



# LiteWebServer 3

## JSP Module Reference Manual

*2003-03-31, Version 1.1.1*

<b>INTRODUCTION</b> .....	<b>2</b>
<b>LICENSES</b> .....	<b>2</b>
<b>CONTACT INFORMATION</b> .....	<b>2</b>
<b>FEATURES</b> .....	<b>2</b>
<b>INSTALLATION</b> .....	<b>3</b>
<b>CONFIGURATION OPTIONS</b> .....	<b>3</b>
<b>INSTALLING TAG LIBRARIES</b> .....	<b>5</b>
Globally Shared Tag Libraries.....	5
Auto-Discovered Tag Libraries .....	5
Deployment Descriptor Declared Tag Libraries.....	5
<b>COMPILING JSP FILES</b> .....	<b>6</b>
<b>REMOVING OPTIONAL JAR FILES</b> .....	<b>8</b>

Thank you for choosing LiteWebServer™ (LWS), a small and easy-to-use Java web server with native support for the Servlet API. It's small size (roughly 1 MB or less, depending on Java version and optional features) makes it ideal for embedding in other Java applications, for bundling with web application demos sent to potential customers, and for running web applications distributed on CDs. It's also ideal for Java web application development, personal use or a small intranet.

This Reference Manual describes how to configure and use the JSP Module. Please see the Base Module Reference Manual for how to install, configure and use LiteWebServer in general. For information about how to develop JSP pages, please use the resources listed on Sun's JSP pages, <http://java.sun.com/products/jsp>. We also recommend Hans Bergsten's *JavaServer Pages* (O'Reilly) if you want to learn more about JSP.

## Introduction

LiteWebServer (LWS) is a pure Java web server with native support for the Java Servlet API. It's a modular server, with a base module that contains the main functionality and add-on modules for additional features. The JSP Module enables support for JavaServer Pages (JSP).

## Licenses

LWS is licensed under a BSD-style open source license. Briefly, this means that you can use the product any way you want as long as you keep all copyright notices and include a note about where it comes from if you redistribute LWS with other software. See the *license.txt* file in the installation directory for details.

The JSP Module includes software developed by the Apache Software Foundation (<http://www.apache.org/>). More specifically, it is based on JSP container from the Apache Tomcat server. See the Apache Software License, Version 1.1 (*Tomcat-license.txt*) in the installation directory for licensing terms with respect to the bundled ASF code.

## Contact information

You can reach Gefion Software through one of the following mail addresses.

[info@gefionsoftware.com](mailto:info@gefionsoftware.com) for general questions or comments about the company and the products.

[support@gefionsoftware.com](mailto:support@gefionsoftware.com) to get help or report a problem with a product. Ideas about new features or other improvements are of course also welcome.

[sales@gefionsoftware.com](mailto:sales@gefionsoftware.com) for questions about licenses.

Visit our web site at <http://www.gefionsoftware.com/> for up-to-date information.

## Features

The LiteWebServer JSP Module offers the following main features:

- JavaServer Pages 1.2 (JSP) compliant JSP container.
- Optional tag handler pooling for better performance.
- Command line JSP compiler to compile JSP pages into servlet classes for deployment of JSP-based application without the full JSP container on a platform without a Java compiler

# Installation

The JSP Module must be installed using the JustGetIt module manager application bundled with the LiteWebServer (LWS) Base Module, see the Base Module Reference Manual for details. You must also have a Java 2 Standard Edition (J2SE, at least 1.3 but we recommend that you use the latest version available for your platform) compatible Java VM with a Java compiler installed. If you don't have one already you can download a Java VM from <http://java.sun.com/j2se/downloads.html>. Note that only the SDK version includes the Java compiler.

Installing the JSP Module using the JustGetIt module manager places files in these directories under the LWS home directory (referred to as `$LWS_HOME` in some parts of this document):

*bin*

The JSP Compiler (JSPC) scripts use to precompile JSP files, see *Compiling JSP Files* for details.

*common*

Classes available to both LWS and all web applications.

*etc*

Files with information about the installed modules.

*server*

Classes used internally by LWS. Web applications do not have access to these files and you must not modify its content.

# Configuration Options

The JSP container (the part of LWS that processed JSP pages) provides some configurable features, such as tag handler pooling. The JSP container is implemented as a servlet, so configuration options are set as servlet initialization parameters in a *web.xml* file. To change the configuration for all web applications, you edit the default *web.xml* file located in the `$LWS_HOME/conf` directory. It defines the JSP container servlet like this:

```
...
<servlet>
  <servlet-name>jsp</servlet-name>
  <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
  <init-param>
    <param-name>enablePooling</param-name>
    <param-value>>false</param-value>
  </init-param>
  <init-param>
    <param-name>logVerbosityLevel</param-name>
    <param-value>WARNING</param-value>
  </init-param>
</servlet>
...
```

You can also make configuration changes for an individual web application by copying the JSP container servlet definition to an application's *web.xml* file and set the appropriate init parameters in that file only.

