



The Definitive Reference

**Jan Bartel
Thomas Becker
Simone Bordet
Joakim Erdfelt
Jesse McConnell
Greg Wilkins**

Edited by Shirley Boulay

Revision History

Revision 9.2.1.v20140609 2014-06-10 08:36:04

This documentation is produced and contributed to under the [EPL](#).

Table of Contents

I. Getting Started With Jetty	1
1. Introducing Jetty	3
What is Jetty?	3
What Version Do I Use?	3
Jetty and Java EE Web Profile	4
2. Using Jetty Introduction	7
Downloading Jetty	7
Running Jetty	8
Deploying Web Applications	10
Finding Jetty in Maven	12
3. Jetty Configuration Introduction	13
How to Configure Jetty	13
What to Configure in Jetty	17
II. Jetty Configuration	24
4. Deploying to Jetty	26
Anatomy of a Web Application	26
Automatic Web Application Deployment	26
Configuring a Specific Web Application Deployment	27
Deployment Processing of WebAppContexts	29
Configuring Static Content Deployment	34
Hot Deployment	34
Deployment Architecture	35
Quickstart Webapps	39
Overlay WebApp Deployer	40
5. Configuring Contexts	47
Setting a Context Path	47
Configuring Virtual Hosts	48
Temporary Directories	51
Serving a WebApp from a Particular Port/Connector	54
Creating Custom Error Pages	55
Setting Max Form Size	58
6. Configuring Jetty Connectors	59
Connector Configuration Overview	59
Configuring SSL	65
Setting Port 80 Access for a Non-Root User	72
7. Configuring Security	76
Jetty 9.1: Using the \${jetty.home} and \${jetty.base} Concepts to Configure Security	76
Authentication	83
Limiting Form Content	89
Aliased Files and Symbolic links	90
Secure Password Obfuscation	91
JAAS Support	93
Spnego Support	99
8. Configuring JSP Support	103
Configuring JSP	103
III. Jetty Administration Guide	112
9. Starting Jetty	115
Startup Overview	115
Managing XML Based Startup Configuration	118
Managing Server Classpath	118
Managing Startup Modules	120
Managing Jetty Base and Jetty Home	124
Using start.jar	128
Startup a Unix Service using jetty.sh	131

Startup via Windows Service	135
10. Session Management	141
Setting Session Characteristics	141
Using Persistent Sessions	144
Session Clustering with a Database	146
Session Clustering with MongoDB	149
11. Configuring JNDI	154
Quick Setup	154
Detailed Setup	154
Working with Jetty JNDI	156
Configuring JNDI	159
Using JNDI with Jetty Embedded	163
Datasource Examples	165
12. Annotations	172
Quick Setup	172
Working with Annotations	173
Using Annotations with Jetty Embedded	175
13. JMX	179
Using Java Management Extensions (JMX)	179
Jetty JConsole	182
Jetty JMX Annotations	184
14. SPDY	187
Introducing SPDY	187
Configuring SPDY	187
Configuring SPDY Proxy	188
Configuring SPDY push	194
Implement a custom SPDY PushStrategy	197
15. ALPN	199
.....	199
16. NPN	203
.....	203
17. FastCGI Support	208
FastCGI Introduction	208
Configuring Jetty for FastCGI	208
18. Bundled Servlets, Filters, and Handlers	212
Default Servlet	212
Proxy Servlet	213
Balancer Servlet	214
CGI Servlet	215
Quality of Service Filter	215
Denial of Service Filter	218
Gzip Filter	220
Cross Origin Filter	222
Resource Handler	223
Debug Handler	225
Statistics Handler	226
IP Access Handler	227
Moved Context Handler	229
Shutdown Handler	230
Default Handler	231
Error Handler	231
19. Jetty Runner	233
Use Jetty without an installed distribution	233
20. Setuid	239
Configuring Setuid	239
21. Optimizing Jetty	241
Garbage Collection	241
High Load	242

Limiting Load	244
22. Jetty Logging	246
Configuring Jetty Logging	246
Default Logging with Jetty's StdErrLog	247
Configuring Jetty Request Logs	249
Example: Logging with Apache Log4j	251
Example: Logging with Java's java.util.logging via Slf4j	252
Example: Logging with Java's java.util.logging via JavaUtilLog	254
Example: Logging with Logback	255
Example: Capturing Multiple Logging Frameworks with Slf4j	256
Example: Centralized Logging with Logback	259
Jetty Dump Tool	262
IV. Jetty Development Guide	283
23. Maven and Jetty	285
Using Maven	285
Configuring the Jetty Maven Plugin	289
Files Scanned by the Jetty Maven Plugin	307
Jetty Jspc Maven Plugin	307
24. Using Ant with Jetty	312
Using the Ant Jetty Plugin	312
25. Handlers	322
Rewrite Handler	322
Writing Custom Handlers	324
26. Embedding	328
Jetty Embedded HelloWorld	328
Embedding Jetty	329
Embedded Examples	334
27. Debugging	339
Options	339
Enable remote debugging	339
Debugging With Eclipse	340
Debugging With IntelliJ	342
28. Frameworks	347
Spring Setup	347
OSGI	348
Weld	362
Metro	364
29. HTTP Client	366
Introduction	366
API Usage	367
Other Features	373
30. WebSocket Introduction	376
What Jetty provides	376
WebSocket APIs	377
31. Jetty Websocket API	378
Jetty WebSocket API Usage	378
WebSocket Events	378
WebSocket Session	379
Send Messages to Remote Endpoint	379
Using WebSocket Annotations	384
Using WebSocketListener	385
Using the WebSocketAdapter	386
Jetty WebSocket Server API	387
Jetty WebSocket Client API	389
32. Java Websocket API	391
Java WebSocket Client API Usage	391
Java WebSocket Server API	391
V. Reference Guide	392

33. Platforms, Stacks and Alternative Distributions	394
Many many options.....	394
Jelastic	394
CloudFoundry	394
Amazon Elastic Beanstalk	396
Fedora	398
Ubuntu	398
34. Architecture	399
Jetty Architecture	399
Jetty Classloading	405
Managing 1xx Responses	408
Creating a Custom Protocol	408
35. Contributing to Jetty	411
Community	411
Documentation	412
Source Control and Building	415
Coding Standards	417
Issues, Features, and Bugs	418
Contributing Patches	418
Releasing Jetty	422
Testing a Jetty Release	425
36. Reference Section	430
Jetty XML Syntax	430
Jetty XML Usage	442
jetty.xml	443
jetty-web.xml	444
jetty-env.xml	445
webdefault.xml	446
Jetty override-web.xml	448
37. Troubleshooting	450
Troubleshooting Zip Exceptions	450
Troubleshooting Locked Files on Windows	450
Preventing Memory Leaks	452
Jetty Security Reports	455

List of Tables

1.1. Jetty Versions	3
1.2. JavaEE7 Web Profile	4
1.3. Java EE 6 Web Profile	5
2.1. Contents	7
4.1. Default Configuration classes	29
4.2. JNDI Configuration classes	31
4.3. Annotation Configuration classes	31
6.1. Connector Configuration	60
8.1. Understanding Apache JspServlet Parameters	104
8.2. Understanding Glassfish JSP Parameters	107
10.1. Init Parameters	141
10.2. Default Values for Session Id Table	148
10.3. Default Values for Session Table	148
11.1. DataSource Declaration Conventions	160
15.1. ALPN vs. OpenJDK versions	202
16.1. NPN vs. OpenJDK versions	206
22.1. Slf4j Logging Grid	257
28.1. Bundle Name Mapping	349
28.2. Jars Required for JSP	360
28.3. Jars Required for Annotations	362
30.1. WebSocket connection states	376
34.1. Default System Classes	406
34.2. Default Server Classes	406
37.1. Resolved Issues	455

