jo!
User Guide

1.0

tagtraum industries

Please send all comments to <feedback@tagtraum.com>

tagtraum industries

jo! 1.0 User Guide

Copyright © 1999-2002 by tagtraum industries

tagtraum industries is a project initiated by Hendrik Schreiber

http://www.tagtraum.com/

All trademarks and registered trademarks herein are the property if their respective owners

Content

1 Introduction

In this section of the user guide we would like to make you familiar with some basic concepts.

1.1 What are servlet containers anyway?

Servlet containers are part of the Java 2 Enterprise Edition (J2EE™) platform defined by Sun. In a business application that utilizes J2EE™ it is the servlet container's task to provide a connection from the actual application (business logic) to the web. This is achieved through the use of servlets, JavaServer™ Pages (JSP) and other static or dynamic resources such as databases or simple files. J2EE™ propagates the use of EJB™, JNDI™ etc. in conjunction with servlets and JSP, but leaves the choice to the developer. He/she does not have to use EJB™, but can use whatever technology he/she believes to suite his/her needs. Because of this, servlets are often referred to as the glue between legacy code, new technologies and the web.

Servlet containers (pre J2EE™-speak: servlet engines) are a standardized platform to build this glue on. They have to comply with the Servlet API specified by Sun™ and are the environment to execute servlets, JSP and even entire web applications (WARs) in.

1.2 Who the heck is jo?

jo! is the servlet container you probably just downloaded. It is a web server written entirely in Java that is compatible to Servlet API 2.2.

At first jo! was developed, because Hendrik Schreiber and Peter Roßbach were writing a book about servlet programming (Java Server and Servlets, AWL) and needed a Servlet API 2.1 implementation to experiment with. Because none was available, they decided to write their own and called it jo! It was the first implementation of the brand new API.

In the beginning jo! was small and quite beautifully designed. For example you could exchange many classes just by editing a configuration file. We believe its design is still stylish, but it got bigger, faster and more compatible to specifications. In other words: jo! grew up.

1.3 Why should I use jo? Aren't plugins for my existing high performance server a better solution?

Not necessarily. High performance servers probably beat jo! when you deal with pure static content. But when you need to execute servlets or JSP you have to consider that the server first has to parse the content, then it needs to decide whether it is the plugin's job or not. If it is the plugin's job the request has to be transported (possibly over an expensive TCP connection) to the plugin and the plugin needs to parse the request again. This is quite an effort you do not have to make. If your server is written in pure Java it does not need any plugins.

1.4 Features

- HTTP/1.1 including persistent connections, pipelining, byte ranges, virtual hosts, etc. as defined in RFC 2616

- Memory sensitive file cache

- Basic and Digest authentication as defined in RFC 2617
- SSL
- Servlet API 2.2 as defined by Sun™
- JSP 1.1 support
- Restoring of serializable sessions after shutdown
- Autointernationalization
- Automatic compression of textual data
- Support for archived web applications (WARs)
- Automatic reload of WARs
- Web applications can be pointed at with an URL
- Drop-in/hot-deployment for WARs
- One classloader per web application
- Automatic reloading of servlets and JSP
- Advanced thread management and very good performance
- Partial support for SSI
- Support for the servlet tag <servlet>
- Can be used as web container for Jboss™ and Avalon-Phoenix

1.5 System requirements

jo! has been tested under Win 2k with Sun JDK 1.3. However, it has been reported that jo! works fine under all kinds of systems and in general it is expected that it runs under any Java 2 compliant system. There are scripts provided to run jo! in a Unix environment.

2 Architecture

jo! consists of different parts. First of all there is the web server. The server has at least one listener and at least one host. Each host has one or more web applications or ServletContexts (these are rough synonyms). Each web application consists of servlets, JSP and other static or dynamic resources.

2.1 Server

The server is the actual root object of jo! It can have one or more listeners to accept requests and has one or more hosts to dispatch these requests to.

2.2 Listeners

Listeners are objects that listen to a specific port/address pair and return incoming connections to the server. You can have more than one listener per server in order to listen on multiple ports and multiple IP addresses. This is especially useful when you want to access the same content in different ways, e.g. over a plain connection and a SSL connection. One could also imagine a server with multiple IP addresses. There should be a listener for each address you want your server to listen on.

2.3 Hosts

In order to save IP addresses HTTP supports multiple virtual hosts on servers with only a single IP address. Therefore a server can have multiple hosts, each acting like a single server.

2.4 Web Applications

Web applications or ServletContexts are usually mapped to a host's URI space. E.g. you can map all URIs beginning with `/~user/` to a certain ServletContext. Or all URIs starting with `/catalogue/` to a web application. ServletContexts enable servlets to share data more easily and therefore provide a mean to act as a collection of interacting resources rather than isolated programs.

2.5 Servlets, JSP and other Resources

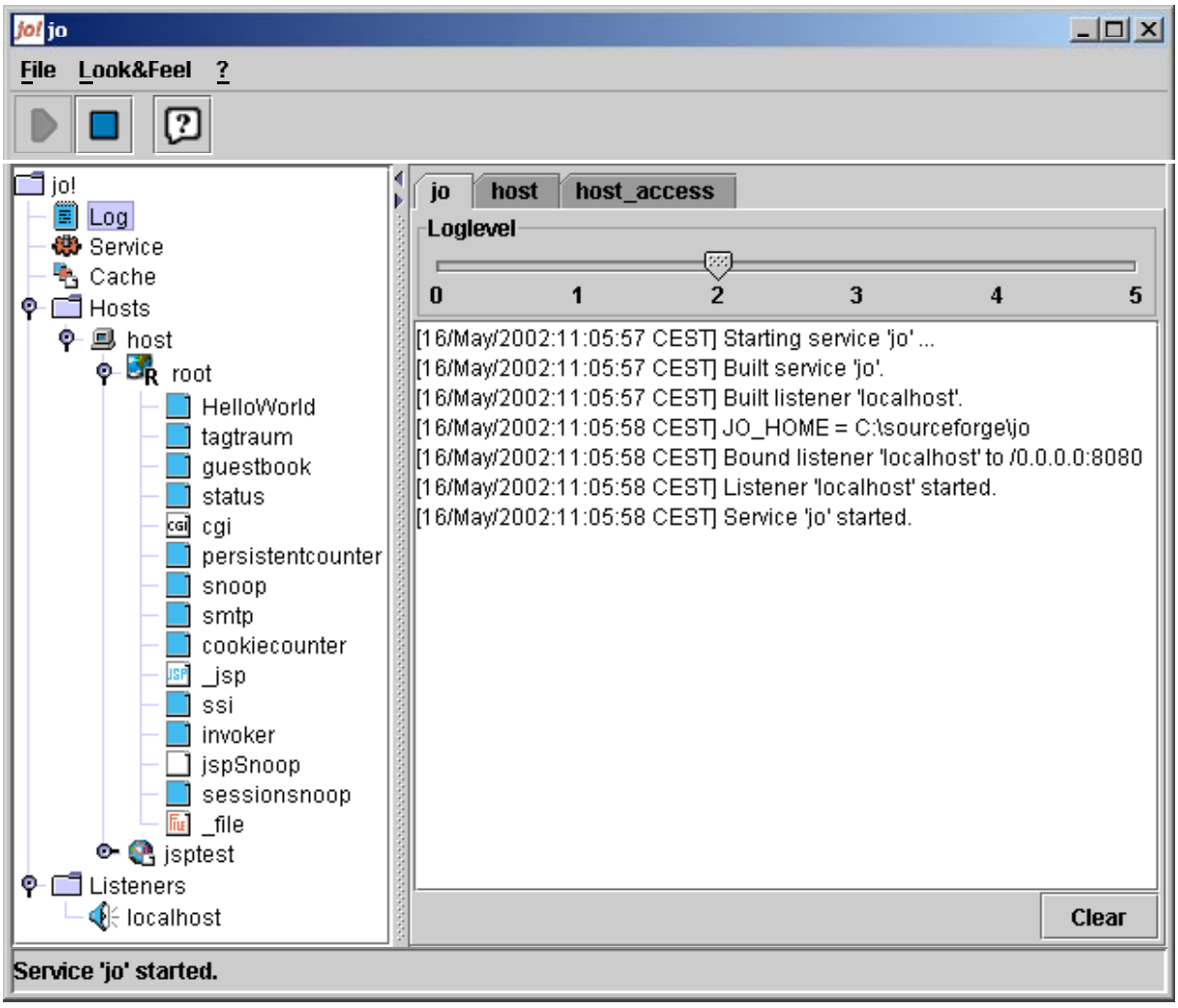
Servlets, JavaServer Pages and other resources are always part of a web application. They can access each other easily. Additionally servlets and JavaServer Pages may use other objects or components to fulfill a task. They often simply request data from such other components and produce some HTML output.

3 Getting started

This chapter will help you make your first steps with jo! Please read it carefully. Most of your question will probably be answered in here.

