndPattern() { return endPat; }

    public void handleTag(ITaggableData td, Servlet jspServlet,
                        HttpServletRequest httpRequest,
                        HttpServletResponse httpResponse)
        throws ServletException, IOException
    {
        getServletBuilder().appendJava(" else { ");
    }
}
}

```

The important thing to note about the implementation of the JSPConditional taglet is its use of internally defined JSPTaglets: *JSPElseIf*, and *JSPElse*. The power about using the Taglet API, is it allows you to define context sensitive taglets for your jsp documents. Context sensitive taglets such as *JSPElseIf* and *JSPElse* should **not** be specified in the **codetaglet.properties** file. Taglets containing context sensitive taglets should handle its own parsing of its tagBody. Parsing is possible by using the JSPTaggableHandler instance.

## Examples for Writing SSITaglets

JRun Scripting Toolkit allows web developers to define taglets used by the SSI Filter (see the *Taglets* chapter under *Section II*). Taglets defined in this context are used to parse through text one last time before being sent to the browser. Output of page compilation documents with a content-type of **jrun-internal/taglet** are sent through the SSI Filter before going to the browser. The Taglet API can be implemented to define SSITaglets that can be loaded and used by the SSI Filter.

## The SSIMail Taglet

The following example is an implementation of a simple SSITaglet that utilizes the *com.livesoftware.jrun.plugins.jsp.beans.JSPMail* bean to send a mail message. The source file *SSIMail.java* can be found in the directory under **<JSP\_InstallDir>/examples/src/**.

### **Example 1-1: SSIMail.java**

```
package com.livesoftware.jrun.plugins.ssi.taglets;

import com.livesoftware.jrun.plugins.ssi.SSITaglet;
import com.livesoftware.jrun.plugins.ssi.ITaggableData;
import com.livesoftware.jrun.plugins.ssi.SSIParseException;

import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.livesoftware.jrun.plugins.jsp.beans.JSPMail;

import java.io.IOException;
import java.util.StringTokenizer;

/**
 * SSIMail is a simple SSITaglet that uses the JSPMail bean to
 * send mail.
 * <p>
 * Example usage of this SSITaglet within a jsp document follows:
 * <p>
 * <pre>
 * <SSIMAIL TO="<%= getParameter("recipient") %>" FROM="me" HOST="myHost"
 * FROM_ADDRESS="<%= request.getParameter()>"
 * <%= db.queryData("select 'message' from database") %>
 * </SSIMAIL>
 * </pre>
 */
public class SSIMail extends SSITaglet
{
    private static char startPat[] = {'<', 'S', 'S', 'I', 'M', 'A', 'I', 'L'};
    private static char endPat[] = {'<', '/', 'S', 'S', 'I', 'M', 'A', 'I', 'L', '>'};

    public char[] getStartPattern()
    {
        return startPat;
    }

    public char[] getEndPattern()
    {
        return endPat;
    }

    public void handleTag(ITaggableData td, Servlet s,
        HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        String paramValue = null;
    }
}
```

```

StringTokenizer st = null;

//Instantiate the JSPMail object
JSPMail mailer = new JSPMail();

//Set the TO parameter. May be comma delimited
paramValue = td.getParameter("to");
if ( paramValue == null || paramValue.equals("") ) throw new
    SSIParseException("Required parameter 'TO' not specified.", td);
st = new StringTokenizer(paramValue, ",");
while( st.hasMoreElements() )
{
    String r = st.nextToken();
    mailer.setRecipient(r);
}

//Set HOST
paramValue = td.getParameter("host");
if ( paramValue == null || paramValue.equals("") ) throw new
    SSIParseException("Required parameter 'HOST' not specified.", td);
mailer.setMailHost(paramValue);

//Set FROM_ADDRESS
paramValue = td.getParameter("from_address");
if ( paramValue == null || paramValue.equals("") ) throw new
    SSIParseException("Required parameter 'FROM_ADDRESS' not specified.", td);
mailer.setFromAddress(paramValue);

//Set FROM_NAME
paramValue = td.getParameter("from_name");
if ( paramValue != null ) mailer.setFromName(paramValue);

//Set SUBJECT
paramValue = td.getParameter("subject");
if ( paramValue != null ) mailer.setSubject(paramValue);

//Set the CC parameter. May be comma delimited
paramValue = td.getParameter("cc");
if ( paramValue != null && !paramValue.equals("") )
{
    st = new StringTokenizer(paramValue, ",");
    while( st.hasMoreElements() )
        mailer.setCcRecipient(st.nextToken());
}