List of Examples

Client Example.	200
Server Example.	200
Client Example.	204
Server Example.	204
26.1. FileServer.java	334
26.2. SplitFileServer.java	334
26.3. ManyConnectors.java	335
26.4. SpdyServer.java	336
26.5. SecuredHelloHandler.java	337
26.6. MinimalServlets.java	337
26.7. OneWebApp.java	338
31.1. Send Binary Message (Blocking)	380
31.2. Send Text Message (Blocking)	380
31.3. Send Partial Binary Message (Blocking)	380
31.4. Send Partial Text Message (Blocking)	381
31.5. Send Ping Control Frame (Blocking)	381
31.6. Send Pong Control Frame (Blocking)	381
31.7. Send Binary Message (Async Simple)	382
31.8. Send Binary Message (Async, Wait Till Success)	382
31.9. Send Binary Message (Async, timeout of send)	382
31.10. Send Text Message (Async Simple)	383
31.11. Send Text Message (Async, Wait Till Success)	383
31.12. Send Text Message (Async, timeout of send)	383
31.13. AnnotatedEchoSocket.java	384
31.14. ListenerEchoSocket.java	386
31.15. AdapterEchoSocket.java	386
31.16. MyEchoServlet.java	387
31.17. MyAdvancedEchoCreator.java	388
31.18. MyAdvancedEchoServlet.java	388
31.19. SimpleEchoClient.java	389
31.20. SimpleEchoSocket.java	390

Part I. Getting Started With Jetty

Table of Contents

1. Introducing Jetty	3
What is Jetty?	3
What Version Do I Use?	3
Jetty and Java EE Web Profile	4
2. Using Jetty Introduction	7
Downloading Jetty	7
Running Jetty	8
Deploying Web Applications	10
Finding Jetty in Maven	12
3. Jetty Configuration Introduction	13
How to Configure Jetty	13
What to Configure in Jetty	17

Chapter 1. Introducing Jetty

Table of Contents

What is Jetty?	3
What Version Do I Use?	3
Jetty and Java EE Web Profile	4

What is Jetty?

Jetty is an open-source project providing an HTTP server, HTTP client, and javax.servlet container.

This guide is in two parts.

- The first part emphasizes beginning to use Jetty. It provides information about downloading Jetty, changing things like the port Jetty runs on, adjusting logging levels, and configuring many of the most common servlet container features such as JNDI, JMX, and session management.
- The second part describes advanced uses of Jetty, providing in depth coverage of specific features like our highly scalable async client, proxy servlet configuration, the Jetty Maven plugin, and using Jetty as an embedded server. The advanced section includes tutorials, howtos, videos, and reference materials.

What Version Do I Use?

Jetty 9 is the most recent version of Jetty and has a great many improvements over previous versions. One improvement is this documentation which focuses on Jetty 9. While many people continue to use older versions of Jetty, we generally recommend using Jetty 9 as it represents the version of Jetty that we will actively maintain and improve over the next few years.

Table 1.1. Jetty Versions

Ver-sion	Year	Home	JVM	Protocols	Servlet	JSP	Status
9.1	2013	Eclipse	1.7	HTTP/1.1 RFC2616, javax.websocket, SPDY v3	3.1	2.3	Stable
9	2012	Eclipse	1.7	HTTP/1.1 RFC2616, Web- Socket RFC 6455, SPDY v3	3.0	2.2	Stable
8	2009-	Eclipse/ Codehaus	1.6	HTTP/1.1 RFC2616, Web- Socket RFC 6455, SPDY v3	3.0	2.2	Mature
7	2008-	Eclipse/ Codehaus	1.5	HTTP/1.1 RFC2616, Web- Socket RFC 6455, SPDY v3	2.5	2.1	Mature
6	2006-2010	Codehaus	1.4-1.5	HTTP/1.1 RFC2616	2.5	2.0	Venerable
5	2003-2009	Sourceforge	1.2-1.5	HTTP/1.1 RFC2616	2.4	2.0	Deprecated

Ver-sion	Year	Home	JVM	Protocols	Servlet	JSP	Status
4	2001-2000	Sourceforge	1.2, J2ME	HTTP/1.1 RFC2616	2.3	1.2	Ancient
3	1999-2000	Sourceforge	1.2	HTTP/1.1 RFC2068	2.2	1.1	Fossilized
2	1998-2000	Mortbay	1.1	HTTP/1.0 RFC1945	2.1	1.0	Legendary
1	1995-1998	Mortbay	1.0	HTTP/1.0 RFC1945	-	-	Mythical

Jetty and Java EE Web Profile

Jetty implements aspects of the Java EE specification, primarily the Servlet Specification. Recent releases of the Java EE platform have introduced a Web Profile, recognizing that many developers need only a subset of the many technologies under the Java EE umbrella.

While Jetty itself does not ship all of the Web Profile technologies, Jetty architecture is such that you can plug in third party implementations to produce a container customized to your exact needs.

Java EE 7 Web Profile

In the forthcoming Java EE-7 specification, the Web Profile reflects updates in its component specifications and adds some new ones:

Table 1.2. JavaEE7 Web Profile

JSR	Name	Included with jetty-9.1.x	Pluggable
JSR 340	Servlet Specification API 3.1	Yes	
JSR 344	Java Server Faces 2.2 (JSF)	No	Yes, Mojarra or MyFaces
JSR 245 / JSR 341	Java Server Pages 2.3/ Java Expression Language 3.0 (JSP/EL)	Yes	Yes
JSR 52	Java Standard Tag Library 1.2 (JSTL)	Yes	Yes
JSR 45	Debugging Support for Other Languages 1.0	Yes (via JSP)	Yes (via JSP)
JSR 346	Contexts and Dependency Injection for the JavaEE Platform 1.1 (Web Beans)	No	Yes, Weld
JSR 330	Dependency Injection for Java 1.0	No	Yes as part of a CDI implementation, Weld
JSR 316	Managed Beans 1.0	No	Yes, as part of another technology
JSR 345	Enterprise JavaBeans 3.2 Lite	No	
JSR 338	Java Persistance 2.1 (JPA)	No	Yes, eg Hibernate

JSR	Name	Included with jetty-9.1.x	Pluggable
JSR 250	Common Annotations for the Java Platform 1.2	Yes	Partially (for non-core Servlet Spec annotations)
JSR 907	Java Transaction API 1.2 (JTA)	Yes	Yes
JSR 349	Bean Validation 1.1	No	Yes as part of another technology eg JSF, or a stand-alone implementation such as Hibernate Validator
JSR 339	Java API for RESTful Web Services 2.0 (JAX-RS)	No	
JSR 356	Java API for Websocket 1.0	Yes	No
JSR 353	Java API for JSON Processing 1.0 (JSON-P)	No	Yes, eg JSON-P reference implementation
JSR 318	Interceptors 1.2	No	Yes as part of a CDI implementation

Jetty EE 6 Web Profile

Here is the matrix of JSRs for Java EE 6 Web Profile, and how they relate to Jetty:

Table 1.3. Java EE 6 Web Profile

JSR	Name	Included with jetty-9.0.x	Pluggable
JSR 315	Servlet Specification API 3.0	Yes	
JSR 314	JavaServer Faces 2.0 (JSF)	No	Yes, for example, Mojarra or MyFaces
JSR 245	JavaServer Pages 2.2/ Java Expression Language 2.2 (JSP/EL)	Yes	Yes
JSR 52	Java Standard Tag Library 1.2 (JSTL)	Yes	Yes
JSR 45	Debugging Support for Other Languages 1.0	Yes (via JSP)	Yes (via JSP)
JSR 299	Contexts and Dependency Injection for the Java EE Platform 1.0 (Web Beans)	No	Yes, Weld or OpenWebBeans
JSR 330	Dependency Injection for Java 1.0	No	Yes as part of a CDI implementation, Weld
JSR 316	Managed Beans 1.0	No	Yes, as part of another technology.
JSR 318	Enterprise JavaBeans 3.1	No	Yes, OpenEJB

JSR	Name	Included with jetty-9.0.x	Pluggable
JSR 317	Java Persistence 2.0 (JPA)	No	Yes, Hibernate
JSR 250	Common Annotations for the Java Platform	Yes	Partially (for non-core Servlet Spec annotations)
JSR 907	Java Transaction API (JTA)	Yes	Implementations are pluggable, such as Atomikos , JOTM , Jencks (Geronimo Transaction Manager)
JSR 303	Bean Validation 1.0	No	Yes as part of another technology (JSF), or a stand-alone implementation such as Hibernate Validator

Chapter 2. Using Jetty Introduction

Table of Contents

Downloading Jetty	7
Running Jetty	8
Deploying Web Applications	10
Finding Jetty in Maven	12

You can use Jetty in many different ways, ranging from embedding Jetty in applications, launching it from different build systems, from different JVM-based languages, or as a standalone distribution. This guide covers the latter, a standalone distribution suitable for deploying web applications.