3.1 Starting jo!

To start jo! change to the jo directory and execute the script `jo`[1]. A Swing™ window will pop up and the server will start. If your system is not connected to a domain name server (DNS) this might take a while, because the system tries to find all IP addresses available on the system.



You should now be able to open the URL http://127.0.0.1:8080/ with your favorite browser.

Note for Windows 95/98/ME users: It is likely that the initial memory for the environment is not big enough. To change that, right-click on `jo.bat`, click `Properties`, choose the tab `Memory` and change the value for `Initial Environment` from `auto` to `1024`. By this a shortcut will be created in the same

[1] All scripts mentioned have either `.sh` or `.bat` as file extension. Please choose the appropriate version for your operating system.

directory. Doubleclick on this shortcut to start jo!.

3.2 Starting without GUI

Starting a Swing™ window has a lot of overhead. Therefore a non-graphical way to start jo! is provided. Simply execute the script `jo_ng` in the jo directory. In order to stop jo!, execute the script `stopjo` in the same directory.

3.3 Starting jo! as MBean

If you have a MBean server available you can start jo! as a managed bean. For this purpose the class `com.tagtraum.jo.Jo` and the interface `com.tagtraum.jo.JoMBean` are provided. They allow you to initialize, start, stop, and destroy the server.

If you want to use this feature you need to make sure that the VM is started with the extra parameter `JO_HOME` set to the jo! home directory, e.g.:

```
java -DJO_HOME=c:\jo1.0 -classpath …
```

The free and pretty cool application server JBoss (http://www.jboss.org/) uses JMX/MBeans extensively. For further information on how to use jo! with JBoss see section 14.2.

3.4 Publishing static content

In order to publish, simply place your files in the directory `webapp/host/root/`. They are now accessible through the server as part of the default ServletContext.

As any server jo! marks resources it delivers with a content type. To help the server setting the right content type, you might have to edit the file `mime.properties` in the jo directory `etc`. It defines a mapping from file extensions such as `*.htm` to mime types such as `text/html`.

3.5 Installing a servlet

To install a servlet you have to place your classes into `JO_HOME/webapp/host/root/WEB-INF/classes/` and/or your jars into `JO_HOME/webapp/host/root/WEB-INF/lib/`.

Then edit the file `web.xml`[2] in `JO_HOME/webapp/host/root/WEB-INF`. Look for the `<web-app></web-app>` section and insert lines similar to these:

```
<servlet>
 <servlet-name>
      HelloWorld
 </servlet-name>
 <servlet-class>
      com.tagtraum.jo.servlets.HelloWorld
 </servlet-class>
</servlet>
```

[2] You might want to use an XML editor for editing the XML files. An overview of editors can be found at http://www.xmlsoftware.com/editors/

The servlet class has to be either in the `CLASSPATH`, in `webapp/host/root/WEB-INF/classes/` or in `webapp/host/root/WEB-INF/lib/foo.jar`. Note that classes in `CLASSPATH` will not be reloaded when jo! is restarted while the other ones will. Also note, that in order to actually execute the servlet, you have to provide a mapping between a URI and a servlet. How this is done is explained in sections 3.6 and 4.4.

3.5.1 Initialization parameters

In order to initialize a servlet you can pass parameters to it. These are specified through the `<init-param>`-tag:

```
<servlet>
…
 <init-param>
       <param-name>
            aParameterName
       </param-name>
       <param-value>
            aParameterValue
       </param-value>
 </init-param>
</servlet>
```

You can specify multiple `<init-param>`-tags per servlet.

3.5.2 Preloading a servlet

If you want the container to instantiate a servlet at startup, you can add the `<load-on-startup>`-tag to the `<servlet>`-tag:

```
<servlet>
 …
 <load-on-startup/>
</servlet>
```

By including a positive number in the `<load-on-startup>`-tag you can influence the order in which servlets are loaded.

3.5.3 Automapping

This option has been removed in favor of 4.4

3.5.4 Remote servlet loading

This feature is no longer supported, because it inherently produces `ClassCastExceptions`.

3.5.5 Instance pooling

If a servlet implements the interface `SingleThreadModel` it is not threadsafe and jo! automatically builds a pool of instances. The default maximum number of instances is 10.

3.6 Mapping a URI to a servlet

After you registered the servlet you still have to tell the server how this servlet should be mapped to a URL or URI. I.e. if you want to execute a certain servlet with the URL `http://127.0.0.1:8080/MyServlet` you will have to define a mapping.

In order to do so you have to edit the file `web.xml` in `webapp/host/root/WEB-INF`. Within the `<web-app></web-app>`-tags this file contains `<servlet-mapping>`-tags, which in turn contain a `<servlet-name>` and a `<url-pattern>` tag. There are four kinds of url-patterns:

1. Exact match
2. Prefix match
3. Suffix match
4. Default match

An exact match pattern simply starts with a `'/'`. A prefix pattern starts with a `'/'` and ends with a `'/*'`. Suffixes start with `'*.'` and the default pattern is a simple `'/'`.

When a request is accepted, the servlet container first tries to find an exact match, then a prefix match, followed by a suffix match. If everything fails, the default match is used. By default the default match is initialized with jo!'s file servlet.

The example shows several possible mappings:

```
<servlet-mapping>
 <servlet-name>
      HelloWorld
 </servlet-name>
 <!-- Exact Path Mapping -->
 <url-pattern>
      /HelloWorld
 </url-pattern>
</servlet-mapping>
<servlet-mapping>
 <servlet-name>
      EcommerceServlet
 </servlet-name>
 <!-- Prefix Mapping -->
 <url-pattern>
      /catalogue/*
 </url-pattern>
</servlet-mapping>
<servlet-mapping>
 <servlet-name>
      JSPServlet
 </servlet-name>
 <!-- Suffix Mapping -->
 <url-pattern>
      *.jsp
 </url-pattern>
</servlet-mapping>
<servlet-mapping>
 <servlet-name>
```

```
        VerySpecialFileServlet
    </servlet-name>
    <!-- Default Mapping -->
    <url-pattern>
        /
    </url-pattern>
</servlet-mapping>
```

You should now be able to map a servlet to a URI.

Note that for rapid development you can also use the Invoker servlet (see 11.3).

4 Web Applications

jo! realizes the concept of web applications defined in Servlet API 2.2. i.e. you can put your applications in jars and manipulate their behavior through configuration files.

4.1 Webapp Structure

To learn about the structure, simply look at the folder `webapp/host/root/` – it contains the default web application. In it you find regular files and a special folder called `WEB-INF`. In this folder all classes, configuration files and other resource reside. jo! expects to find two files. These are called `web.xml` and `jo.xml`. While `web.xml` needs to be present, `jo.xml` does not. Additionally you can add two folders called `lib` and `classes`. `lib` might contain jars, `classes` should contain classes. Note that classes located in these two folders are loaded with a web application specific classloader.

Please take a look at the servlet spec 2.2 (http://java.sun.com/products/servlet/download.html#specs) to learn about the format of `web.xml`. Note that several J2EE related tags are not supported. These are: `<resource-ref>`, `<env-entry>` and `<ejb-ref>`.

4.2 web.xml

jo! supports the automatic reloading of WARs, if they come in a single file and are accessible through the local system.

4.3 jo.xml

Additionally to the parameters defined in `web.xml` you can set some jo! specific parameters in the file `jo.xml` located in the same folder as `web.xml`. The *current* DTD (http://www.tagtraum.com/dtds/jo-web-app_1.0.dtd) of the file looks like this:

```
<!-- The jo-web-app element is the root of the deployment
descriptor for a web application -->

<!ELEMENT jo-web-app (auto-internationalization?, auto-
mapping?, auto-reload-servlets?, file-transfer-compression?,
show-index?, jsp-config?, restore-sessions)>

<!-- If this tag is present jo! will try to auto-
internationalize requests. I.e. if there is a request for
index.html and the preferred language of the user is German,
it will look for the file index_de.html and deliver it, if
it exists. -->

<!ELEMENT auto-internationalization EMPTY>

<!-- Indicates that servlets and JSP should be automatically
reloaded whenever their class/source files change. Note that
this seriously decreases performance. -->

<!ELEMENT auto-reload-servlets EMPTY>

<!-- Indicates that all servlets should also be mapped to
the stated directory/servletname. E.g.: "/servlets/" -->
```

<!ELEMENT auto-mapping (#PCDATA)>

<!-- Controls the compression of static files. -->

<!ELEMENT file-transfer-compression (algorithm, repository, extension+)>

<!-- States the algorithm to use. Currently the only legal value is gzip. -->

<!ELEMENT algorithm (#PCDATA)>

<!-- Path to the repository to store compressed files in. -->

<!ELEMENT repository (#PCDATA)>

<!-- Files with the extension should be compressed. E.g.: txt -->

<!ELEMENT extension (#PCDATA)>

<!-- Sets whether a table of contents for a directoty should be shown to the user or not. This feature only has an effect if the web-app is local and not in a jar. -->