The following table describes all initialization parameters you can set:

Property	Default	Description
checkInterval	300	The time in seconds between checks to see if a page needs to be recompiled when the development parameter is set to false.
compiler	javac	The compiler used to compile JSP

Property	Default	Description
		pages. Only <code>javac</code> is supported in this release.
<code>classdebuginfo</code>	<code>true</code>	If <code>true</code> , the JSP page is compiled with debug info.
<code>classpath</code>	All classes in <code>\$LWS_HOME/common</code> and <code>shared</code> directories plus the <code>WEB-INF/lib</code> and <code>WEB-INF/classes</code> directories for the web application.	A custom classpath to use when compiling JSP pages. Use of this parameter is strongly discouraged.
<code>development</code>	<code>true</code>	If <code>true</code> , JSP pages are recompiled if they have been modified since the last request and <code>reloading</code> is set to <code>true</code> . If <code>false</code> , <code>checkInterval</code> is enabled.
<code>enablePooling</code>	<code>true</code> (but set to <code>false</code> in the default <code>web.xml</code> file).	If <code>true</code> , tag handler instances are pooled and reused according to the rules defined by the JSP 1.2 specification. Note that many existing tag libraries are not implemented correctly to take advantage of this feature.
<code>fork</code>	<code>true</code>	Compile JSP implementation classes in a separate process, to avoid all possible multithreading issues. Setting this property to <code>true</code> improves JSP auto-compilation performance but may cause problems if the server is hit with simultaneous requests for JSP pages that have not been compiled yet.
<code>ieClassId</code>	<code>clsid:8AD9C840-044E-11D1-B3E9-00805F499D93</code>	The <code>class-id</code> value used by <code>&lt;jsp:plugin&gt;</code> for Internet Explorer.
<code>javaEncoding</code>	<code>UTF-8</code>	The character encoding used for the Java files generated from JSP pages.
<code>keepgenerated</code>	<code>true</code>	If <code>true</code> , the generated Java files are kept in the <code>work</code> subdirectory for the application after compile.
<code>largefile</code>	<code>false</code>	If <code>true</code> , the static content of a JSP page is saved in external files to keep the generated Java class smaller.
<code>logVerbosityLevel</code>	<code>WARNING</code>	The level of detailed messages to be written to the log file. One of <code>FATAL</code> , <code>ERROR</code> , <code>WARNING</code> , <code>INFORMATION</code> , <code>DEBUG</code> .
<code>mappedfile</code>	<code>false</code>	If <code>true</code> , each line of static content in a JSP page results in one <code>println()</code> statement in the generated Java class.

Property	Default	Description
reloading	true	If true, JSP pages are recompiled if they have been modified since the last request.
scratchdir	The work directory for the application.	The directory to use for generated Java source and class files.

You must restart LWS after changing the configuration for the changes to take effect.

## Installing Tag Libraries

One of the most powerful JSP features is the support for custom tag libraries. Tag libraries encapsulate common logic, implemented as pure Java classes, and let page authors invoke this logic by adding HTML-like markup *custom action* elements in JSP pages.

Tag libraries are available from a lot of different sources: commercial vendors, open source projects, or in-house development teams. A tag library needs to be installed before it can be used in a JSP page, and LWS offers you the choice of making a library available to all web applications or install it on a per-application basis. In either case, in the JSP pages where you want to use the library, you need to declare it with a JSP directive like this:

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

The `prefix` attribute defines the XML namespace prefix you want to use for the custom action elements in the page, i.e. `<prefix:myAction>`. The `uri` attribute is the unique URI that identifies the library, such as `http://java.sun.com/jstl/core` for the JSP Standard Tag Library (JSTL) core library. How the URI is associated with the library depends on how it's made available to the application, see below for details.

## Globally Shared Tag Libraries

A tag library that is packaged in a JAR file and contains a Tag Library Descriptor (TLD) that defines the tag library URI can be shared by all web applications served by LWS by placing the JAR file (plus all other JAR files it uses, if any) in the `$LWS_HOME/shared/lib` directory.

After restarting LWS, any JSP page can use the tag library just by specifying the tag library URI as the `uri` attribute value in the `taglib` directive.

## Auto-Discovered Tag Libraries

LWS supports auto-discovery of tag libraries, as defined by the JSP 1.2 specification. This means that on startup (or restart), LWS scans all directories under the application's `WEB-INF` directory for Tag Library Descriptor (TLD) files (files with extension `.tld`) and all JAR files under `WEB-INF/lib` that contains one or more TLD files in the `META-INF` structure, and makes all tag libraries with a URI defined in these TLDs available to the application. If a URI found this way is the same as one of the globally shared tag libraries, the application-specific library is used.

After restarting LWS, any JSP page can use the tag library just by specifying the tag library URI as the `uri` attribute value in the `taglib` directive.

## Deployment Descriptor Declared Tag Libraries

If the Tag Library Descriptor (TLD) for a tag library doesn't define a URI, or the URI it defines clashes with another URI, you can declare the tag library with a unique URI in the `web.xml` file for the application instead:

```

<web-app>
...
<taglib>
  <taglib-uri>
    http://mycompany.com/mylib
  </taglib-uri>
  <taglib-location>
    /WEB-INF/lib/mylib.jar
  </taglib-location>
</taglib>
...
</web-app>

```

Note that the `<taglib-uri>` value doesn't have to look like a URL, and if it does, it doesn't have to point to an existing resource; it's just an identifier so any value that is unique within the application is fine. An HTTP URL is just a convention, because it offers some guarantee for uniqueness.