Downloading Jetty

Downloading the Jetty Distribution

The standalone Jetty distribution is available for download from the Eclipse Foundation:

<http://download.eclipse.org/jetty>

It is available in both zip and gzip formats; download the one most appropriate for your system. Notice that there are a number of other files with extensions of .sha or .md5 which are checksum files. When you download and unpack the binary, it is extracted into a directory called `jetty-distribution-VERSION`. Put this directory in a convenient location. The rest of the instructions in this documentation refer to this location as either `JETTY_HOME` or as `$(jetty.home)`.

Distribution Content

A quick rundown of the distribution's contents follows. The top-level directory contains:

Table 2.1. Contents

Location	Description
license-eplv10-aslv20.html	License file for Jetty
README.txt	Useful getting started information
VERSION.txt	Release information
bin/	Utility shell scripts to help run Jetty on Unix systems
demo-base/	A Jetty base directory to run a Jetty server with demonstration webapps
etc/	Directory for Jetty XML configuration files
lib/	All the JAR files necessary to run Jetty
logs/	Directory for request logs
modules/	Directory of module definitions
notice.html	License information and exceptions
resources/	Directory containing additional resources for classpath, activated via configuration
start.d/	Directory of *.ini files containing arguments that are added to the effective command line (see start.ini)

Location	Description
start.ini	File containing the arguments that are added to the effective command line (modules, properties and XML configuration files)
start.jar	Jar that invokes Jetty (see also the section called “Running Jetty”)
webapps/	Directory containing webapps that run under the default configuration of Jetty

Running Jetty

To start Jetty on the default port of 8080, run the following command:

```
> cd $JETTY_HOME
> java -jar start.jar
2013-09-06 13:52:43.326:INFO:oejs.Server:main: jetty-9.1.0
2013-09-06 13:52:43.358:INFO:oejdp.ScanningAppProvider:main: Deployment monitor [file:/home/gregw/src/jetty-9.1/jetty-distribution/target/distribution/webapps/] at interval 1
2013-09-06 13:52:43.390:INFO:oejs.ServerConnector:main: Started
ServerConnector@2edf362d{HTTP/1.1}{0.0.0.0:8080}
```

You can point a browser at this server at <http://localhost:8080>.

Demo Base

Since release 9.1.0, the jetty distribution does not deploy any demo web applications, so to see a more interesting demonstration of the server you need to run from the `demo-base` directory as follows:

```
> cd $JETTY_HOME/demo-base/
> java -jar ./start.jar
2013-09-06 13:56:35.939:WARN::main: demo test-realm is deployed. DO NOT USE IN
PRODUCTION!
2013-09-06 13:56:35.943:INFO:oejs.Server:main: jetty-9.1.0-SNAPSHOT
2013-09-06 13:56:35.976:INFO:oejdp.ScanningAppProvider:main: Deployment monitor [file:/home/gregw/src/jetty-9.1/jetty-distribution/target/distribution/demo-base/webapps/] at
interval 1
2013-09-06 13:56:36.240:INFO:oejpw.PlusConfiguration:main: No Transaction manager found -
if your webapp requires one, please configure one.
2013-09-06 13:56:36.732:WARN::main: async-rest webapp is deployed. DO NOT USE IN
PRODUCTION!
[...]
2013-09-06 13:56:38.572:INFO:oejsh.ContextHandler:main: Started
o.e.j.w.WebAppContext@2757052e{/,file:/home/gregw/src/jetty-9.1/jetty-distribution/
target/distribution/demo-base/webapps/ROOT/,AVAILABLE}{/ROOT}
2013-09-06 13:56:38.579:INFO:oejs.ServerConnector:main: Started
ServerConnector@19e0cb78{HTTP/1.1}{0.0.0.0:8080}
```

Pointing a browser at <http://localhost:8080>, will now show a welcome page and several demo/test web applications.



Warning

The demonstration web applications are not necessarily secure and should not be deployed in production web servers.

Creating a new Jetty Base

The `demo-base` directory described above is an example of the `jetty.base` mechanism added in Jetty 9.1. A jetty base allows the configuration and web applications of a server instance to be stored separately from the jetty distribution, so that upgrades can be done with minimal disruption. Jetty's default configuration is based on two properties:

jetty.home

The property that defines the location of the jetty distribution, its libs, default modules and default XML files (typically start.jar, lib, etc)

jetty.base

The property that defines the location of a specific instance of a jetty server, its configuration, logs and web applications (typically start.ini, start.d, logs and webapps)

The `jetty.home` and `jetty.base` properties may be explicitly set on the command line, or they can be inferred from the environment if used with commands like:

```
> cd $JETTY_BASE  
> java -jar $JETTY_HOME/start.jar
```

The following commands create a new base directory and enable a HTTP connector and web application deployer in it:

```
> mkdir /tmp/mybase  
> cd /tmp/mybase  
> java -jar $JETTY_HOME/start.jar  
WARNING: Nothing to start, exiting ...  
  
Usage: java -jar start.jar [options] [properties] [configs]  
       java -jar start.jar --help # for more information  
  
>java -jar $JETTY_HOME/start.jar --add-to-startd=http,deploy  
http      initialised in ${jetty.base}/start.d/http.ini (created)  
server   initialised in ${jetty.base}/start.d/server.ini (created)  
deploy   initialised in ${jetty.base}/start.d/deploy.ini (created)  
MKDIR: ${jetty.base}/webapps  
server   initialised in ${jetty.base}/start.d/server.ini  
  
> java -jar $JETTY_HOME/start.jar  
2013-09-06 14:59:32.542:INFO:oejs.Server:main: jetty-9.1.0-SNAPSHOT  
2013-09-06 14:59:32.572:INFO:oejdp.ScanningAppProvider:main: Deployment monitor [file:/  
tmp/mybase/webapps/] at interval 1  
2013-09-06 14:59:32.602:INFO:oejs.ServerConnector:main: Started  
ServerConnector@405a2273{HTTP/1.1}{0.0.0.0:8080}  
  
[...]
```

Changing the Jetty Port

You can configure Jetty to run on a different port by setting the `jetty.port` Property on the command line:

```
> cd $JETTY_HOME/demo-base  
> java -jar start.jar jetty.port=8081
```

Alternatively, property values can be added to the effective command line built from the `start.ini` file and `start.d/*.ini` files. By default, the jetty distribution defines the `jetty.port` property in the `start.d/http.ini` file, which may be edited to set another value.



Note

The configuration by properties works via the following chain:

- The `start.d/http.ini` file is part of the effective command line and contains the `--module=http` argument which activates the http module
- The `modules/http.mod` file defines the http module which specifies the `etc/jetty-http.xml` configuration file and the template ini properties it uses.
- The `jetty.port` property is used by the `Property` XML element in `etc/jetty.http.xml` to inject the `ServerConnector` instance with the port

For more information see the [Quickstart Configuration Guide](#) and [Configuring Connectors](#).

Starting HTTPS

To add the HTTPS connector to a jetty configuration, the https module can be activated by the following command:

```
> java -jar start.jar --add-to-startd=https
https          initialised in ${jetty.home}/start.d/https.ini (created)
ssl           initialised in ${jetty.home}/start.d/ssl.ini (created)
server        enabled in      ${jetty.home}/start.ini
resources     enabled in      ${jetty.home}/start.ini
ext           enabled in      ${jetty.home}/start.ini

> java -jar start.jar
2013-09-06 13:52:43.326:INFO:oejs.Server:main: jetty-9.1.0
[...]
```

The --add-to-startd command sets up the effective command line in the ini files to run a https connection as follows:

- creates the https.ini file that activates and configures the https connector module. The https module adds the etc/jetty-https.xml file to the effective command line.
- creates the start.d/ssl.ini file that activates and configures the SSL keystore using demonstration passwords. The ssl module adds the etc/jetty-ssl.xml file to the effective command line.
- checks for the existence of a etc/keystore file and if not present, downloads a demonstration keystore file.