<!ELEMENT show-index EMPTY>

<!-- Flag indicating that session should be serialized and saved to disc when the webapp is shut down and loaded when it is restarted. -->

<!ELEMENT restore-sessions EMPTY>

<!-- Controls the behavior of the jsp engine for this web application. -->

<!ELEMENT jsp-config (compiler-command?, scratch-dir, send-error-to-client?, generators?)>

<!-- Sets the compiler command to use. %scratchdir%, %classpath% and %encoding% will be replaced by the engine. -->

<!ELEMENT compiler-command (#PCDATA)>

<!-- Name of the directory with source and class files of the jsp in. -->

<!ELEMENT scratch-dir (#PCDATA)>

<!-- Flag indicating that detailed error messages including stacktraces should be sent to the client. -->

<!ELEMENT send-error-to-client EMPTY>

<!-- Registers Sourcecode Generators for JSP -->

```
<!ELEMENT generators (language?, classname)>

<!-- Name of a language for which to generate code. Defaults
to java. -->

<!ELEMENT language (#PCDATA)>

<!-- Fully qualified name of a class -->

<!ELEMENT classname (#PCDATA)>
```

4.4 Automapping

With the tag `<auto-mapping>` you can make sure that all servlets are automatically mapped to a given path like `/servlet/<servletname>` or `/servlets/<servletname>`. To do so you have to set the element <auto-mapping> to the desired path, e.g.: `/servlet/`. Basically the parameter will serve as a prefix to the servletname. To map servlets to a path like `/invoker/<classname>` see 11.3.

Note that jo! controls access to resources based on URL patterns. So don't forget to protect `/servlet(s)/...`-like URLs if necessary.

4.5 Autoreloading

With the tag `<auto-reload-servlets/>` you can indicate that you want the engine to check whether class files of servlets or source files of JSPs have changed and need to be reloaded. If the tag is present, servlets and JSP are automatically reloaded.

Note that this means that the classloader for a specific servlet is removed and a new classloader is instantiated. Within a webapplication this can lead to `ClassCastExceptions`. This will not happen when the whole application is restarted, e.g. when a jar is exchanged or the restart button in the GUI is hit. Also note that auto-reloading decreases the performance significantly.

4.6 Compressed transport

There is a mechanism in jo! which lets you automatically compress static files before transfer, if the user agent is capable of accepting compressed data. Especially when you need to transport a lot of textual data like HTML or ASCII texts it saves quite a bit of bandwidth. To benefit from this mechanism you have to specify a list of file extensions to specify the kind of files you wish to compress before transfer. This is done via the tag `<file-transfer-compression>`. Because compressing is usually a very expensive task, jo! stores compressed versions of your files in a repository. To specify the location of this repository use the tag `<repository>` inside of `<file-transfer-compression>`. To specify the file types you want to compress, you can use (multiple) `<extension>`-tags. You can also specify multiple algorithms to use. For now you must specify `gzip`.

Example:

```
<file-transfer-compression>
 <repository>./compressed_files</repository>
 <algorithm>gzip</algorithm>
```

```
  <extension>txt</extension>
</file-transfer-compression>
```

Netscape Browsers (<=4.7x) do not seem to load linked-in stylesheets when using compressed transport.

4.7 Automatic internationalization

Todays websites often have to live in different languages. Therefore jo! supports a mechanism that automatically chooses the most suitable version of a static file or a JSP for a given request. To profit from autointernationalization you have to place different versions of your resource in each folder. If `welcome_en.html` contained an English version of a welcome file, `welcome_de.html` should contain the German version. Additionally the file `welcome.html` must contain a default version. The same pattern applies to all other resources. Note that this pattern corresponds to the one used in `java.util.ResourceBundle`. The language codes are taken from ISO 639 (http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt).

To turn this feature on or off, please set the `<auto-internationalization/>` tag.

In case you wonder how it is done – it is simple. Most user agents, i.e. browsers, send a header called `Accept-Language` when requesting a document. This header is analyzed, the appropriate file is searched for and delivered.

4.8 Showindex

If you want jo! to automatically show a table of contents of a folder, you have to set the `<show-index/>` tag.

4.9 Restore Sessions

If `<restore-sessions/>` is present, the webapp tries to automatically save all sessions into a file when it is shut down. This file is then loaded when the webapp is restarted. Note that sessions and their contents must be serializable for this feature to work.

4.10 Archives

Web applications can be put into a single archive called WAR (**W**eb Application **AR**chive). To build this archive you should use the jar tool provided by Sun™. Another great way to build WARs is the war task in the java based build tool Ant[3].

4.11 Deploying a WAR

There are two ways to deploy a WAR with jo! – an easy one and a more flexible one.

Either you create a directory `<JO_HOME>/webapp/<hostname>/` and drop the WAR file right in there. For deployment you don't even have to restart the server. It will be reloaded every time it is replaced and you can even manipulate the extracted files you will find in the same directory once you have started jo! A webapplication `MyApp.war` will be accessible

[3] http://jakarta.apache.org/ant/index.html

under `http://127.0.0.1:8080/MyApp/`, i.e. the mapping is `/MyApp`. Note that a WAR called `root.war` will be mapped to `http://127.0.0.1:8080/`, i.e. the root of the URI space.

Or you can edit the file `hosts.properties` (see also section 8) in folder `etc`. Just add the following two lines:

```
<hostname>.webapp.<webappname>.mapping=<URI prefix not
ending with '/'>
<hostname>.webapp.<webappname>.docbase=<URL>
```

The first line defines where in the URI space of the server the web application should be located. It is important that this URI prefix does not end with a `'/'`. The second line specifies where the actual files are located. Here you can set a URL pointing at a WAR or a folder. Note that if you want to point at a folder, you need to add a trailing slash. Also please note that under windows an absolute URL pointing at a local file has the following form: `file:/<drive>:/[path/]<file>`

5 JSP Support

jo! supports JSP 1.1 including tag libraries. You can find some examples for JSPs in the `webapp/host/root/examples/` folder. An example tag library is provided in the package `com.tagtraum.taglib`.

For details about how to develop with JSP please take a look at e.g. http://directory.google.com/Top/Computers/Programming/Internet/JSP/Tutorials/ and http://java.sun.com/products/jsp/. We also recommend taking a look at Struts (http://jakarta.apache.org/struts/).

As the support is built in, you don't need to install a specific servlet. Note that the JSP engine is configured with the entries in the file `jo.xml` (see 4.2). A typical configuration looks like this:

```
<jsp-config>
 <scratch-dir>jsp_scratch_dir</scratch-dir>
 <send-error-to-client/>
</jsp-config>
```

5.1 Java compiler

The JSP engine will automatically detect the Sun compilers `sun.tools.javac.Main` (classic) and `com.sun.tools.javac.Main` (modern). To use these compilers you should add the archive `JAVA_HOME/lib/tools.jar` to your classpath. This archive is only included in the JDK distribution of Java, but not in the JRE.

Additionally, jo! will search in your current path for the fast IBM compiler `jikes`. If you have trouble using `jikes` you should add the archive `JAVA_HOME/jre/lib/rt.jar` to your classpath.

If the JSP servlet neither finds the modern or the classic Sun compilers nor `jikes` it will look for `javac` in your path. If no compiler is found at all, an exception is thrown.

Please note that jo! uses `jikes` over any other compiler even if it finds a compiler by Sun, because it seems to be the fastest compiler available today.

If your compiler is not matched by the detection process or you want to override it, you can specify a compiler by adding the `<compiler-command>` tag. The strings `%encoding%`, `%classpath%`, `%source%` and `%outputdir%` will be substituted by the engine.

Example:

```
<jsp-config>
 <compiler-command>/myPath/mySpecialJavaCompiler -cp
%classpath% -output %outputdir% -enc %encoding%
%source%</compiler-command>
</jsp-config>
```

Note that some versions of `jikes` do not properly support unicode. Therefore you might run into trouble when you use a non-latin characterset (e.g. Shift_JIS) in your JSP. Please check the `jikes` homepage (http://oss.software.ibm.com/developerworks/opensource/jikes/project/) for the latest version of `jikes`. `jikes` also does not seem to support spaces in pathnames. If you

experience `VerifyErrors` while using `jikes`, consider getting a newer version (>=1.15) or switch to another compiler. However, jo! will automatically try to use a different compiler, if `VerifyErrors` are encountered. You will find a corresponding note in the log file.

If you are unsure about which compiler jo! uses, please check the log file.