//Set MIME
paramValue = td.getParameter("mime_type");
if ( paramValue != null ) mailer.setMimeType(paramValue);

//Set the body of the message
String body = td.getTagBody();
if (body == null) mailer.setBodyText("");
mailer.setBodyText(body);

//Send mail
mailer.sendMail();
}
}

```

The SSIMail taglet is a subclass of SSITaglet. In order for this class to be loaded and used by the SSI parser, an entry for the SSIMail taglet must be entered in the **codetaglet.properties** file. The codetaglet.properties file should have the following entry:

```
com.livesoftware.jrun.plugins.ssi.taglets.SSIMail=ssi
```

## Extensions to Servlet API

JRun has added support for jsp documents by implementing the ability for a **servlet** to programmatically call a jsp document as well as provide support for passing request scope objects on to a jsp document. The following objects are available from within a servlet invoked under the JSE.

### The Class

#### `com.livesoftware.jrun.JrunServletRequest`

This class implements the `javax.servlet.http.HttpServletRequest` interface and adds two methods to set and retrieve attributes defined by name. The relevant methods follow:

**getAttribute(String name):**

This method retrieves an attribute from the request context.

**Parameters:**

**name:** - The name of the object to retrieve, or null if an object with the given name does not exist.

**setAttribute(String name, Object object):**

This method sets an attribute

**Parameters:**

**name:** - The name of the object being stored in the Request

**value:** - The object being stored in the Request

### The Class

#### `com.livesoftware.jrun.JRunServletResponse`

This class implements the `javax.servlet.http.HttpServletResponse` interface and adds a method that allows servlets to call pages and optionally pass a context. The relevant method follows:

**callPage(String fileName, HttpServletRequest req):**

This method is used to serve a JSP page from within a servlet. Some context can be passed to it via the request object. The file passed back will be passed with a header directive that will indicate to the browser that this page is not to be cached.

**Parameters:**

**fileName:** - The name of the URL that identifies a file that will be used to generate the output and present the content. When the name begins with a "/" it is assumed relative to the document root. When it does not begin with a "/" it is assumed to be relative to the URL that this current request was invoked with.

**req:** - The HttpServletRequest object of the servlet invoking this method. Typically the content is passed as a bean in the context of the request object.

An example call to a jsp page from within a servlet's service method follows:

```
service(HttpServletRequest req, HttpServletResponse res)
throws IOException, ServletException
{
    JRunServletRequest jrunReq = (JRunServletRequest)req;
    jrunReq.setAttribute("greeting", "Hello World");

    JRunServletResponse jrunRes = (JRunServletResponse)res;
    jrunRes.callPage("/a.JSP", jrunReq);
}
```

The JSP file **a.jsp** would retrieve the data as follows:

```
<%
    String greeting = (String)request.getAttribute("greeting");
%>
```

## *Page Compilation Scripting Guide*

This section will explain the main parts of page compilation supported by Live Software.

### Compatibility

The following specs for page compilation is a superset of the Java Server Pages specs described by JavaSoft and the Java Web Server team. Therefore, any page compilation page conforming to the Java Server Pages spec is compatible to be used under our page compilation environment. Additionally, Live Software's page compilation adds the use of Microsoft Active Server Pages™ (ASP)-like objects. One such functionality includes the ability to set multi-valued cookies. This ASP-like capability is geared toward web developers who are looking to migrate their work from ASP format to our page compilation format.

### Available objects

The following objects are predefined for use within the service method of the compiled page.

**Request and request:**

the servlet request class as defined by  
`com.livesoftware.jrun.plugins.JSP.JSPRequest` **JSPRequest** is a concrete class that implements the `HttpServletRequest`

**Response and response:**

the servlet response class as defined by  
`com.livesoftware.jrun.plugins.JSP.JSPResponse` **JSPResponse** is a concrete class that implements the `HttpServletResponse`

**out:**

the servlet output stream class as defined by  
`javax.servlet.ServletOutputStream`

A function descriptions of these objects as well as others defined specifically for use within page compilation files, can be found in the *Page Compilation Object Reference* chapter. In addition, see the servlet api documentation for possible

additional method definitions of these objects located in your `<JSP_InstallDir>/docs/` directory.

## Syntax

Basic page compilation is made up of 5 main types of tags.

1. directive taglet
2. class wide declaration taglet
3. scriptlet taglet
4. expression taglet
5. BEAN taglet

## The Directive Taglet

The *directive taglet* specifies properties for the generation of the servlet corresponding to the jsp file. The general syntax for the directive taglet follows:

```
<%@ {variable = "< value >"}+ %>
```

Where

```
variable = {language|method|import|implements|  
extends|include|vinclude|content_type }
```

Values of the variables must be literal strings. The directive taglet is considered a pre-processing tag. The values of the directive taglet are evaluated during the parse time of a jsp document and is used to set certain properties for the servlet that is being created. Since the directive taglet is evaluated during parse-time and NOT run-time of the compiled page, Java expressions used for the variable values will not make sense in the pre-processing context of the directive taglet. For this reason the following example usage of the directive taglet would not correct.

```
<% String myVar = "myPackage.*,com.foo.myClass"; %>  
...  
INCORRECT <%@ import="<%=myVar%>" %>
```

In short, directive taglets are in no way nestable, and cannot include any other page compilation taglet. This includes the expression taglet `<%= %>`.

Description of the variables and their values follow:

### **language :**

The *language* variable defines the scripting language used in the file. The scope of this tag spans the entire file. When used more than once, only the first tag is significant. If omitted entirely, the default scripting language used is *java* (for the Java programming language). This is the only value that *language* can accept at this time.

### **method:**

The *method* variable specifies the method name that will contain the generated servlet code. By default, the method defined in the generated servlet is *service*. When a specific method name is used, the generated code becomes the body of the specified method name. An example is shown below.

```
<%@ method = "doPost" %>
```

**import:**

The *import* variable specifies a comma delimited list of any packages that the compiled page will need to import for use. An example is shown below.

```
<%@ import = "java.io.*,java.util.Hashtable" %>
```

**implements:**

The *implements* variable specifies a comma delimited Java interface which the compiled page will implement. An example is shown below.

```
<%@ implements = "com.myPackage.MyServletInterface" %>
```

**extends:**

The *extends* variable specifies a Java servlet which the compiled page will subclass.

```
<%@ extends = "com.myPackage.AServletImplementation" %>
```

**include:**

The *include* variable will allow you to insert the contents of a file relative to the local file system.

**vinclude:**

The *vinclude* variable will allow you to insert the contents of a file relative to the web document root.

**content type:**

The *content\_type* variable defines the content type of the generated response. By default, a `jrunit-internal/taglet` content type is generated. But the default can be overridden by setting this directive. When used more than once, only the first tag is significant. An example is:

```
<%@ content_type="text/html;charset=UTF-8" %>
```



Note: By overriding the default `content_type` of `jrunit-internal/taglet`, the compiled page will not be post-processed for SSITaglets, the `<servlet>` tag, and `<!--#include>` tags.

## The Class Wide Declaration Taglet

The page compilation engine allows class-wide declarations within the compiled page through the *SCRIPT* taglet. The *SCRIPT* taglet opens with a

```
<SCRIPT runat=server>
```

and is closed with

```
</SCRIPT>
```

See the following example.

```
<SCRIPT runat=server>
private String foo = null;
public String getFoo() { return this.foo; }
</SCRIPT>
```

The **runat** variable is **required** to let the page compilation engine know the tag is to be processed on the server end. The script taglet is usually used to define any class wide variables, or methods (see the method `getFoo()` ).

## The Scriptlet Taglet

The scriptlet taglet specifies the Java code to be generated into the compiled page's service method. If a method is specified explicitly through the directive taglets's **method** property, the Java code will be generated within the specified method. Any valid Java code can be specified within the body of the scriptlet taglet. The general syntax for the scriptlet tag follows.

```
<%
  service scope JAVA code goes here...
%>
```

An example of embedded Java code would be the following:

```
<%
  String greeting = request.getParameter("aGreeting");
  out.print(greeting);
%>
```

## The Expression Taglets

Java expressions which are to be evaluated before going to the client-browser are specified by using the expression taglet. For convenience to the page compilation writer, two syntactical variations of performing this task can be used.

Variation 1:     <%= javaExpression %>

Variation 2:     `javaExpression`

Variation 2 surrounds the javaExpression with back-tics. Both variations perform equivalent tasks. An example of variation 1 follows:

```
<table>
<tr>
<td><%= Request("data1")%></td>
<td><%= Request("data2")%></td>
</tr>
</table>
```

An example of variation 2 follows:

```
<table>
<tr>
<td>`Request("data1")`</td>
<td>`Request("data2")`</td>
</tr>
</table>
```

The two equivalent examples will evaluate the Java expressions before the output goes to the client browser.

## The JSP BEAN Taglet

The JSP BEAN taglet is a powerful tool that brings Java Bean technology to page compilation writers, allowing the reuse of server-side components. By using the BEAN taglet, the page compilation writer has the ability to create the bean from a serialized file or a class file, and set its properties from either session, request, or application parameters. A Bean can also be passed on to a compiled page from another servlet, allowing access to the Bean by the compiled page. JSP allows access to a bean via the BEAN tag. The general syntax for opening the bean tag follows:

```
<BEAN name=" < value > " varname=" < value > " type=" < name > " introspect="{yes|no}" beanName=" < value > " create="{yes|no}" scope="{request|session|application}">
```

The bean tag is closed with the **required** </BEAN> tag. Between the < BEAN ... > and the < /BEAN > tag, a list of PARAM tags can be optionally specified. The attributes inside the BEAN tag need not appear in any particular order. A description of the bean parameters follows:

**name:**

The value of the ***name*** parameter specifies the name of which the bean will be associated with through its life. This name is used in fetching and storing the bean in its associated scope object. This is a **required** parameter.

**varname:**

The value of the ***varname*** parameter identifies the variable name that will refer to this bean in the scope of the jsp page. This is an optional attribute. If it is not specified, the name of the variable defaults to the name of the bean as specified in the ***name*** attribute above.

**type:**

The value of the ***type*** attribute is the name of the class or interface that defines the bean instance in the code. When not specified the value defaults to the type *Object*.

**introspect:**

The value of the ***introspect*** parameter is either a *yes* or a *no*. If it is not specified, the value defaults to *yes*. When the value is *yes*, form/query data properties passed in through the request are inspected and used to set matching properties in the bean. The resulting bean will have its properties set to the parameter values as passed in by the client that called the page. If set to *no*, then the form/query data properties in the request object are not used to set the bean's properties and the bean will keep its current state.

**create:**

The value of the ***create*** attribute is either a *yes* or a *no*. If it is not specified, the value defaults to "yes". When the value is *yes*, the bean is created if it is not found in the scope specified.

**scope:**

The value of the ***scope*** parameter can be one of these values:

**request:**

This bean is retrieved from the ***request*** context. The bean is set as a context in the request by a servlet invoking this JSP page using the APIs described later in this document. This is the default value used when scope is not specified. If the bean is not part of the request context, then the bean is created and stored in the request context if the *create* parameter is *yes*.

**session:**

This bean is reused from the current **session** if present. If not present, it is created and stored as part of the session if the *create* parameter is *yes*.

**application:**

This bean is retrieved from the **application** if present. If not present, it is created and stored in the application if the *create* parameter is *yes*.

The **scope** attribute is optional. If it is not specified, it defaults to the *request* scope.

**beanName:**

The value of the beanName attribute is the name of the serialized file or class file that contains the bean. This attribute is used only when the bean is not present in the scope of the BEAN tag and the value of *create* is *yes*. See the description of Beans.instantiate() for more details.

As described earlier, the BEAN tags can optionally have PARAM tags between its open and closing tags. The syntax of the PARAM tag that is allowed to be optionally nested inside the BEAN tag is shown below:

```
<PARAM {name="value"}+>
```

The PARAM tag can define a value for a list properties that will be automatically set in the bean via JavaBeans introspection. These properties are set once for each bean when the bean is created. If the request has parameters that set the value of the same properties that the PARAM tag set, the request properties override the PARAM properties.

An example *BEAN* tag might be:

```
<BEAN name="foobar" type="FooClass" scope="request">  
<PARAM fooProperty="fooValue" barProperty="1">  
</BEAN>
```

Once a bean is declared it can be accessed anywhere in the file. For example, the bean declared in the example above can be accessed inside a compiled page as follows:

```
The name of the row is <%= foobar.getRowName() %>
```

## *Page Compilation Object Reference*

### From ASP to JSP

The main motivation for creating the page compilation system was to provide web developers with the best and the latest of server-side development technology. Live Software has realized the need for a platform independent server-side scripting solution, and has developed a system that combines the ease of ASP with the power of Java Servlets. Before the release of JSP, the expense to ramp up on the Java Servlet API may have discouraged away ASP developer's, forcing them to drudge on their limiting, and platform dependent development path. Hence, a needed solution for ASP writers who have seen the greener grass of Java Servlets, and needing a less expensive way to exploit its advantages. This chapter will acquaint you with the built-in asp-like objects only available within a jsp document. For the complete list of the method definitions for the following objects see the api docs located in your `<JSP_InstallDir>/docs`.

### Application Object

You can use the *Application* object to share information among all users of a given application. The Application object is of type `com.livesoftware.jrun.plugins.jsp.JSPApplication`. A JSP-based application is defined as all the .jsp files in a virtual directory and its subdirectories. For the complete list of the `JSPApplication` method definitions see the api docs located in your `<JSP_InstallDir>/docs`.

Syntax

**Application.method**

Methods

**setAttribute(String objName, Object object)**

Store the object with the given object name in this application

**getAttribute(String objName, String defaultMessage)**

Returns the object with the objName.

Returns defaultMessage if the object of the objName does not exist.

**get(String objName)**

Returns the object with the objName

Returns null if the object of the objName does not exist.

**removeAttribute(String objName)**

Remove the object with the given name from the application.

The following illustrates an example usage of the Application object, and its ASP equivalence.

***JSP Example of Application Object***

```
---file1.jsp---
<%
//Creating and initializing the array
String[] MyArray = new String[5];
MyArray[0] = "hello";
MyArray[1] = "some other string";

//Storing the array in the Application object
Application.setAttribute("StoredArray", MyArray);
Response.sendRedirect("file2.jsp");
%>

---file2.jsp---
<%
//Retrieving the array from the Application Object
//and modifying its second element.
//Changes made to the array automatically updates the global .
String[] NotLocalArray = (String[])Application.get("StoredArray");
NotLocalArray[1] = " there";

'printing out the string "hello there"
Response.write(NotLocalArray[0] + NotLocalArray[1]);
%>
```

***ASP Equivalent Example of Application Object***

```
---file1.asp---
<%
'Creating and initializing the array
dim MyArray()
Redim MyArray(5)
MyArray(0) = "hello"
MyArray(1) = "some other string"

'Storing the array in the Application object
Application.Lock
Application("StoredArray") = MyArray
Application.Unlock
Response.Redirect("file2.asp")
%>

---file2.asp---
<%
'Retrieving the array from the Application Object
'and modifying its second element
LocalArray = Application("StoredArray")
LocalArray(1) = " there"

'printing out the string "hello there"
Response.Write(LocalArray(0)&LocalArray(1))

'Re-storing the array in the Application object
'This overwrites the values in StoredArray with the new values
Application.Lock
Application("StoredArray") = LocalArray
```

```
Application.Unlock
%>
```

As you can see, the syntax is of course different, one uses Java the other uses VB. The most noticeable difference is the use of `Application.Lock` and `Application.Unlock` in the ASP example when getting or putting the array object into the `Application` object. In JSP this does not have to be specified, the method of locking and unlocking of getting or putting objects into a `JSPApplication` is handled internally. Another important issue to point out is, in ASP objects retrieved from an `Application` are simply copies of the object that was stored. In JSP objects retrieved from an `Application` are the same reference to the object that was stored. This is why `file1.asp` has to re-assign the `LocalArray` to the `Application` at the end of the file in so the new value of “hello there” can be retrieved by another ASP page. The `file1.jsp` does not need to reassign the array into the `Application`, since it is the same reference to the object that was stored.

## Request/request Object

The **Request** object retrieves the values that the client browser passed to the server during an HTTP request. The **Request** object is of type `com.livesoftware.jrun.plugins.jsp.JSPRequest`. The following definitions are a subset of methods defined in the the `JSPRequest` class. Additionally the `JSPRequest` class subclasses `HttpServletRequest`. This means all methods of `HttpServletRequest` are also implemented in the `JSPRequest` class. Additional supported methods implemented in the `JSPRequest` class are defined in the api docs. For a complete listing of methods defined in **JSPRequest**, see the docs under `<JRun_InstallDir>/docs` directory.

All variables can be accessed directly by calling **Request(String name)**. In this case, the page compilation engine searches the collections in the following order.

1. Form/QueryString
3. Cookies
5. ServerVariables

If a variable with the same name exists in more than one collection, **Request** returns the first instance that it encounters.

Syntax

**Request**[*.Method*](*variables...*)

Methods

**Cookies()**

Returns an array of cookie objects. An empty cookie array will be returned if no cookies exist.

**Example:**