Note

If a single start.ini file is preferred over individual start.d/*.ini files, then the option --add-to-start=module may be used to append the module activation to the start.ini file rather than create a file in start.d

More start.jar options

The job of the start.jar is to interpret the command line, start.ini and start.d to build a Java classpath and list of properties and configuration files to pass to the main class of the Jetty XML configuration mechanism. The start.jar mechanism has many options which are documented in the [startup](#) administration section and you can see them in summary by using the command:

```
> java -jar start.jar --help
```

Deploying Web Applications

Jetty server instances that configures the deploy module will have a web application deployer that hot deploys files found in the webapps directory. Standard WAR files and jetty configuration files that you place in the webapps directory are hot deployed to the server with the following conventions:

- A directory called example/ is deployed as a standard web application if it contains a WEB-INF/ subdirectory, otherwise it is deployed as context of static content. The context path is /example (that is, http://localhost:8080/example/) unless the base name is ROOT (case insensitive), in which case the context path is /. If the directory name ends with ".d" it is ignored (but may be used by explicit configuration).

- A file called `example.war` is deployed as a standard web application with the context path / `example` (that is, `http://localhost:8080/example/`). If the base name is `ROOT` (case insensitive), the context path is `/`. If `example.war` and `example/` exist, only the WAR is deployed (which may use the directory as an unpack location).
- An XML file like `example.xml` is deployed as a context whose configuration is defined by the XML. The configuration itself must set the context path. If `example.xml` and `example.war` exist, only the XML is deployed (which may use the WAR in its configuration).

If you have a standard web application, you can hot deploy it into Jetty by copying it into the `webapps` directory.

Jetty Demonstration Web Applications

The demo-base/webapps directory contains the following deployable and auxiliary files:

ROOT/

A directory of static content that is deployed to the root context / due to it's name. Contains the Jetty demo welcome page.

test.xml

A context configuration file that configures and deploys `test.war`. The additional configuration includes the context path as well as setting additional descriptors found in the `test.d` directory.

test.war

The demonstration web application that is configured and deployed by `test.xml`.

test.d

A directory containing additional configuration files used by `test.xml` to inject extra configuration into `test.war`.

async-rest.war

A web application demonstration of asynchronous REST to eBay, automatically deployed to / `async-rest` based on the file name.

test-jaas.war

A demonstration web application utilising **JAAS** for authentication.

test-jaas.xml

A context configuration file that configures `test-jaas.war`. Additional configuration includes setting up the **LoginService** for authentication and authorization.

test-jndi.war

A demonstration web application showing the use of **JNDI**.

test-jndi.xml

A context configuration file that configures `test-jndi.war`. Additional configuration includes defining objects in the naming space that can be referenced from the webapp.

test-spec.war

A demonstration web application that shows the use of annotations, fragments, ServletContainerInitializers and other Servlet Specification 3.0/3.1 features.

test-spec.xml

A context configuration file that configures `test-spec.war`. Additional configuration includes setting up some objects in the naming space that can be referenced by annotations.

xref-proxy.war

A demonstration web application that uses a transparent proxy to serve the Jetty src [xref](#) from the Eclipse [website](#).

example-moved.xml

A demonstration context configuration file that shows how to use the [MovedContextHandler](#) to redirect from one path to another.

Finding Jetty in Maven

Maven Coordinates

Jetty has existed in Maven Central almost since its inception, however under a couple of different coordinates over the years. When Jetty was based at SourceForge and then The Codehaus it was located under the groupId of `org.mortbay.jetty`. With Jetty 7 the project moved to the Eclipse foundation and to a new groupId at that time to reflect its new home.

The top level POM for the Jetty project is located under the following coordinates.

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-project</artifactId>
  <version>${project.version}</version>
</dependency>
```

Changelogs in Central

The changes between versions of Jetty are tracked in a file called `VERSIONS.txt`, which is under source control and is generated on release. Those generated files are also uploaded into Maven Central during the release of the top level POM. You can find them as a classifier marked artifact.

<http://repo2.maven.org/maven2/org/eclipse/jetty/jetty-project/9.0.0.RC0/jetty-project-9.0.0.RC0-version.txt>

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-project</artifactId>
  <version>${project.version}</version>
  <classifier>version</classifier>
  <type>txt</type>
</dependency>
```

Chapter 3. Jetty Configuration

Introduction

Table of Contents

How to Configure Jetty	13
What to Configure in Jetty	17

How to Configure Jetty

To understand Jetty configuration, you need to understand the "How" and the "What". This section covers how to configure Jetty in terms of what mechanisms exist to perform configuration. The [next section](#) gives an overview of the action components and fields that you can configure with these mechanisms.

Jetty POJO Configuration

The core components of Jetty are simply Plain Old Java Objects ([POJOs](#)); the process of configuring Jetty is mostly the process of instantiating, assembling and setting fields on the Jetty POJOs. You can achieve this either by:

- Writing Java code to directly instantiate and assemble Jetty objects. This is referred to as the section called “Embedding Jetty”.
- Using Jetty XML configuration, which is an [Inversion of Control \(IoC\)](#) framework, to instantiate and assemble Jetty objects as XML objects. The `etc/jetty.xml` file is the main Jetty XML configuration file, but there are many other `etc/jetty-feature.xml` files included in the jetty distribution.
- Using a third party [IoC](#) framework like [Spring](#), to instantiate and assemble Jetty objects as Spring beans.

Because the main Jetty configuration is done by IoC, the [Jetty API documentation](#) is the ultimate configuration reference.

Jetty Start Configuration Files

The Jetty distribution uses the following configuration files to instantiate, inject and start server via the start.jar mechanism

ini files

The Jetty Start mechanism uses the command line, the `start.ini` file and any `start.d/*.ini` files to create an effective command line of arguments. Arguments may be:

- XML files in Jetty IoC (or spring) XML format
- Module activations in the form `--module=name`
- Property in the form of `name=value`, used to parameterize Jetty IoC XML
- A standard [Java property file](#) containing additional start properties
- Other start.jar options (see `java -jar start.jar --help`)
- Some JVM options

Ini files are located in the jetty.base (if different from jetty.home) and are typically edited to change the configuration.

mod files

The modules/*.mod files contain the definition of modules that can be activated by --module=name. Each mod file defines:

- Module dependencies for ordering and activation
- The libraries needed by the module to be added to the classpath
- The XML files needed by the module to be added to the effective command line
- Files needed by the activated module
- A template ini file to be used when activating with the --add-to-start=name option

Mod files are normally located in jetty.home, but may be overridden in jetty.base if need be. Typically module files are rarely edited and only for significant structural changes.

XML files

XML files in Jetty or spring IoC format are listed either on the command line, ini files or added to the effective command line by a module definition. The XML files instantiate and inject the actual Java objects that comprise the server, connectors and contexts. Because Jetty IoC XML files frequently use properties, many common configuration tasks can be accomplished without editing these XML files. If XML configuration changes are required, the XML file should be copied from jetty.home/etc to jetty.base/etc before being modified.

Other Configuration Files

In addition to the start configuration files described above, the configuration of the server can use the following file:

Context XML files

Any XML files in Jetty or spring IoC format that is discovered in the webapps directory are used by the deploy module to instantiate and inject HttpContext instances to create a specific context. These may be standard web applications or bespoke contexts created from special purpose handlers.

web.xml

The [Servlet](#) Specification defines the [web.xml](#) deployment descriptor that defines and configures the filters, servlets and resources a [web application](#) uses. The Jetty WebAppContext component uses this XML format to:

- Set up the default configuration of a web application context.
- Interpret the application-specific configuration supplied with a web application in the WEB-INF/web.xml file.
- Interpret descriptor fragments included in the META-INF directory of Jar files within WEB-INF/lib.

Normally the web.xml file for a web application is found in the WEB-INF/web.xml location within the war file/directory or as web.xml fragments with jar files found in WEB-INF/lib. Jetty also supports multiple web.xml files so that a default descriptor may be applied before WEB-INF/web.xml (typically set to etc/webdefault.xml by the deploy module) and an override descriptor may be applied after WEB-INF/web.xml (typically set by a context XML file see test.xml)

Property Files

Standard [Java property files](#) are also used for Jetty configuration in several ways:

- To parameterize Jetty IoC XML via the use of the Property element.

- To configure the default logging mechanism (StdErrLog). You can also plug in other logging frameworks, and also use property files (for example, log4j).
- As a simple database for login usernames and credentials.