5.2 Autoreloading

The JSP engine supports autoreloading of JSP. Whenever the source file or one of the included files are changed the JSP will be recompiled. This feature is not activated by default. To turn it on you have to set the tag <`auto-reload-servlets/>` in `jo.xml`. Note that turning autoreloading on can significantly decrease the performance, as the server has to access the filesystem quite often. See 4.5

5.3 Send errors to client

To send detailed error messages regarding JSP to the client you can set the tag <`send-error-to-client/>`. This includes stacktraces and should be turned off in a production environment. You will find the same messages including stacktraces in the log.

5.4 Code-Generators

With this tag you can register your own code generator for JSP. A code generator class has to implement the interface `com.tagtraum.jo.jsp.JspGenerator` and has to have a no-arg constructor. The default implementation is `com.tagtraum.jo.jsp.JspJavaGenerator`.

6 Server side includes (SSI) and the servlet tag

jo! partially supports SSI and fully supports the servlet tag `<servlet>`. Please note that these are proprietary extensions of the Java Web Server (now iPlanet™). They will not be developed any further (except for bug fixes).

6.1 Installation

SSI and the servlet tag are implemented by the servlet `com.tagtraum.jo.ssi.SSIServlet`. If you want to add SSI support to a web application you have to install this servlet. For your convenience this is already done in the default configuration. The entry in `web.xml` should look like this:

```
<servlet>
 <servlet-name>
      ssi
 </servlet-name>
 <servlet-class>
      com.tagtraum.jo.ssi.SSIServlet
 </servlet-class>
 <load-on-startup/>
 <init-param>
      <param-name>
           buffersize
      </param-name>
      <param-value>
           4096
      </param-value>
 </init-param>
 <init-param>
      <param-name>
           expire
      </param-name>
      <param-value>
           0
      </param-value>
 </init-param>
 <init-param>
      <param-name>
           autointernationalization
      </param-name>
      <param-value>
           true
      </param-value>
 </init-param>
</servlet>
<servlet-mapping>
 <servlet-name>
      ssi
 </servlet-name>
 <url-pattern>
      *.shtml
 </url-pattern>
</servlet-mapping>
```

Just as the web application the SSIServlet supports automatic internationalization. The argument `expire` refers to the expire header of HTTP. With this initialization-parameter you can set a time in seconds your page should be considered as fresh by other proxies and caches. Finally the `buffersize` argument lets you set a buffersize used for output.

Note that the SSIServlet uses a memory sensitive filecache.

6.2 SSI

SSI commands have the following format:

```
<!--#command tag1="value1" tag2="value2" ... tagN="valueN" -
->
```

jo! currently supports the following commands:

- **config** configure SSI for a request
- **include** insert the contents of another file into the response
- **echo** insert the value of a variable into the response
- **fsize** insert the size of a resource into the response
- **random** generate a random number for a request

Each of the commands is described in detail below.

6.2.1 Config

The config command is used to configure SSI processing for a request. Currently only the size format can be specified.

This is done with the attribute `sizefmt`. The possible values are `abbrev` and `bytes` (default). `abbrev` causes the file size to be printed in kb while the `bytes` causes the file size to be output in bytes. E.g.:

```
<!--#config sizefmt="abbrev" -->
```

6.2.2 Include

With the include command you can insert a resource into your response page. This mechanism is not limited to static files, but can also be used for dynamic content such as servlets, SSI or JSP. It is comparable with the RequestDispatcher of the servlet API. A resource can be included by specifying either its virtual path or its path relative to the current document. A virtual path must begin with a '/' while a file path can not begin with a '/'.

Example:

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

or

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

6.2.3 Echo

The echo command is used to print the value of a variable. For example, to insert the server software name the following syntax would be used:

```
<!--#echo var="server_software" -->
```

The following variables are currently supported:

- **auth_type** the authentication type (Basic or Digest)
- **content_length** content length of data for POST and PUT requests
- **content_type** content type of data for POST and PUT requests
- **document_name** the current document name
- **document_uri** the virtual path to this document
- **http_xxx** HTTP header value (e.g., http_accept)
- **path_info** extra path information
- **path_translated** virtual-to-physical translation of path info
- **query_string** escaped query string
- **query_string_unescaped** unescaped query string
- **random** the current random value for the request
- **remote_addr** remote's IP address
- **remote_host** remote's host name
- **remote_user** the user name sent to be authenticated
- **request_method** the request method (i.e., GET or POST)
- **server_name** the server's host name
- **server_port** the port the server is listening on
- **server_protocol** name and version of the used protocol
- **server_software** name and version of server software

6.2.4 Fsize

The fsize command is used to print size of a file into the response page. A file can be specified using the virtual and file tags as described for the include command.

Example:

```
<!--#fsize file="somefile.html" -->
```

6.2.5 Random

The random command is used to generate a new random number for a request.

Syntax:

```
<!--#random -->
```

6.3 Servlet tag

You can find out about the servlet tag in the Java Web Server documentation:

http://www.sun.com/software/jwebserver/techinfo/doc/jws20_doc.zip

7 Security

jo! offers basic and digest authentication as defined in RFC 2617 and form based authentication as defined in the servlet spec 2.2. Please note that form based and basic authentication are not secure and digest authentication, which is a bit more secure, is not supported by all browsers. If you use either one exclusively over a secure protocol such as HTTPS (HTTP over SSL – see Chapter 10 for details on configuring SSL) they are considered to be secure.

However, in order to use authentication you have to define what to protect by editing the file `webapp/host/root/WEB-INF/web.xml`.

Please note that any roles referred to in the servlet or jsp code must be registered with `<security-role-ref>`s in the `<servlet>` section pointing at a `<security-role>` defined within the `<web-app>` container. I.e.: `<security-role-ref>` points at `<security-role>` and `<security-role>` points at the actual names defined in `roles.properties` (see 7.2).

7.1 Example for a deployment descriptor using security

How the security features used in the `web.xml` work exactly is described in the DTD (http://java.sun.com/j2ee/dtds/web-app_2_2.dtd) and the servlet spec 2.2 (http://java.sun.com/products/servlet/download.html#specs). To give you an impression on how it works, you might want to take a look at the following example.

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.2//EN" "http://java.sun.com/j2ee/dtds/web-
app_2_2.dtd">

<web-app>
 <!-- define a security role -->
 <security-role>
      <role-name>admin</role-name>
 </security-role>
 <servlet>
      <servlet-name>adminservlet</servlet-name>
      <servlet-class>
           com.tagtraum.AdminServlet
      </servlet-class>
      <security-role-ref>
           <!-- role name used in code -->
           <role-name>ADM</role-name>
           <!-- reference to the role defined in this
                descriptor (above) -->
           <role-link>admin</role-link>
      </security-role-ref>
 </servlet>
 <servlet-mapping>
      <servlet-name>adminservlet</servlet-name>
      <url-pattern>/adminapp/*</url-pattern>
 </servlet-mapping>
 <!-- Use HTTP BASIC authentication for login -->
 <login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>basicRealm</realm-name>
```

```
    </login-config>
    <security-constraint>
          <web-resource-collection>
                <web-resource-name>
                      SecureAdminExample
                </web-resource-name>
                      <!-- restrict access to URIs matched
                            by uri-patterns to certain
                            http-methods and roles -->
                <url-pattern>/adminapp/*</url-pattern>
                <http-method>GET</http-method>
                <http-method>POST</http-method>
                <auth-constraint>
                      <!-- reference to the role defined in
                            this descriptor (above) -->
                      <role-name>admin</role-name>
                </auth-constraint>
          </web-resource-collection>
     </security-constraint>
   </web-app>
```

The deployment descriptor defines a servlet named `adminservlet` and maps it to the URIs starting with `/adminapp/`. It also defines that, if the servlet calls the method `request.isUserInRole("ADM")`, the hardcoded rolename should be mapped to the rolename `admin`. This role has been registered with the tag `<security-role>`. This indirection is necessary in order to use hardcoded rolenames in your sourcecode, which then can be mapped to any role in your deployment environment.

The deployment descriptor also makes sure that all requests for resources, whose URI start with `/adminapp/`, have to be either `GET` or `POST` requests and the requestor has to be in the role `admin`.

7.2 Roles, users and groups

jo! uses a role based authentication scheme. This means that you have to map users and groups to roles. These roles are the basis for restricting and providing access to a resource. The advantage is that a user does not have to re-authenticate himself each time he enters a new application on the same server. The server knows who he is – the only question is, whether the user is in a certain role or not.

To define roles you have to edit the file `roles.properties` in `etc`. But before you do that you should register some users. This can be done with the script called `user`.

Please be aware of the fact that this script is a better-than-nothing solution – it is commandline based and **does not hide passwords** as you type them.

However, with the script you can manipulate the file `users.properties` in folder `etc`. You can add and modify entries and provide passwords. The passwords are not stored in clear text, but only as hash values. This means, that the passwords are only recoverable with a brute force attack.

Now that you have defined some users, you can group them together by editing the file `groups.properties` in `etc`. The entries should look like this:

```
<groupname>=<comma delimited list of groups and/or users>
```

Note that groups and users share the same namespace. This means that a user cannot have the same name as a group.

To define a role, proceed the same way as with groups. Just edit `roles.properties` in `etc`:

```
<rolename>=<comma delimited list of groups and/or users>
```

7.3 Using JAAS

For authenticating users jo! can use JAAS (Java Authentication and Authorization Service – http://java.sun.com/products/jaas). JAAS is an optional extension for JDK 1.3.x and has been incorporated into JDK 1.4. However, this version of jo! comes bundled with the extension (`JO_HOME/lib/jaas.jar` and the sample JndiLoginModule `JO_HOME/lib/jaasmod.jar`).

In order to use JAAS you have to do some configuring.

First edit the file `JO_HOME/etc/factory.properties`. You have to replace the line

```
AccessController=com.tagtraum.jo.security.JoFileUserManager
```

with the line (written as *one* line)

```
AccessController=
com.tagtraum.jo.security.JoJAASAccessController
```

Then edit the file `JO_HOME/etc/sample_jaas.config`. You have to set the classname of the JAAS loginmodule you want to use. Depending on the module, you might have to specify some parameters. E.g. for the sample JndiLoginModule provided by Sun™ (which is included in this distribution) the file might look like this:

```
jo {

   com.sun.security.auth.module.JndiLoginModule
      required
      debug=true
      user.provider.url="ldap://hostname/LDAPName"
      group.provider.url="ldap://hostname/LDAPName";

};
```

Where `hostname` could be `ldap.tagtraum.com` and `LDAPName` `ou=People,o=tagtraum,c=DE` or `ou=Groups,o=tagtraum,c=DE`, respectively.

Now you have to tell JAAS where to look for the JAAS configuration file. This can be achieved by adding the following parameter to the startup script (`jo` or `jo_ng`):

```
-Djava.security.auth.login.config=<your JAAS config file>
```

You have to place it after the classpath and before the main class.

Please note that the provided JAAS based authentication is only an example. It is not an industry strength implementation for the following reasons:

- All Principals are cached - i.e. this implementation is not suited for *millions* of users.
- Groups are not supported by this implementation.

Please also note that the NTLoginModule and SolarisLoginModule provided by Sun as examples for the JAAS 1.0 release are not suited for authentication for users for your site, as they only provide information about the user who is running jo!, but not about any other users.

7.4 Howto write your own Authentication/Authorization

To write an authentication module for users only is pretty simple. Just write a class that implements the interface `com.tagtraum.jo.security.I_JoAccessController`.

```
public interface I_JoAccessController {

   public I_JoServletService getService();

   public void setService(I_JoServletService service);

   public Principal getPrincipal(String username);

   public byte[] getHash(String username, String realm);

   public char[] getHexHash(String username, String realm);

   public boolean isValid(String username, String password);

}
```

Please note that the methods `getHash()` and `getHexHash()` are only needed for (the unpopular) Http Digest Authentication. Therefore they are optional. There are some sample implementations of this interface:

- `com.tagtraum.jo.security.JoFileUserManager`
- `com.tagtraum.jo.security.JoJAASAccessController`
- `com.tagtraum.jo.jboss.JoAuthentication`

Only the first class builds an internal hierarchy of users and groups. I.e. `getPrincipal()` does not only return users, but also groups (i.e. they share the same namespace). For a real world implementation you might want to make sure, that you do the same. The classes `JoGroup` and `JoPrincipal` might come in handy, if the number of users isn't too high and you can keep all of them in memory.

However, this is only the authentication part for users and groups. Probably you also want to assign users and groups to roles. To do so, you have to implement the interface `com.tagtraum.jo.security.I_JoRoleManager`.

```
public interface I_JoRoleManager {

   public I_JoServletService getService();

   public void setService(I_JoServletService service);
```

```
   public String[] getRoleNames(String name,
I_JoAuthConstraint authConstraint);

   public boolean isUserInRole(String name, String roleName,
I_JoAuthConstraint authConstraint);

}
```

As an example this interface is implemented in the class `com.tagtraum.jo.security.JoFileRoleManager`.

Both role methods are called in a context, when a user wants to access a resource that is protected. But only, if the user is in a certain role, he or she is allowed to access the resource. The provided `I_JoAuthConstraint` object tells you, which roles the user must have for a requested resource. For your implementation you might want to use the class `JoRole`.

Both I_JoRoleManager and I_JoAccessController need to use each other. They can do so by calling the methods `getService().getRoleManager()` and `getService.getAccessController()`.

To install your newly written classes you have to edit the file `JO_HOME/etc/factory.properties`. Just substitute the values for `AccessController` and `RoleManager` with the names of your classes and restart the server. That's it.

7.5 General security

In order to make jo! secure, please remove all web application not provided by yourself. There is no guarantee that example or test applications are secure!

If you don't need the cgi servlet, disable it.

The invoker servlet is a natural security hazard. If you don't need it, disable it.

Please note that there is absolutely no guarantee that jo! is secure. tagtraum industries will not be liable for any damages caused by security holes. If you become aware of any security issues, please notify tagtraum industries immediately. Thanks.

8 Virtual Hosts

jo! supports the concept of virtual or vanity hosts as defined in RFC 2616. These hosts are hosts identified by the header field `Host` rather than the URL. This allows multiple virtual hosts per physical server.

In order to register a host you have to edit the file `hosts.properties` in `etc`. It has the following syntax:

```
<hostname>.hostnames=*|<hostname>[, <hostname>, ...]
<hostname>.eventlog=<URL to local file>|<filename>|NONE
<hostname>.eventloglevel=<a positive integer>
<hostname>.eventlogbuffer=<a positive integer>
<hostname>.accesslog=<URL to local file>|<filename>|NONE
<hostname>.accesslogbuffer=<a positive integer>
<hostname>.factory=<URL>|<filename>
<hostname>.webapp.<webappname>.mapping=<URI without trailing
'/'>
<hostname>.webapp.<webappname>.docbase=<URL>|<filename>
```

Each virtual host has an internal name called `<hostname>`. The hostnames the host should be accessible via the web are specified with the parameter `<hostname>.hostnames`. If you would like this host to react to any hostname set the parameter to `'*'`. To install multiple hosts, you need to specify all the entries above for each host, starting with a different `<hostname>`. Note that no more than one host should be mapped to `'*'`.

Each host has two logfiles it writes to: an event log and an access log. Events are exceptions, data logged by servlets and other messages. To take influence on how detailed events are logged you can specify the parameter `<hostname>.eventloglevel`. A value of zero means no logging at all, one means error logging, two module level and three is method level, i.e. the higher the level the more messages you will get. Both logs are buffered. You can set the buffer size with the parameters `<hostname>.eventlogbuffer` and `<hostname>.accesslogbuffer`. Both have default values of 25, i.e. 25 log events, not bytes or kb.

If you set the name of the logfile to `NONE`, no logevents are actualle written to a file.

The access log contains access data in common log format. You can change this by changing the entry `AccessLogEvent` in the file `factory.properties`.

Just like most other parts of jo! the classes in hosts are instantiated through a factory. Each host has a separate one you can configure through a configuration file. This file can be set with `<hostname>.factory`. It is safe to leave this parameter untouched.

The last two parameters should be used to register web applications. Note the at least one application needs to be registered per virtual host. For more information and how to deploy a web application without having to register it manually, please see Chapter 4.

9 Server properties

Of course there also some serverwide features you might want to manipulate. To do so, please edit the file `server.properties` in `etc`. It should look like this:

```
<servername>.name=<servername>
<servername>.log=<local URL or file>|NONE
<servername>.loglevel=0..?
<servername>.logbuffer=0..?
<servername>.stdout=<local filename>
<servername>.stderr=<local filename>
<servername>.maxhandlerthreads=<positive integer>
<servername>.minhandlerthreads=<positive integer>
<servername>.maxrequests=<positive integer>
<servername>.keepalive=<positive integer>
<servername>.so_timeout=<time in ms>
<servername>.handlerclassname=<handlerclassname or alias>
<servername>.majorversion=<positive integer>
<servername>.minorversion=<positive integer>
<servername>.factory=<URL or filename>
<servername>.maxcacheentrysize=<integer>
<servername>.cachecapacity=<integer>
<servername>.strongrefcapacity=10
<servername>.authentificator.BASIC=<handlerclassname or
alias>
<servername>.authentificator.DIGEST=<handlerclassname or
alias>
<servername>.authentificator.FORM=<handlerclassname or
alias>
<servername>.group=<groupname>
<servername>.user=<username>
```

9.1 The server log

The server log contains messages about starting and stopping the server. It does not contain any access logs or host/servlet-specific events. Its location can be set via the `<servername>.log` parameter.

To take influence on how detailed events are logged you can specify the parameter `<servername>.loglevel`. A value of zero means no logging at all, one means error logging, two module level and three is method level, i.e. the higher the level the more messages you will get.

The semantic of log levels might change in future releases.

Besides normal log messages you can re-route standard error and standard output streams to files. The parameters used for this are `<servername>.stdout` and `<servername>.stderr`

9.2 Handlers

As jo! has an advanced thread management you can control the maximum and minimum number of handlerthreads. The more threads you have, the more connections can be handled concurrently. Note that this does not necessarily accelerate your system. The contrary can be the case: requests might be responded to slower, because the system is

busy with switching between threads. On the other hand it is obvious that a single thread doing all the work is not an optimal solution either. In order to find the best value you have to experiment with the system you want to run the server on.

However, values can be set via the `<servername>.maxhandlerthreads` and `<servername>.minhandlerthreads` parameter.

Additionally it is possible to set the classname or a classname alias with the parameter `<servername>.handlerclassname`.

If the server has not handled any requests for a while it can automatically reduce the number of instantiated handlerthreads to a smaller number and eventually to the minimum. The idle time can be set with the parameter `<servername>.so_timeout`. Note that the value has to be set in milliseconds.

9.3 Persistent connections

jo! is able to use persistent connections. As connections are a valuable resource it is desirable to be able to configure jo!'s exact behavior. Therefore you can set the maximum number of acceptable requests per connection with the parameter `<servername>.maxrequests`. You might also set the time jo! keeps a connection open waiting for the next request. This is done via `<servername>.keepalive`. The value must be set in seconds.

9.4 Versions

jo! allows you to set its version. The version number will be sent to user agents as part of the response header. It has no other meaning. You can manipulate the version through `<servername>.majorversion` and `<servername>.minorversion`.

9.5 Factory

Some classes used to fulfill certain roles inside of jo! are instantiated by a configurable factory. You can set the file to use with the parameter `<servername>.factory`.

The most important classes instantiated by the server factory are service (alias: ServletService), host (alias: Host), listener (alias: Listener) and handler (alias: Handler).

In most cases it is safe to simply reference the file `factory.properties` in `etc`.

9.6 File cache

To configure the file cache you can set two parameters: the maximum cache entry size in bytes and the cache's capacity in bytes.

The maximum cache entry size is the maximum size files are allowed to have in order to be added to the cache. To set is use the parameter `<servername>.maxcacheentrysize`. It makes sense to limit this to 100.000 bytes as according to SPEC 99% of the files requested are usually smaller than 100kb. If you don't set a maximum entry size you have to set this parameter to a negative value.

jo!'s file cache is memory sensitive. I.e., it uses as much memory as available to the Java VM. However, if you don't want to rely on the correct implementation of your VM and the quality of the garbage collector, you can set the maximum cache size via the

<servername>.cachecapacity parameter. If you do not want to limit the cache's capacity set this parameter to a negative value.

Also, if your VM tends to collect garbage a lot, the cache might be cleaned sooner than expected. Therefore you can set a number of strongly referenced cache entries. These entries will never be cleared by the garbage collector and behave in a least recently used manner. The parameter to set is <servername>.strongrefcapacity.

Sun VM 1.3.1 and 1.4.x can also be tuned with the parameter -XX:SoftRefLRUPolicyMSPerMB. This parameter regulates, how many milliseconds per free mbyte heap memory the garbage collection shall wait, until the reference is cleared. Note that for the server VM the total possible heap size is used for the calculation, while for the client VM the current heap size is used.

9.7 Authenticators

jo! supports pluggable authenticators you can register by configuration. In order to do so you have to add a line similar to this:

```
<servername>.authenticator.<authmethod>=<classname or class
alias>
```

<authmethod> can be BASIC, DIGEST, FORM or other schemes. It must match the value authmethod set in web.xml. Note that CERT is not supported yet.

9.8 Changing the user after starting jo!

On Unix systems it is desirable to start a web server as root, so that it can bind to port 80 and then change to a user with less privileges. In pure Java this is not possible. However, jo! comes with a shared object for Linux x86 that can be used to achieve the desired behavior.

- Set both <servername>.user and <servername>.group to the desired values.
- Make sure your VM runs as a single process. For Sun JDK 1.3.1 you have to set the flag -classic right after the java command. Edit the start up script accordingly.
- Make sure all paths are accessible to the user you want to run jo! under. This also applies to directories that are created by jo! like JO_HOME/work and JO_HOME/temp.

To change the user under different Un*xes than Linux, edit the file JO_HOME/src/c/tagtraum.c, modify JO_HOME/build.xml to properly compile tagtraum.c, and run ant compile.

10 Configuring a listener

As already described in 2.2 jo! accepts connections with listeners. There can be multiple listeners for a single server. Each listener accepts connections on a single port and a single address or a single port and all valid addresses for the machine it runs on.

All listeners are configured in a single file called `listener.properties` located in `etc`. The file looks like this:

```
<listenername>.port=<portnumber>
<listenername>.bindaddress=<address to bind to>
<listenername>.backlog=<length of connection queue>
<listenername>.classname=<classname or class alias>
<listenername>.protocol=SSL|TLS|<blank>
<listenername>.keystore=<URL or file>
<listenername>.keystoreformat=JKS|PKCS12
<listenername>.passphrase=<the keystore's passphrase>
<listenername>.needClientAuth=true|false
```

10.1 Setting the port

In order to change the port of a listener, simply edit `<listenername>.port`. Note that under Unix systems you might not be able to use a port less than 1024 if you are not root.

10.2 Setting the bind address

Simply edit `<listenername>.bindaddress`. If you want to listen on all addresses please enter 0.0.0.0.

Note that Java™ looks up addresses over DNS before binding. If your computer is not connected to a DNS, the server will start very slowly (due to a timeout). You might want to change 0.0.0.0 into a real address. This should speed up the initial binding.

10.3 Backlog

When all handlerthreads are busy, incoming connections must be queued. The maximum length of this queue is set with the parameter `<listenername>.backlog`. When running benchmarks it is important that this parameter is very high (e.g. 1000). For normal use a value of 50 should be enough. Busy sites might have to set a higher value.

10.4 Listenerclass

In order to set the class used as listener you can use the parameter `<listenername>.classname`. As value you can set a classname or a class alias that can be resolved with the factory set for the service (see 9.5).

If you want to use a secure listener you have to set an appropriate classname here, i.e. `SSLListener`, that is resolved to the corresponding class as defined in `factory.properties`.

10.5 Using a secure protocol (e.g. SSL)

If you want to use a secure protocol like SSL (Secure Socket Layer) you can specify this with `<listenername>.protocol`. Valid values are SSL or TLS. Note that the standard port for SSL is 443 and that you have to set it manually!

In order to use a secure protocol you need a keystore with your server certificate. Specify this with the `<listenername>.keystore` parameter and set the used format with the `<listenername>.keystoreformat` parameter. Supported are JKS and PKCS12. As the keystore is protected by a passphrase you have to state this with `<listenername>.passphrase`.

If you want clients to prove their identity with a certificate, set `<listenername>.needClientAuth` to true.

Note that jo! takes some more time to start when run with a secure protocol. This is due to the fact that the class `SecureRandom` used in the implementation takes a very long time to initialize.

Please see the JSSE documentation (available from Sun as optional package and since JDK 1.4 as part of the JDK) and the Java-Security mailing list (http://archives.java.sun.com/cgi-bin/wa?S1=java-security) for further details.

10.6 How to get a signed certificate from a certification authority (CA)

In order to get a certificate from a CA such as http://www.verisign.com/ you have to generate a Cerificate Signing Request (CSR), send this to a CA, and install the returned signed certificate. You might also have to install a root certificate by your ca (step 4). Note that RSA is not supported by all JDK versions. We successfully tested it with Sun JDK 1.3 for Windows

1. Generate a self signed certificate using the keytool program provided with your JDK. Note that the common name (CN) must not contain spaces.
   ```
   keytool -genkey -alias <somename> -keyalg rsa
   ```

2. Generate the CSR:
   ```
   keytool -certreq -file <somename>.csr -alias <the above used aliasname>
   ```

3. Submit the CSR to the CA of your choice.

4. (If you are using a CA other than VeriSign or using a test certificate by VeriSign you need to import a root certificate for the CA.
   When using VeriSign just follow the instructions for installing the Trial CA Root in your browser, but shift-click the accept-button. Rename the now downloaded file to something ending with `.cer`. This is the Trial CA Root certificate. Then import it with the keytool:
   ```
   keytool -import -alias <alias for the CA> -file <the CA's certificate>
   ```
   Note that you have to install the certificate in your browser, too. To do that, simply follow the instructions by VeriSign.)

5. Import the signed certificate returned by the CA. Before doing that make sure that there is a linefeed character after the last line:
   ```
   keytool -import -trustcacerts -alias <the aliasname used in step 1> -file <the returned certificate>
   ```

6. Reference the used keystore (typically a file called `.keystore` in your home directory) with <listenername>.keystore and set <listenername>.keystoreformat to JKS.

Further information about using the keytool can be found in the JDK documentation.

11 Servlets

jo! comes with some servlets to ease your life a bit.

11.1 Status

With the current release of jo! comes a servlet called `com.tagtraum.jo.servlets.HealthWatcher`. It is basically a servlet that gives you some status information. Please note that the cachesize displayed by this servlet is not precise.

11.2 CGI

The CGIServlet is installed in `/cgi-bin/` and gives you the opportunity to execute CGI programs. Note that this is currently only possible for executables or batch files. On Windows machines perl scripts will not be executed correctly.

11.3 Invoker

Sometimes it is useful to invoke servlets without having to register them in the `web.xml` configuration file. This can be achieved with the InvokerServlet (`com.tagtraum.jo.servlets.InvokerServlet`). The servlet is usually mapped to `/invoker/*` and interprets the following path element as classname of a servlet it then executes.

E.g.
`http://127.0.0.1:8080/invoker/com.tagtraum.jo.servlets.SnoopServlet`
executes the servlet with the classname `com.tagtraum.jo.servlets.SnoopServlet`.

Note that the invoker introduces a vulnerability as every servlet in the classpath can be executed by any user!

12 Debugging with jo!

Debugging with jo! is quite simple. It probably works best with an IDE:

1. Make sure that all jars in `JO_HOME/lib` are in your classpath
2. Set `com.tagtraum.jo.gui.MainController` (with GUI) or `com.tagtraum.jo.JoIgnitionNG` (without GUI) as main class
3. Set the `jo1.0/`-directory as your working directory
4. Set `./etc/` as command line argument, if you can specify a working directory. If not, set the environment variable JO_HOME to the jo! installation directory.
5. Now start your VM in debug mode

Additionally it is very useful to provide a jo.xml file for your web application with auto-reload and send-errors-to client-enabled. See section 4.3 and following for details.

13 Building jo! from source

jo! is an open source project. I.e. all sources necessary to build jo! are included in the distribution.

To build jo! from source, you have to have a recent version of the build tool ant (http://jakarta.apache.org/ant/) installed.

To recompile jo!, simply change to `JO_HOME` and execute `ant compile`.

14 Integration

Of course jo! can be integrated with other applications. The following sections describe how this can be done.

14.1 Embedding jo! in other applications

A major advantage of using a pure Java™ web server is, that it is easy to embed in other applications. Possible resulting products are CD-ROM catalogues or in general software that should be easy to run on multiple platforms. To do this with jo! simply add some lines like the following to your program:

```
import java.net.*;

import com.tagtraum.jo.*;
import com.tagtraum.jo.builder.*;
import com.tagtraum.framework.util.*;
import com.tagtraum.framework.server.*;

...

URL theURL = new URL("http://myconfig.host.com/")
I_Builder theJoBuilder = (I_Builder) new
JoPropertyServiceBuilder();
theJoBuilder.setName("jo");
theJoBuilder.setURL(theURL);
I_JoServletService theService = null;
try {
   theService = (I_JoServletService)theJoBuilder.build()
   theService.start();
   ...
   // insert your business logic here - jo! is now available
   ...
   theService.stop();
   ...
}
catch (BuildException be) {
   System.err.println(be);
}
catch (ServerException se) {
   System.err.println(se);
}

...
```

The name jo has to reflect the `<servername>` in the file `server.properties`. The variable `theURL` has to point at a folder, not a file. The resulting `I_JoServletService`-Object can be started, stopped and restarted via the methods `start()`, `stop()`, and `restart()`. All of them may throw the exception `com.tagtraum.framework.server.-ServerException`.

14.2 JBoss

You can use jo! as servlet container for JBoss 2.2.2 or JBoss 2.4.x (http://www.jboss.org/). See `JO_HOME/integration/jboss2.x.x/README` for the latest details.

To build the necessary files, you will need to install ant (http://jakarta.apache.org/ant/), set the `jboss.home` property in the `build.xml` file in `JO_HOME/integration/jboss2.x.x/` and execute `ant dist`. You will then find a working distribution in the directory `dist`.

14.3 Avalon-Phoenix

You can use jo! as a servlet container/web-server-component in the Avalon-Phoenix framework (http://jakarta.apache.org/avalon/phoenix/). To do so you have to download the Phoenix release and have ant (http://jakarta.apache.org/ant/) installed. This version of jo! has been tested with Phoenix version 4.0. The latest info can be found in `JO_HOME/integration/phoenix4.0/README`.

To build jo! for phoenix edit the file `JO_HOME/integration/phoenix4.0/build.xml` (or `JO_HOME/integration/phoenix4.0/build-4.0a1.xml`, respectively) You have to set the property `phoenix.home` of the `build.xml` file to the path where you installed phoenix. To do so, look for a line like this:

```
<property name="phoenix.home" value="SET THE RIGHT PATH
HERE!!!"/>
```

Adjust the value attribute and run `ant install`.

If the build is successful, the jo!-SAR is installed and ready to run. Just execute one of the `run` scripts in the `bin` directory of your phoenix installation.

Note that the jo! service offers methods to deploy und un-deploy WARs:

```
public void deployWAR(String hostName, String name, String
ctxPath, String warUrl) throws JoException;

public void undeployWAR(String hostName, String name) throws
JoException;
```

In this context the variables are defined as follows:

- Hostname: Name of the host to deploy to. Null is the default host.
- CtxPath: Context path
- WarUrl: URL where the war can be found
- Name: Name of this WAR. If this is null, warUrl will be used as name

15 Starting and stopping jo! from a remote machine

When jo! is started with the `jo_ng` script, a little server is started on the machine the script is run on. This server can be configured with the files `/etc/metaserver.properties` and `/etc/metalistener.properties`.

Edit `metaserver.properties` so that the hostname of your client machine is listed in `MetaServer.validaddresses= ...` If it is not listed, it is not allowed to connect to the MetaServer.

Edit `metalistener.properties` so that `localhost.bindaddress` is either `0.0.0.0` or the real IP address of that machine, but not `127.0.0.1`.

You now can 'talk' with this server from a remote machine using the Script `metaclient.sh`, `metaclient.bat` or simple telnet. Just like telnet the metaclient scripts need the IP address or hostname of the server and the port as argument! E.g.:

```
metaclient.bat foo.bar.com 9090
```

Or:

```
telnet foo.bar.com 9090
```

Note: You are not authenticating yourself in any other way than your IP address. So this is not secure!

Once you are connected with the MetaServer you can start and stop jo! using the following commands:

```
start jo
```

```
stop jo
```

To stop the MetaServer, type:

```
stop MetaServer
```

To exit the client (not the server):

```
exit
```

Instead of using the metaserver you should consider using an MBean server and starting jo! as Mbean (see 3.3).

16 Running jo! behind a proxy

Sometimes it is desirable to run jo! behind a proxy. This can lead to problems with URL-rewriting especially in connection with the Jakarta Struts framework. Struts creates the rewritten URLs as absolute URLs pointing to the server the webapplication runs on. If jo! runs behind a proxy, those URLs should point to the proxy, but do point to jo! In order to convince jo! that he really should return the name and port of the proxy as server name and server port, you may install a request interceptor.

Simply edit the file `server.properties` so that it contains the following entries:

```
<servername>.requestinterceptor.hostrewrite=com.tagtraum.jo.
requestinterceptor.HostRewrite
<servername>.requestinterceptor.hostrewrite.parameters=<jo-
hostname>[:<jo-port>]=<proxy-hostname>[:<proxy-port>][, ...]
<servername>.requestinterceptor.hostrewrite.order=1
```

If you are using Apache as a proxy, you should configure it like this:

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_connect_module modules/mod_proxy_connect.so
LoadModule proxy_http_module modules/mod_proxy_http.so

<IfModule mod_proxy.c>
ProxyRequests On

<Proxy http://<proxy-hostname>/<proxy-path>/*>
 ProxyPass http://<jo-hostname>/<jo-path>/
 ProxyPassReverse http://<jo-hostname>/<jo-path>/
 Options Indexes FollowSymlinks MultiViews
       AllowOverride None
       Order allow,deny
       Allow from all
</Proxy>
```

For more information about the Apache proxy module, please look at the Apache documentation at http://www.apache.org/.

17 Installing jo! as service under NT

In order to install jo! as NT service you can use one of the freely available tools found under http://pharos.inria.fr/Java/search?term=s1:3428.

If you use one of these tools, it is likely that when the server is shut down the servlets' destroy methods are not called properly. Also session serialization and restore might not work. To be certain, please check the log file, whether jo shut down properly.

18 Setting the browser for viewing the documentation

On Unix/Linux systems jo! will try to launch Netscape for displaying the documentation. This requires a correct installation of Netscape and the Adobe Acrobat Reader plugin (http://www.adobe.com/products/acrobat/). If you prefer a different viewer than Netscape, you may edit the file `jo.properties` in your home directory. Simply set the entry `browser` to the value of your choice. E.g. `browser=konqueror` for KDE-users.