```
<%
Cookie[] c = Request.Cookies();
for (int i=0; i<c.length; i++)
    Response.write( c[i].getName() + "<BR>");
%>
```

**Cookies(String name)**

Returns a single cookie object with the given name. If there are more than one cookie with the same name, then the cookie with the longest path is returned. Returns null if the named cookie does not exist.

**Example:**

```
<% Cookie c = Cookies("myCookieName"); %>
```

**Cookies(String name, String subkey)**

A method used to access a subvalue of a multivalued cookie.

*name* is the name of the cookie to access.

*subkey* is the key of the value to obtain.

**Example:**

```
<%= Request.Cookies("mycookie","type1") %>
```

**CookiesArray(String)**

Returns an array of cookie objects with the given name. An empty cookie array object will be returned if no cookies of the given name has been set.

**Form()**

Returns a comma delimited String of all form names or empty string.

**Form(String name)**

Returns a comma delimited String of form values with the given name. Returns empty string if a value of the given name was not found. This function is equivalent to a call to *QueryString(String name)*.

**FormArray()**

Returns an array of parameter names from the client browser. Returns an empty String[] of length 0 if there are none.

**FormArray(String name)**

Returns the values of the specified parameter for the request as an array of strings, or an empty array if the named parameter does not exist. This function is equivalent to a call to *QueryStringArray(String name)*.

**QueryString()**

Returns the queryString. Empty string if not found.

**QueryString(String name)**

Returns a comma delimited String of values with the given name. Returns empty string if no value of the given name was found. This function is equivalent to a call to *Form(String name)*.

**QueryStringArray(String name)**

Returns the values of the specified parameter for the request as an array of strings, or an empty String[] if no value of the specified name can be found. This function is equivalent to a call to *FormArray(String name)*.

**ServerVariables()**

A method that returns all header names in an array. Returns an empty string array if no headers were found.

**ServerVariables(String header)**

A method that returns the header value of the given header. Returns an empty String if the given header was not found.

**setAttribute(String name, Object value)**

A method that allows setting of request scope attributes. The parameter *name* is the name used to retrieve the stored object. This method is very useful in passing objects from page to page. For example, an object can be stored in the request prior to a CallPage. The called page can then retrieve the object by its stored name by using the `getAttribute` method.

**getAttribute(String name)**

A method that allows retrieval of request scope objects. The parameter *name* is the name of the object to fetch. The `getAttribute()` method will return null if there is no object of the given name in the request object.

## Response/response Object

The **Response** object sends data to the client. The **Response** object is of type **com.livesoftware.jrun.plugins.jsp.JSPResponse**. The following definitions are a subset of methods defined in the `JSPRequest` class. The `JSPResponse` class subclasses **HttpServletResponse**. This means all methods of `HttpServletResponse` are also implemented in the `JSPResponse` class. For a complete listing of methods defined in `JSPResponse`, see the docs under `<JSP_InstallDir/docs>` directory.

Syntax

**Response**.*Method(variables...)*

Methods

**charset(String charset)**

Appends the name of the character set (for example, ISO-LATIN-7) to the content-type header in the response object.

**contentType(String contentType)**

Specifies the HTTP content type for the response. If no content type is specified, the default is text/html.

**contentType(String contentType, String charset)**

Specifies the HTTP content type for the response. If no content type is specified, the default is text/html.

**addHeader(String name, String value)**

The `addHeader` method adds an HTML header with a specified value. This method will replace an existing header of the same name.

**sendRedirect(String url)**

The `redirect` method causes the browser to attempt to connect to a different URL.

**sendRedirect(String url, boolean abortExecution)**

When the value of abortExecution is set to true, a JSPAbortException will be thrown which will force the jsp page to discontinue execution after the call to sendRedirect.

**buffer(boolean buffer)**

Indicates whether to buffer page output. When page output is buffered, the server does not send a response to the client until all of the server scripts on the current page have been processed, or until the Flush or End method has been called.

**appendToLog(String logString)**

The appendToLog method adds a string to the end of the Web server log entry for this request. You can call it multiple times in one section of script. Each time the method is called it appends the specified string to the existing entry.

**binaryWrite(byte[] data)**

The binaryWrite method writes the specified information to the current HTTP output without any character conversion. This method is useful for writing non-string information such as binary data required by a custom application.

**clear()**

The clear method erases any buffered HTML output. However, the Clear method only erases the response body; it does not erase response headers. You can use this method to handle error cases

**end()**

The end method causes the Web server to stop processing the script and return the current result. The remaining contents of the file are not processed, this is accomplished by throwing a JSPAbortException.

**flush()**

The flush method sends buffered output immediately. This method will cause a run-time error if Response.Buffer has not been set to TRUE.

**write(String s)**

The write method writes a specified string to the current HTTP output

**Cookies(String key, String subKey, String subValue)**

Adds the subKey/subValue pair to the cookie of the given key name.

**Example:**