Jetty IoC XML Configuration format

To understand the Jetty IoC XML format, consider the following example of an embedded Jetty server instantiated and configured in java:

```
package org.eclipse.jetty.embedded;

import org.eclipse.jetty.server.Connector;
import org.eclipse.jetty.server.Handler;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.server.ServerConnector;
import org.eclipse.jetty.server.handler.DefaultHandler;
import org.eclipse.jetty.server.handler.HandlerCollection;
import org.eclipse.jetty.servlet.ServletContextHandler;
import org.eclipse.jetty.util.thread.QueuedThreadPool;

public class ExampleServer {

    public static void main(String[] args) throws Exception {
        Server server = new Server();
        ServerConnector connector = new ServerConnector(server);
        connector.setPort(8080);
        server.setConnectors(new Connector[] { connector });
        ServletContextHandler context = new ServletContextHandler();
        context.setContextPath("/hello");
        context.addServlet(HelloServlet.class, "/");
        HandlerCollection handlers = new HandlerCollection();
        handlers.setHandlers(new Handler[] { context, new DefaultHandler() });
        server.setHandler(handlers);
        server.start();
        server.join();
    }
}
```

Jetty IoC XML allows you to instantiate and configure the exact same server in XML without writing any java code:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure id="ExampleServer" class="org.eclipse.jetty.server.Server">

    <Set name="connectors">
        <Array type="org.eclipse.jetty.server.Connector">
            <Item>
                <New class="org.eclipse.jetty.server.ServerConnector">
                    <Arg><Ref refid="ExampleServer"/></Arg>
                    <Set name="port">8080</Set>
                </New>
            </Item>
        </Array>
    </Set>

    <New id="context" class="org.eclipse.jetty.servlet.ServletContextHandler">
        <Set name="contextPath">/hello</Set>
        <Call name="addServlet">
            <Arg>org.eclipse.jetty.embedded.HelloServlet</Arg>
            <Arg>/</Arg>
        </Call>
    </New>

    <Set name="handler">
        <New class="org.eclipse.jetty.server.handler.HandlerCollection">
            <Set name="handlers">
                <Array type="org.eclipse.jetty.server.Handler">
                    <Item>
                        <Ref refid="context" />
                    </Item>
                </Array>
            </Set>
        </New>
    </Set>

```

```
</Item>
<Item>
<New class="org.eclipse.jetty.server.handler.DefaultHandler" />
</Item>
</Array>
</Set>
</New>
</Set>
</Configure>
```

In practise, most commonly used Jetty features have XML files that are included in the standard distribution in the `etc` directory. Thus configuring Jetty is often a matter of just editing existing XML files and altering the configuration values within them.

Configuring the Jetty Distribution

With a normal distribution of Jetty, the configuration mechanisms introduced above are typically used as follows:

`$JETTY_HOME/start.ini`

Used for global command line options. In the standard Jetty distribution since release 9.1 the contents of `start.ini` have been moved to modular ini files under the `start.d/`.

`$JETTY_HOME/start.d`

A directory of modular ini files that are used to set the OPTION, parameters and configuration files for key Jetty modules. These modules may be enabled/disabled by renaming the files or since release 9.1 by using the `--enable` or `--disable` option of `start.jar`

`$JETTY_HOME/lib/*.xml`

Jetty IoC XML files that configure individual features; for example `jetty.xml` (for the server), `jetty-http.xml`, `jetty-https.xml`, `jetty-jmx.xml`.

`$JETTY_HOME/webapps/*`

The directory in which standard WAR files, web applications and context IoC XML files are deployed.

Jetty Port Configuration Example

The following example examines the common Jetty configuration mechanism using the HTTP port as an example. To run the Jetty server from the distribution, it is just a matter of running the following command:

```
> java -jar start.jar
```

The Jetty distribution uses the `start.d/` directory to define what modules, configuration files and properties are defined. The `start.d/http.ini` module is enabled by default and contains the following lines (among others):

```
--module=http
jetty.port=8080
```

This sets the Jetty port parameter and activates the http module as defined by the `modules/http.mod` file, which lists the `etc/jetty-http.xml` configuration file as an effective command line argument. Looking inside that configuration file, we see that it calls `addConnector` on the server with a new instance of the `ServerConnector` class. This instance is configured in XML and includes the line:

```
<Set name="port"><Property name="jetty.port" default="80" /></Set>
```

The effect of this line is to call `ServerConnector.setPort(int)` with either the value of the `jetty.port` property if set or with the value 80. We can see from the `start.ini` except above that the `jetty.port` property line is set to 8080, so out of the box, the Jetty distribution will use the default 8080 port set inside the `start.d/http.ini` file.

If you wish to change the port, then the start.d/http.ini file can be edited and the default value there changed to the new port. However this does mean that a file which is part of the Jetty distribution has been modified, which can make future upgrades more difficult. Thus it is normally better to create a jetty.base directory and configure a new instance of a server there.

What to Configure in Jetty

This section gives an overview of the components of Jetty you typically configure using the mechanisms outlined in the previous section. The section called “Jetty Architecture” describes the structure of a Jetty server, which is good background reading to understand configuration, and is vital if you want to change the structure of the server as set up by the default configurations in the Jetty distribution. However, for most purposes, configuration is a matter of identifying the correct configuration file and modifying existing configuration values.

Configuring the Server

The Server instance is the central coordination object of a Jetty server; it provides services and life cycle management for all other Jetty server components. In the standard Jetty distribution, the core server configuration is in `etc/jetty.xml` file, but you can mix in other server configuration which can include:

ThreadPool

The Server instance provides a ThreadPool instance that is the default Executor service other Jetty server components use. The prime configuration of the thread pool is the maximum and minimum size and is set in `etc/jetty.xml`.

Handlers

A Jetty server can have only a single Handler instance to handle incoming HTTP requests. However a handler may be a container or wrapper of other handlers forming a tree of handlers that typically handle a request as a collaboration between the handlers from a branch of the tree from root to leaf. The default handler tree set up in the `etc/jetty.xml` file is a Handler Collection containing a Context Handler Collection and the Default Handler. The Context Handler Collection selects the next handler by context path and is where deployed Context Handler and Web Application Contexts are added to the handler tree. The Default Handler handles any requests not already handled and generates the standard 404 page. Other configuration files may add handlers to this tree (for example, `jetty-rewrite.xml`, `jetty-requestlog.xml`) or configure components to hot deploy handlers (for example, `jetty-deploy.xml`).

Server Attributes

The server holds a generic attribute map of strings to objects so that other Jetty components can associate named objects with the server, and if the value objects implement the LifeCycle interface, they are started and stopped with the server. Typically server attributes hold server-wide default values.

Server fields

The server also has some specific configuration fields that you set in `etc/jetty.xml` for controlling among other things, the sending of dates and versions in HTTP responses.

Connectors

The server holds a collection of connectors that receive connections for HTTP and the other protocols that Jetty supports. The next section, the section called “Configuring Connectors” describes configuration of the connectors themselves. For the server you can either set the collection of all connectors or add/remove individual connectors.

Services

The server can hold additional service objects, sometimes as attributes, but often as aggregated LifeCycle beans. Examples of services are Login Services and DataSources, which you configure at the server level and then inject into the web applications that use them.

Configuring Connectors

A Jetty Server Connector is a network end point that accepts connections for one or more protocols which produce requests and/or messages for the Jetty server. In the standard Jetty server distribution, several provided configuration files add connectors to the server for various protocols and combinations of protocols: `jetty-http.xml`, `jetty-https.xml` and `jetty-spdy.xml`. The configuration needed for connectors is typically:

Port

The TCP/IP port on which the connector listens for connections is set using the XML Property element which looks up the `jetty.port` (or `jetty.tls.port`) property, and if not found defaults to 8080 (or 8443 for TLS).

Host

You can configure a host either as a host name or IP address to identify a specific network interface on which to listen. If not set, or set to the value of 0.0.0.0, the connector listens on all local interfaces. The XML Property element is used to look up the host value from the `jetty.host` property.

Idle Timeout

The time in milliseconds that a connection can be idle before the connector takes action to close the connection.

HTTP Configuration

Connector types that accept HTTP semantics (including HTTP, HTTPS and SPDY) are configured with a `HttpConfiguration` instance that contains common HTTP configuration that is independent of the specific wire protocol used. Because these values are often common to multiple connector types, the standard Jetty Server distribution creates a single `HttpConfiguration` in the `jetty.xml` file which is used via the XML Ref element in the specific connector files.

SSL Context Factory

The TLS connector types (HTTPS and SPDY) configure an SSL Context Factory with the location of the server keystore and truststore for obtaining server certificates.



Note

Virtual hosts are not configured on connectors. You must configure individual contexts with the virtual hosts to which they respond.



Note

Prior to Jetty 9, the type of the connector reflected both the protocol supported (HTTP, HTTPS, AJP, SPDY), and the nature of the implementation (NIO or BIO). From Jetty 9 onwards there is only one prime Connector type (`ServerConnector`), which is NIO based and uses Connection Factories to handle one or more protocols.

Configuring Contexts

A Jetty context is a handler that groups other handlers under a context path together with associated resources and is roughly equivalent to the standard `ServletContext` API. A context may contain either standard Jetty handlers or a custom application handler.



Note

The servlet specification defines a web application. In Jetty a standard web application is a specialized context that uses a standard layout and `WEB-INF/web.xml` to instantiate and configure classpath, resource base and handlers for sessions, security, and servlets, plus servlets for JSPs and static content. Standard web applications often need little or

no additional configuration, but you can also use the techniques for arbitrary contexts to refine or modify the configuration of standard web applications.

Configuration values that are common to all contexts are:

contextPath

The contextPath is a URL prefix that identifies which context a HTTP request is destined for. For example, if a context has a context path /foo, it handles requests to /foo, /foo/index.html, /foo/bar/, and /foo/bar/image.png but it does not handle requests like /, /other/, or /favicon.ico. A context with a context path of / is called the root context.

The context path can be set by default from the deployer (which uses the filename as the basis for the context path); or in code; or it can be set by a Jetty IoC XML that is either applied by the deployer or found in the WEB-INF/jetty-web.xml file of a standard web app context.

virtualHost

A context may optionally have one or more virtual hosts set. Unlike the host set on a connector (which selects the network interface on which to listen), a virtual host does not set any network parameters. Instead a virtual host represents an alias assigned by a name service to an IP address, which may have many aliases. To determine which virtual host a request is intended for, the HTTP client (browser) includes in the request the name used to look up the network address. A context with a virtual host set only handles requests that have a matching virtual host in their request headers.

classPath

A context may optionally have a classpath, so that any thread that executes a handler within the context has a thread context classloader set with the classpath. A standard web application has the classpath initialized by the WEB-INF/lib and WEB-INF/classes directory and has additional rules about delegating classloading to the parent classloader. All contexts may have additional classpath entries added.

attributes

Attributes are arbitrary named objects that are associated with a context and are frequently used to pass entities between a web application and its container. For example the attribute javax.servlet.context.tempdir is used to pass the File instance that represents the assigned temporary directory for a web application.

resourceBase

The resource base is a directory (or collection of directories or URL) that contains the static resources for the context. These can be images and HTML files ready to serve or JSP source files ready to be compiled. In traditional web servers this value is often called the docroot.

Context Configuration by API

In an embedded server, you configure contexts by directly calling the [ContextHandler](#) API as in the following example:

```
package org.eclipse.jetty.embedded;

import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.server.handler.ContextHandler;

public class OneContext {

    public static void main(String[] args) throws Exception {
        Server server = new Server(8080);
        ContextHandler context = new ContextHandler();
        context.setContextPath("/");
        context.setResourceBase(".");
        context.setClassLoader(Thread.currentThread().getContextClassLoader());
        context.setHandler(new HelloHandler());
```

```
        server.setHandler(context);
        server.start();
        server.join();
    }
```

Context Configuration by IoC XML

You can create and configure a context entirely by IoC XML (either Jetty's or Spring). The deployer discovers and hot deploys context IoC descriptors like the following which creates a context to serve the Javadoc from the Jetty distribution:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Configure PUBLIC
  "-//Mort Bay Consulting//DTD Configure//EN"
  "http://www.eclipse.org/jetty/configure_9_0.dtd">

<!--
  Configure a custom context for serving javadoc as static resources
-->

<Configure class="org.eclipse.jetty.server.handler.ContextHandler">
  <Set name="contextPath">/javadoc</Set>
  <Set name="resourceBase"><SystemProperty name="jetty.home" default="."/>/javadoc/</Set>
  <Set name="handler">
    <New class="org.eclipse.jetty.server.handler.ResourceHandler">
      <Set name="welcomeFiles">
        <Array type="String">
          <Item>index.html</Item>
        </Array>
      </Set>
      <Set name="cacheControl">max-age=3600,public</Set>
    </New>
  </Set>
</Configure>
```

Configuring Web Applications

The servlet specification defines a web application, which when packaged as a zip is called WAR file (Web application ARchive). Jetty implements both WAR files and unpacked web applications as a specialized context that is configured by means of:

- A standard layout which sets the location of the resourceBase (the root of the WAR) and initializes the classpath from jars found in WEB-INF/lib and classes found in WEB-INF/classes.
- The standard WEB-INF/web.xml deployment descriptor which is parsed to define and configure init parameters, filters, servlets, listeners, security constraints, welcome files and resources to be injected.
- A default web.xml format deployment descriptor provided either by Jetty or in configuration configures the JSP servlet and the default servlet for handling static content. The standard web.xml may override the default web.xml.
- Annotations discovered on classes in Jars contained in WEB-INF/lib can declare additional filters, servlets and listeners.
- Standard deployment descriptor fragments discovered in Jars contained in WEB-INF/lib can declare additional init parameters, filters, servlets, listeners, security constraints, welcome files and resources to be injected.
- An optional WEB-INF/jetty-web.xml file may contain Jetty IoC configuration to configure the Jetty specific APIs of the context and handlers.

Because these configuration mechanisms are contained within the WAR file (or unpacked web application), typically a web application contains much of its own configuration and deploying a WAR

is often just a matter of dropping the WAR file in to the webapps directory that is scanned by the [jetty deployer](#).

If you need to configure something within a web application, often you do so by unpacking the WAR file and editing the `web.xml` and other configuration files. However, both the servlet standard and some Jetty features allow for other configuration to be applied to a web application externally from the WAR:

- You configure datasources and security realms in the server and inject them into a web application either explicitly or by name matching.
- Jetty allows one or more override deployment descriptors, in `web.xml` format, to be set on a context (via code or IoC XML) to amend the configuration set by the default and standard `web.xml`.
- The normal Jetty Java API may be called by code or IoC XML to amend the configuration of a web application.

Setting the Context Path

The web application standard provides no configuration mechanism for a web application or WAR file to set its own `contextPath`. By default the deployer uses conventions to set the context path: If you deploy a WAR file called `foobar.WAR`, the context path is `/foobar`; if you deploy a WAR file called `ROOT.WAR` the context path is `/`. However, it is often desirable to explicitly set the context path so that information (for example, version numbers) may be included in the filename of the WAR. Jetty allows the context Path of a WAR file to be set internally (by the WAR itself) or externally (by the deployer of the WAR).

To set the `contextPath` from within the WAR file, you can include a `WEB-INF/jetty-web.xml` file which contains IoC XML to set the context path:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Configure PUBLIC
  "-//Mort Bay Consulting//DTD Configure//EN"
  "http://www.eclipse.org/jetty/configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath">/contextpath</Set>
</Configure>
```

Alternately, to configure the classpath externally without the need to modify the WAR file itself, instead of allowing the WAR file to be discovered by the deployer, an IoC XML file may be deployed that both sets the context path and declares the WAR file that it applies to:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Configure PUBLIC
  "-//Mort Bay Consulting//DTD Configure//EN"
  "http://www.eclipse.org/jetty/configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="war"><SystemProperty name="jetty.home" default="."/>/webapps/test.war</Set>
  <Set name="contextPath">/test</Set>
</Configure>
```

An example of setting the context path is included with the Jetty distribution in `$JETTY_HOME/webapps/test.xml`.

Setting an Authentication Realm

The authentication method and realm name for a standard web application may be set in the `web.xml` deployment descriptor with elements like:

```
...
<login-config>
```

```
<auth-method>BASIC</auth-method>
<realm-name>Test Realm</realm-name>
</login-config>
...
```

This example declares that the BASIC authentication mechanism will be used with credentials validated against a realm called "Test Realm." However the standard does not describe how the realm itself is implemented or configured. In Jetty, there are several realm implementations (called Login-Services) and the simplest of these is the HashLoginService, which can read usernames and credentials from a Java properties file.

To configure an instance of HashLoginService that matches the "Test Realm" configured above, the following \$JETTY_HOME/etc/test-realm.xml IoC XML file should be passed on the command line or set in start.ini.