19 Loadbalancing

For scalability you might need to run jo! on multiple machines. You can do this successfully with hardware oriented devices which serve exactly that purpose. But usually these devices are very expensive. So you might want to try a cheap solution first. Such a solution is jobalancer. It is a little experimental load balancer that works as described below.

jobalancer accepts connections and basically forwards them to other registered servers. If one of the servers becomes unavailable (e.g. due to a crash, a network problem, maintenance, etc.), a different server will be taken. The server also is then marked as unavailable for a while. In order to allow sessions to work, requests from one IP address will always be forwarded to the same server (sticky). This also means that once a server becomes unavailable all sessions on that particular server are lost.

jobalancer can be configured with the file `/etc/balancer.xml`.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
This is the experimental configuration file for JoBalancer,
a tool for balancing the load for
several jo!s on different servers.
-->

<balancer>
<!--
These parameters are pretty similar to jo!s
-->
<service>
 <name>JoBalancer</name>
 <handler-
class>com.tagtraum.jobalancer.JoBalancerHandler</handler-
class>
 <version>
      <major>0</major>
      <minor>0</minor>
 </version>
 <log-config>
      <file>../log/balancer.log</file
      <level>2</level>
      <buffer-size>0</buffer-size>
 </log-config>
 <thread-config>
      <min>5</min>
      <max>100</max>
      <pool-reduction-timeout>5000</pool-reduction-timeout>
 </thread-config>
 <listener>
      <name>Localhost</name>
      <address>0.0.0.0</address>
      <port>8081</port>
      <backlog>50</backlog>
 </listener>
</service>
```

```
<!--
Servers at the back end. Requests are delegated to these.
-->
<server>
 <address>127.0.0.1</address>
 <port>8080</port>
</server>
<server>
 <address>127.0.0.1</address>
 <port>8079</port>
</server>

<!--
Requests from a particular IP address will always be
delegated to the same server, unless there hasn't been a
request for the specified timeout in min.
-->
<sticky-timeout>30</sticky-timeout>

<!--
If a connection to a server couldn't be established it is
marked unavailable for a certain time (in min). After this
time is over it is contacted again.
-->
<unavailable-timeout>1</unavailable-timeout>

</balancer>
```

20 Performance-Tuning

If you want to test the performance of jo!, please make sure that you consider the following.

- recompile jo! with optimization (see `compile` target in `JO_HOME/build.xml`)
- disable auto-reloading of servlets and JSP
- give as much memory to the VM as possible
- enable the cache
- set the cache's strong referenced parameter to a high value (depending on your memory and on your max cache entry size)
- use the *non*-gui version
- set all loglevels to 0 (don't do this in a production environment)
- set the log buffers to a value that is at least as high as you expect requests per second
- experiment with `server.maxrequests` and `server.keepalive` depending on what version of HTTP you use for your test
- experiment with listener backlog
- if you are using a Sun VM, use the HotSpot server version

We are always interested in performance numbers. If you have any, please send them along with an exact description of your test to <feedback@tagtraum.com>. Thanks!

21 Migrating from older versions

Because parameters and configuration files still change, this chapter tries to help you with migrating from one version to another.

21.1 jo! 1.0b7 to jo! 1.0

There shouldn't be any problems.

21.2 jo! 1.0b6 to jo! 1.0b7

The DTD for jo.xml has changed. This shouldn't lead to problems though. Note that the all xml documents are now verified. This means that xml documents with elements in the wrong order will lead to error messages in the log.

21.3 jo! 1.0b4 to jo! 1.0b5/6

There shouldn't be any problems.

21.4 jo! 1.0b3 to jo! 1.0b4

The DTD for jo.xml changed. The `<restore-sessions/>`-tag has been added. Therefore you must now refer to `http://www.tagtraum.com/dtds/jo-web-app_1.0b4.dtd`

This also means that session aren't restored automatically as default behavior anymore.

21.5 jo! 1.0b2 to jo! 1.0b3

jo! now supports auto deployment of wars in `JO_HOME/webapp/<hostname>/`

21.6 jo! 1.0b1 to jo! 1.0b2

The preferred way to deploy a war is now to drop it into `JO_HOME/webapp/<hostname>/`. The default war is no longer registered in `hosts.properties`.

jo! now correctly resolves dtds. So they should be present.

21.7 jo! 1.0a10 to jo! 1.0a11/b1

Note that an entry in `factory.properties` has changed. The proper entry is now:

```
AccessLogEvent=com.tagtraum.jo.log.CLFLogEvent
```

The DTD for `web.xml` has changed. The new DTD is now available from `http://www.tagtraum.com/dtds/ jo-web-app_1.0a11.dtd` . The element `<auto-reload/>` in `<jsp-config>` has been removed in favor of `<auto-reload-servlets/>` in `<web-app>`. If `<auto-reload-servlets/>` is present, *both* servlets and JSP are automatically reloaded when their source/class-files change. Note that this feature is for developing only, as it seriously affects the performance. In real life rather exchange the whole war to replace an application securely and without major performance effects.

21.8 jo! 1.0a9 to jo! 1.0a10

In previous versions jo! tried to automatically convert property files to xml, if no `web.xml` file was found. This feature has been disabled.

21.9 jo! 1.0a4x to jo! 1.0a5/6/7/8/9

The big change between these versions was to switch from properties to xml as configuration for the web application.

21.9.1 Switching from properties to xml

To assist you in migrating from 1.0a4x to 1.0a5 the current version comes with a little tool called `properties2xml`. It takes the base directory (*not* the `/WEB-INF/`-directory) of a web application as argument and tries to convert the property files to the deployment descriptors jo.xml and web.xml as well as possible. As jo! has changed somewhat some conversions will fail. To make sure that you application still works check the following (this might not be a complete list of problems):

- The parameter `automapping` defined in `servlets.properties` has been replaced by the tag `<auto-mapping>` in `jo.xml`. Note that you need to specify this yourself. Servlets are no longer mapped to `/servlets/<servletname>` and `/servlet/<servletname>` automatically (see 4.4).

- Options for the JspEngineServlet are not set. Please see 5 for details on how to configure JSP support.

21.9.2 Keeping properties

Support for properties will not be further developed!

If you rather work with property files for now, you can do so. Please note that not all features are supported – i.e. there is no way to configure the jsp engine.

- Edit the file `factory.properties` in `etc` so, that PeerBuilder equals `com.tagtraum.jo.builder.JoPropertyServletContextPeerBuilder` and *not* `com.tagtraum.jo.builder.JoXMLServletContextPeerBuilder`:

```
#PeerBuilder=com.tagtraum.jo.builder.JoPropertyServletContex
tPeerBuilder

PeerBuilder=com.tagtraum.jo.builder.JoXMLServletContextPeerB
uilder
```

- Change the parameter `gziprepository` in `webapp.properties` to `repository`

- Change the parameter `gzipextensions` in `webapp.properties` to `extensions`

- Remove the JspEngineServlet from `servlets.properties`

- Remove the mapping for the JspEngineServlet from `mappings.properties`

- Add the following line to `webapp.properties`:

```
algorithms=gzip
```

- Now cross your fingers and start the engine… ☺

22 Feedback and mailing list

If you have any question or comment about jo! – good or bad, please send mail to <feedback@tagtraum.com>. I am very eager to improve jo!, make it faster, even more stable and better fitting your needs. In case you find any bugs, please send mail to <bugs@tagtraum.com>.

Please join the announcement mailing list by sending a mail to <listserv@tagtraum.com> containing 'subscribe jo' in the body of the message.

Or, if you are interested in helping developing jo!, please join the developer's mailing list at http://lists.sourceforge.net/lists/listinfo/tagtraum-jo-development.

Thank you for using jo!

23 References

- Servlet API, http://java.sun.com/products/servlet/
- JSP, http://java.sun.com/products/jsp/
- J2EE, http://java.sun.com/j2ee/
- RFC 1630 Uniform Resource Identifiers (URI)
- RFC 1738 Uniform Resource Locators (URL)
- RFC 1808 Relative Uniform Resource Locators
- RFC 1945 Hypertext Transfer Protocol (HTTP/1.0)
- RFC 2045 MIME Part One: Format of Internet Message Bodies
- RFC 2046 MIME Part Two: Media Types
- RFC 2047 MIME Part Three: Message Header Extensions for non-ASCII text
- RFC 2048 MIME Part Four: Registration Procedures
- RFC 2049 MIME Part Five: Conformance Criteria and Examples
- RFC 2109 HTTP State Management Mechanism
- RFC 2145 Use and Interpretation of HTTP Version Numbers
- RFC 2616 Hypertext Transfer Protocol (HTTP/1.1)
- RFC 2617 HTTP Authentication: Basic and Digest Authentication

You can locate the online versions of any of these RFCs at: http://www.rfc-editor.org/