```
<% Response.Cookies("mycookie","type1","sugar"); %>  
<% Response.Cookies("mycookie","type2","ginger snap"); %>
```

**Cookies(String name, String value)**

Assigns the value to the cookie of the given name. If this cookie exists the old value will be overwritten. This also applies to cookies set with subkeys and subvalues. If this is the case, then all the subkeys and subvalues of the named cookie will be overwritten with the single given value.

**Example:**

```
<% Response.Cookies("mycookie","milk"); %>
```

**Cookies(String name)**

Returns the Cookie object with the given name. The Cookie's properties can be set using this method.

**Example:**

```
<% Response.Cookies("myCookie").setMaxAge(60*60*24*7*8); %>
```

## Server Object

The **Server** object provides access to methods and properties on the server. Most of these methods and properties serve as utility functions.

Syntax

**Server**.*Method(variables...)*

Methods

**CreateObject(String classname)**

Instantiates an object from the given classname. The package of the object to create must be in the JRun's classes directory, or must be set in JRun's classpath.

**Note:** Although this method is available, it is recommended to use JSP's BEAN tag convention to instantiate an object.

**HTMLEncode(String string)**

Applies HTML encoding to a specified string.

**MapPath(String string)**

Maps the specified relative or virtual path to the corresponding physical directory on the server. This method is available from within the service method.

**URLEncode(String string)**

Applies URL encoding rules, including escape characters, to the string.

## Session Object

You can use the **Session** object to store information needed for a particular user-session.

Variables stored in the Session object are not discarded when the user jumps between pages in the application; instead, these variables persist for the entire user-session.

The JSP automatically creates a Session object when a web page from the application is requested by a user who does not already have a session. The server destroys the Session object when the session expires or is abandoned. The Session object is of type **JSPSession**, the following is a subset of methods defined in JSPSession. For a complete description of defined methods see the api docs in `<JSP_installdir>/docs`.

**Note:** Session state is only maintained for browsers that support cookies.

Syntax

**Session**.*Method(variables...)*

Methods

**getSessionID()**

Returns this session's id.

**putValue(String name, Object object)**

Stores the object with the given name in this session.

**getValue(String name)**

Returns the object stored in this session with the given name

**getValueNames()**

*Returns an array of the names of all the application layer data objects bound into the session.*

**removeValue()**

*Removes the object bound to the given name in the session's application layer data*

**getLastAccessedTime()**

Returns the last time the client sent a request carrying the identifier assigned to the session.

**getCreationTime()**

Returns the time at which this session representation was created, in milliseconds since midnight, January 1, 1970 UTC.

Section

V

# *Appendixes*



## *JSP Troubleshooting Guidelines*

This chapter contains troubleshooting guides for errors that may be displayed by a browser, or within JRun log files. You may also use Live Software's mailing lists and updated FAQs at:



<http://www.livesoftware.com/support/>

### **java.lang.NoClassDefFound and java.lang.ClassNotFoundException**

If your servlet has trouble finding a class, it may report these errors. JRun may display the `java.lang.NoClassDefFound` error message on the HTML page, or the `java.lang.ClassNotFoundException` may be logged in the `JRunServletError.log`.

The possible cause of the `java.lang.NoClassDefFound` error may be that the class wasn't found, or that the servlet engine did not have permission to read the class file, or any of its parent directories.

If you get the `java.lang.ClassNotFoundException` you should be able to see, in the *JRunServletErrorLog*, the exact line on which the program failed, and thus determine the offending class. This error may not occur on your browser, so knowing that you have this error means frequent checking of the logs whenever you have trouble.

One way to watch these errors is to open four shells and run `tail -f JRunXXXXXX.log` in each one of them. There are four log files, hence four shells. The `tail -f` command will basically stream the contents of the file to the terminal window as it comes in, so you can see it. If you are on WindowsNT, you can get the `gnuwin32` tool kit free from [cygnus.com](http://cygnus.com) which will give you `tail` and other useful commands.

### **Document Contains No Data**

This error message is put up by your browser when the servlet has crashed before it sent any information to the output stream ( the browser ). Check the error log files.

### **Whenever I do a redirect within by JSP document, a JSPAbortException is thrown? What is this?**

The **JSPAbortException** is jsp's mechanism for discontinuing execution of a jsp document after a call to `redirect`.

This exception is caught and handled in the JSP system and is not meant to be caught within the jsp document unless you would like to implement any cleanup yourself.

In other words, in most cases, the **JSPAbortException** should be allowed to propagate up passed your jsp document and up to JSP to be handled. JSP's implementation of handling the **JSPAbortException** includes committing any cookie data.

The following try/catch is **NOT** recommended within a JSP document.