The `<taglib-location>` element value must be a context-relative path to the JAR file that contains the TLD, or to the TLD file itself.

After restarting LWS, any JSP page in the application can use the tag library just by specifying the tag library URI as the `uri` attribute value in the `taglib` directive.

## Compiling JSP Files

Before a web container, such as LiteWebServer, can process a JSP page it must be turned into a Java class and compiled. By default, this happens the first time a JSP page is accessed and every time it's modified thereafter.

In some cases, such as when you want to distribute a web application to third parties or run it in an environment with limited processing, memory and disk resources, it may be better to convert all JSP pages to Java classes on demand. With appropriate mappings in the *web.xml* file, these classes can then be invoked directly in response to a JSP page request. Some advantages are:

- No need for a Java compiler in the target environment.
- Less disk space needed, since only the JSP container runtime classes need to be available in the target environment and the class files for the JSP pages can be packaged in a JAR file.
- Slightly less processing required, since JSP page requests invoke the class file directly instead of passing through the JSP container servlet first.
- The JSP files cannot be modified as easily (this can of course also be a drawback depending on the nature of the application).

The JSP Module includes a command line tool that makes it easy to convert the JSP pages in a web application to Java classes and compile them. It is called the JSP Compiler (JSPC) and is executed as a script located in the *\$LWS\_HOME/bin* directory (*jspc.bat* for Windows, *jspc.sh* for all other operating systems). While it supports many options, this is how it's typically used (Windows syntax; adjust the paths for other operating systems):

```
C:\lws-3.0\bin> jspc -d \dist -webinc \dist\web.inc -webapp ..\webapps\myapp
```

The `-webapp` option specifies the web application root directory. JSPC processes all JSP files found in this directory as well as all subdirectories. The `-d` option specifies the top directory for the output. For each subdirectory in the web application that contains JSP pages, the corresponding subdirectory is created for the generated Java files. The `-webinc` option specifies a file where *web.xml* mapping elements for all JSP pages are generated. These mappings tell the web container to invoke the Java class directly when it receives a JSP request. For instance, these elements say that a request for */foo/bar.jsp* is handled by invoking the generated Java class `foo.bar_jsp`:

```

<servlet>
  <servlet-name>foo.bar_jsp</servlet-name>
  <servlet-class>foo.bar_jsp</servlet-name>
</servlet>
...
<servlet-mapping>
  <servlet-name>foo.bar_jsp</servlet-name>
  <url-pattern>/foo/bar.jsp</url-pattern>
</servlet-mapping>

```

Just include all the servlet and mapping elements in the *web.xml* file for the application. Alternatively, if you don't have any other elements in your *web.xml* file, you can generate a complete *web.xml* file with JSPC by using the `-webxml` option instead of `-webinc`.

The command example above creates Java source files for all JSP pages in the application. If you use a build system, such as Apache Ant, you probably want to use it to compile these Java files. But you can also use JSPC to both generate the Java files and compile them. Just add the `-compile` option:

```
C:\lws-3.0\bin> jspc -compile -d \dist -webinc \dist\web.inc -webapp ..\webapps\myapp
```

When you use this option, *.class* files are created in the same directories as the Java source files.

Here's the formal syntax for the JSPC command:

```
jspc [options] -webapp web-app-root-dir
```

The following table describes the options\*.

Option	Default	Description
<code>-vN</code>	2	Verbosity level. The higher the number, the more information about what's going on is produced. Note that there's no whitespace between the option and the number.
<code>-d output-dir</code>	The directory specified by the <code>java.io.tmpdir</code> system property	The top directory for the generated Java files.
<code>-l</code>	No error messages	List error messages when a JSP page cannot be compiled and continue with the next page.
<code>-s</code>	No success messages	List JSP pages that compile successfully.
<code>-compile</code>	Java classes are not compiled	In addition to generating Java class source files for each JSP page, compile the class as well.
<code>-webinc web-inc-file</code>	No file generated	A file where servlet declarations and mapping are written. This file can then be included in a <i>web.xml</i> file.
<code>-webxml web-xml-file</code>	No file generated	A <i>web.xml</i> file where servlet declarations and mapping are written.

---

\* JSPC actually supports additional options (inherited from the Apache Tomcat code) but they are not documented here because their use is discouraged and they are not verified to work.

With all JSP pages converted to Java classes and compiled, you can package them in a JAR file, place the JAR file in the application's *WEB-INF/lib* directory and remove all JSP files. If you want to distribute the application with a LiteWebServer that is as small as possible, you can exclude the *common/lib/jasper-compiler.jar* file from the distribution.

## Removing Optional JAR Files

If you're running on a J2SE 1.4 (or later) platform, you remove the *dom.jar*, because it's bundled with J2SE starting with the 1.4 version.