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-// http://www.eclipse.org/jetty/configure_9_0.dtd">
<Configure id="Server" class="org.eclipse.jetty.server.Server">
    <!-- ===== -->
    <!-- Configure Authentication Login Service -->
    <!-- Realms may be configured for the entire server here, or -->
    <!-- they can be configured for a specific web app in a context -->
    <!-- configuration (see ${jetty.home}/webapps/test.xml for an -->
    <!-- example). -->
    <!-- ===== -->
    <Call name="addBean">
        <Arg>
            <New class="org.eclipse.jetty.security.HashLoginService">
                <Set name="name">Test Realm</Set>
                <Set name="config"><Property name="demo.realm" default="etc/realm.properties"/>
            </Set>
                <Set name="refreshInterval">0</Set>
            </New>
        </Arg>
    </Call>

    <Get class="org.eclipse.jetty.util.log.Log" name="rootLogger">
        <Call name="warn"><Arg>demo test-realm is deployed. DO NOT USE IN PRODUCTION!</Arg></Call>
        <Get>
    </Get>
</Configure>
```

This creates and configures the LoginService as an aggregate bean on the server. When a web application is deployed that declares a realm called "Test Realm," the server beans are searched for a matching Login Service.

Web Application Deployment

Jetty is capable of deploying a variety of Web Application formats. This is accomplished via scans of the \${jetty.home}/webapps directory for contexts to deploy.

A Context can be any of the following:

- A standard WAR file. (must in ".war").
- A directory containing an expanded WAR file. (must contain {dir}/WEB-INF/web.xml file).
- A directory containing static content.
- A XML descriptor in the section called “Jetty XML Syntax” that configures a [ContextHandler](#) instance (Such as a [WebAppContext](#)).

The new WebAppProvider will attempt to avoid double deployments during the directory scan with the following heuristics:

- Hidden files (starting with ".") are ignored

- Directories with names ending in ".d" are ignored
- If a directory and matching WAR file exist with the same base name (eg: `foo/` and `foo.war`), then the directory is assumed to be the unpacked WAR and only the WAR is deployed (which may reuse the unpacked directory)
- If a directory and matching XML file exists (eg: `foo/` and `foo.xml`), then the directory is assumed to be an unpacked WAR and only the XML is deployed (which may use the directory in its own configuration)
- If a WAR file and matching XML file exist (eg: `foo.war` and `foo.xml`), then the WAR is assumed to be configured by the XML and only the XML is deployed.



Note

In prior versions of Jetty there was a separate ContextDeployer that provided the XML based deployment, the ContextDeployer no longer exists starting in Jetty9 and its functionality has been merged with the new [WebAppProvider](#) in to avoid several double deployment scenarios that are possible.

A Context is an instance of ContextHandler that aggregates other handlers with common resources for handling HTTP requests (such as resource base, class loader, configuration attributes). A standard web application is a specialized instance of a context (called a WebAppContext) that uses standard layouts and `web.xml` deployment descriptors to configure the context.

Part II. Jetty Configuration

Table of Contents

4. Deploying to Jetty	26
Anatomy of a Web Application	26
Automatic Web Application Deployment	26
Configuring a Specific Web Application Deployment	27
Deployment Processing of WebAppContexts	29
Configuring Static Content Deployment	34
Hot Deployment	34
Deployment Architecture	35
Quickstart Webapps	39
Overlay WebApp Deployer	40
5. Configuring Contexts	47
Setting a Context Path	47
Configuring Virtual Hosts	48
Temporary Directories	51
Serving a WebApp from a Particular Port/Connector	54
Creating Custom Error Pages	55
Setting Max Form Size	58
6. Configuring Jetty Connectors	59
Connector Configuration Overview	59
Configuring SSL	65
Setting Port 80 Access for a Non-Root User	72
7. Configuring Security	76
Jetty 9.1: Using the \${jetty.home} and \${jetty.base} Concepts to Configure Security.....	76
Authentication	83
Limiting Form Content	89
Aliased Files and Symbolic links	90
Secure Password Obfuscation	91
JAAS Support	93
Spnego Support	99
8. Configuring JSP Support	103
Configuring JSP	103

Chapter 4. Deploying to Jetty

Table of Contents

Anatomy of a Web Application	26
Automatic Web Application Deployment	26
Configuring a Specific Web Application Deployment	27
Deployment Processing of WebAppContexts	29
Configuring Static Content Deployment	34
Hot Deployment	34
Deployment Architecture	35
Quickstart Webapps	39
Overlay WebApp Deployer	40

This chapter discusses various ways to deploy applications with Jetty. Topics range from deployment bindings to deploying third party products. It also includes information about the Deployment Manager, WebApp Provider, and Overlay Deployer.

Anatomy of a Web Application

The standard Jetty distribution is capable of deploying standard Servlet Spec Web Applications, and Jetty internal ContextHandler deployment descriptors, or even a mix of the two.

Web Applications are deployable collections of dynamic (servlets, filters, jsps, etc..) and static content, support libraries, and descriptive metadata that are bound to a specific context path on Jetty.

Ultimately the format and layout are defined by the Servlet Spec, and you should consult the official Servlet Spec documentation for a more detailed understanding of Web Application layout and structure, however this will outline basics about how Jetty views these requirements.

Web Applications can be bundled into a single Web Archive (WAR file) or as a directory tree.

/WEB-INF/

Special Servlet API defined directory used to store anything related to the Web Application that are not part of the public access of the Web Application.

If you have content that is accessed by your Web Application internally, but that should also never be accessed directly by a web browser, this is the directory you would place them in.

/WEB-INF/web.xml

Required deployment descriptor defining various behavior of your Web Application.

/WEB-INF/classes/

Location for Web Application specific compiled java classes

/WEB-INF/lib/

Directory for JAR files (libraries)

The Jetty internal WebAppClassLoader will load classes from /WEB-INF/classes/ first, then from jar files found in /WEB-INF/lib/.

Automatic Web Application Deployment

The most basic technique for deploying Web Applications is to simply put your WAR file or Exploded WAR directory into the \${jetty.home}/webapps/ directory and let Jetty's deployment scanner find it and deploy it under a Context path of the same name.

Only Web Applications that follow the Web Application Layout will be detected by Jetty and deployed this way.

The Context Path assigned to this automatic deployment is based the filename (or directory name) of your WAR.

File or Directory Name	Assigned Context Path
/webapps/footrope.war	http://host/footrope/
/webapps/baggywrinkle-1.0.war	http://host/baggywrinkle-1.0/
/webapps/lazaret-2.1.3-SNAPSHOT.war	http://host/lazaret-2.1.3-SNAPSHOT/
/webapps/belaying-pins/WEB-INF/web.xml	http://host/belaying-pins/
/webapps/root.war (<i>special name</i>)	http://host/
/webapps/root/WEB-INF/web.xml (<i>special name</i>)	http://host/

Configuring a Specific Web Application Deployment

Using the Automatic Web Application Deployment model is quick and easy, but sometimes you might need to tune certain deployment properties (for example, you want to deploy with a context path that is not based on the file name, or you want to define a special database connection pool just for this web application). You can use a the section called “Jetty Deployable Descriptor XML File” to accomplish such tuning.

Jetty Deployable Descriptor XML File

Jetty supports deploying Web Applications via XML files which will build an instance of a [ContextHandler](#) that Jetty can then deploy.

Using Basic Descriptor Files

In a default Jetty installation, Jetty scans its \$JETTY_HOME/webapps directory for context deployment descriptor files. To deploy a web application using such a file, simply place the file in that directory.

The deployment descriptor file itself is an xml file that configures a [WebAppContext](#) class. For a basic installation you probably need to set only two properties:

war

the filesystem path to the web application file (or directory)

contextPath

the context path to use for the web application

For example, here is a descriptor file that deploys the file /opt/myapp/myapp.war to the context path /wiki:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath"/>/wiki</Set>
  <Set name="war"/>/opt/myapp/myapp.war</Set>
</Configure>
```