```
try
{
    response.sendRedirect( myUrl );
}
catch( Exception e )
{
    System.err.println( this.getClass().getName() + " Exception"
);
    e.printStackTrace( System.err );
}
```

JSPAbortException subclasses IOException which subclasses Exception. The above will **disallow** the JSP system for handling the JSPAbortException.

A simple response.sendRedirect() without the try/catch block is recommended.

```
response.sendRedirect( myUrl );
```

If it is **absolutely necessary** for you to catch an exception out of response.sendRedirect() then you can do the following:

```
try
{
    response.sendRedirect( myUrl );
}
catch( IOException e )
{
    if( e instanceof JSPAbortException ) throw e;

    System.err.println( this.getClass().getName() + " Exception"
);
    e.printStackTrace( System.err );
}
```

## How can I add content from other files into my jsp document?

JRun Server Pages a few ways for accomplishing this. Which convention you use depends on your reasoning for including another document. Such reasons may be:

- **To include logic and/or content from another jsp page.** JRun Server Pages allows the inclusion of content or logic from another document through the following conventions:

```
<%@ include="foo.jsp" %>
```

or

```
<%@ vinclude="/foo.jsp" %>
```

The above tag is considered a pre-processed include. It simply replaces any occurrence of the tag with the contents of the specified file in the jsp document. **One restriction** for using this tag, is that the file being included cannot be a variable or a jsp expression.

For example the following is an example of **incorrect** use.

```
<% String myFile = "foo.jsp"; %>
.
.
.
<%@ include=myFile %>
```

The following is also **incorrect**

```
<%@ include=<%=myFile%> %>
```

The above example will not work because the `<%@ include %>` tag is handled before the jsp servlet is actually ran. The variable 'myFile' only has meaning during the runtime of the servlet generated from the jsp document.

One workaround for the above restriction is to use the `CallPage()` method within the JSP document. **One restriction** to using the `CallPage()` method is that you have to be calling another JSP document. The `CallPage()` method is executed during the runtime of the jsp servlet. This means **variables can be evaluated** for the file name. The `CallPage()` method is used to call an existing JSP document from within another JSP document. The resulting output of the called JSP document will be sent through the same output stream as the calling JSP document. An example follows:

```
<% CallPage("foo.jsp"); %>
```

By using the `CallPage()` method, the following example is **allowable**.

```
<% String myFile="foo.jsp"; %>
.
.
.
<% CallPage(myFile); %>
```

- **To include content from a non-jsp document.**

The following convention allows inclusion of only non-jsp documents, and when the content-type of the jsp document doing the include is the default `run-internet/taglet`.

```
<!--# include file="foo.html" -->
```

or

```
<!--# include virtual="/foo.html" -->
```

The good news about this tag, is that it does allow the use of using variables for the filename. The reason for this is that it is considered a post-processing tag. i.e. the evaluation of the tag is implemented on the output of the jsp servlet. The resulting output is then sent to the client-browser.

## **I don't want my jsp document to be processed for the `<servlet>` tag, `<!--#include>` tag, and user defined SSITaglets. How do I let JRun know to send my jsp documents output strait to the client-browser?**

Specify a content-type within your jsp document to something other than `run-internet/taglet`. See the following example.

```
<%@ content_type="text/html" %>
```

## **Why do JSP documents have a default content-type of `run-internet/taglet`?**

The `run-internet/taglet` content type tells JRun to send output of a JSP document to be processed by the `JSPSSIFilter` for the `<servlet>` tag, `<!--#include>` tag, and possibly user defined SSITaglets. The result of the processed data is then sent to the client-browser.