Note

You can use the `SystemProperty` and `Property` elements in your descriptor file. For example, if you set the system property `myapp.home=/opt/myapp`, you can rewrite the previous example as:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath">/wiki</Set>
  <Set name="war"><SystemProperty name="myapp.home"/>/myapp.war</Set>
</Configure>
```

If you need to change the home path for your application, you can simply change the system property. This is useful if you frequently switch among multiple versions of an app.

Configuring Advanced Descriptor Files

If you look at the documentation for the [WebAppContext](#) class, you notice that there are a lot more properties than just the two mentioned above. Here are some examples that configure advanced options with your descriptor file.

This first example tells Jetty not to expand the WAR file when deploying it. This can help make it clear that people should not be making changes to the temporary unpacked WAR because such changes do not persist, and therefore do not apply the next time the web application deploys.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath">/wiki</Set>
  <Set name="war"><SystemProperty name="myapp.home"/>/myapp.war</Set>
  <Set name="extractWAR">false</Set>
</Configure>
```

The next example retrieves the JavaEE Servlet context and sets an initialization parameter on it. You can also use the `setAttribute` method to set a Servlet context attribute. However, since the `web.xml` for the web application is processed after the deployment descriptor, the `web.xml` values overwrite identically named attributes from the deployment descriptor.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath">/wiki</Set>
  <Set name="war"><SystemProperty name="myapp.home"/>/myapp.war</Set>
  <Get name="ServletContext">
    <Call name="setInitParameter">
      <Arg>myapp.config</Arg>
      <Arg><SystemProperty name="myapp.home"/>/config/app-config.xml</Arg>
    </Call>
  </Get>
</Configure>
```

The following example sets a special `web.xml` override descriptor. This descriptor is processed after the web application's `web.xml`, so it may override identically named attributes. This feature is useful if you want to add parameters or additional Servlet mappings without breaking open a packed WAR file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">
```

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath">/wiki</Set>
  <Set name="war"><SystemProperty name="myapp.home"/>/myapp.war</Set>
  <Set name="overrideDescriptor">/opt/myapp/overlay-web.xml</Set>
</Configure>
```

The next example configures not only the web application context, but also a database connection pool (see the section called “Datasource Examples” that our application can then use. If the `web.xml` does not include a reference to this data source, you can use the override descriptor mechanism (the previous example), to include it.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath">/wiki</Set>
  <Set name="war"><SystemProperty name="myapp.home"/>/myapp.war</Set>
</Configure>

<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
  <Arg></Arg>
  <Arg>jdbc/DSTest</Arg>
  <Arg>
    <New class="org.apache.commons.dbcp.BasicDataSource">
      <Set name="driverClassName">org.some.Driver</Set>
      <Set name="url">jdbc.url</Set>
      <Set name="username">jdbc.user</Set>
      <Set name="password">jdbc.pass</Set>
    </New>
  </Arg>
</New>
```

There are many other settings that you can change on a `WebAppContext`. The [javadoc](#) for `WebAppContext` is a good source of information. Also see the documentation on [avoiding zip file exceptions](#) for a description of `WebAppContext` settings that determine such things as whether or not the war is automatically unpacked during deployment, or whether certain sections of a webapp are copied to a temporary location.

Deployment Processing of WebAppContexts

Web applications require a certain amount of processing before they can go into service: they may need to be unpacked, a special classloader created for their jars, `web.xml` and `web-fragment.xml` descriptors processed, and classes scanned for annotations amongst many other things. As web applications have become more complex, we have added ways to help you customize by either broadening or lessening the amount of processing that is done at deployment time. In this section we will look at this processing, and how you can tailor it.

If instead you're looking for information on how to configure a specific `WebAppContext` - such as its context path, whether it should be unpacked or not - then you can find that in the section entitled [Configuring a Specific WebApp Deployment](#).

Configuration Classes

As a webapp is being deployed, a series of [org.eclipse.jetty.webapp.Configuration](#) classes are applied to it, each one performing a specific function. The ordering of these Configurations is significant as subsequent Configurations tend to build on information extracted or setup in foregoing Configurations. These are the default list, in order, of Configurations that are applied to each [org.eclipse.jetty.webapp.WebAppContext](#):

Table 4.1. Default Configuration classes

org.eclipse.jetty.webapp.WebInfConfiguration	Extracts war, orders jars and defines classpath
--	---

org.eclipse.jetty.webapp.WebXmlConfiguration	Processes a WEB-INF/web.xml file
org.eclipse.jetty.webapp.MetaInfConfiguration	Looks in container and webapp jars for META-INF/resources and META-INF/web-fragment.xml
org.eclipse.jetty.webapp.FragmentConfiguration	Processes all discovered META-INF/web-fragment.xml files
org.eclipse.jetty.webapp.JettyWebXmlConfiguration	Processes a WEB-INF/jetty-web.xml file

Anatomy of a Configuration Class

A Configuration class is called 5 times in different phases of the WebAppContext's lifecycle:

preConfigure

As the WebAppContext is starting up this phase is executed. The Configuration should discover any of the resources it will need during the subsequent phases.

configure

This phase is where the work of the class is done, usually using the resources discovered during the preConfigure phase.

postConfigure

This phase allows the Configuration to clear down any resources that may have been created during the previous 2 phases that are not needed for the lifetime of the WebAppContext.

deconfigure

This phase occurs whenever a WebAppContext is being stopped and allows the Configuration to undo any resources/metadata that it created. A WebAppContext should be able to be cleanly start/stopped multiple times without resources being held.

destroy

This phase is called when a WebAppContext is actually removed from service. For example, the war file associated with it is deleted from the \$JETTY_HOME/webapps directory.

Each phase is called on each Configuration class in the order in which the Configuration class is listed. So for example, using our default Configuration classes as an example, preConfigure() will be called on WebInfConfiguration, WebXmlConfiguration, MetaInfConfiguration, FragmentConfiguration and then JettyWebXmlConfiguration. The cycle begins again for the configure() phase and again for the postConfigure() phases. The cycle is repeated *in reverse order* for the deconfigure() and eventually the destroy() phases.

Extending Container Support by Creating Extra Configurations

As we have seen, there is a default set of Configurations that support basic deployment of a webapp. You will notice that we have not mentioned JavaEE features such as JNDI, nor advanced servlet spec features such as annotations. That is because Jetty's philosophy is to allow the user to tailor the container exactly to his needs. If you do not need these kind of features, then you do not pay the price of them - an important consideration because features such as annotations require extensive and time-consuming scanning of WEB-INF/lib jars. As modern webapps may have scores of these jars, it can be a source of significant deployment delay. We will see in the section [Other Configuration](#) another helpful webapp facility that Jetty provides for cutting down the time spent analysing jars.

Jetty makes use of the flexibility of Configurations to make JNDI and annotation support pluggable.

Firstly, lets look at how Configurations help enable JNDI.

Example: JNDI Configurations

JNDI lookups within web applications require the container to hookup resources defined in the container's environment to that of the web application. To achieve that, we use 2 extra Configurations:

Table 4.2. JNDI Configuration classes

org.eclipse.jetty.plus.webapp.EnvConfiguration	Creates java:comp/env for the webapp, applies a WEB-INF/jetty-env.xml file
org.eclipse.jetty.plus.webapp.PlusConfiguration	Processes JNDI related aspects of WEB-INF/web.xml and hooks up naming entries

These configurations must be added in *exactly* the order shown above and should be inserted *immediately before* the [org.eclipse.jetty.webapp.JettyWebXmlConfiguration](#) class in the list of configurations. To fully support JNDI, you need to do a couple of other things, full details of which you can find [here](#).

Example: Annotation Configurations

We need just one extra Configuration class to help provide servlet annotation scanning:

Table 4.3. Annotation Configuration classes

org.eclipse.jetty.annotations.AnnotationConfiguration	Scans container and web app jars looking for @WebServlet, @WebFilter, @WebListener etc
---	--

The above configuration class must be *inserted immediately before* the [org.eclipse.jetty.webapp.JettyWebXmlConfiguration](#) class in the list of configurations. To fully support annotations, you need to do a couple of other things, details of which can be found [here](#).

How to Set the List of Configurations

You have a number of options for how to make Jetty use a different list of Configurations.

Setting the list directly on the WebAppContext

If you have only one webapp that you wish to affect, this may be the easiest option. You will, however, either need to have a context xml file that represents your web app, or you need to call the equivalent in code. Let's see an example of how we would add in the Configurations for both JNDI *and* annotations:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">

    <Set name="war"><SystemProperty name="jetty.home" default=". "/>/webapps/my-cool-
webapp</Set>

    <Set name="configurationClasses">
        <Array type="java.lang.String">
            <Item>org.eclipse.jetty.webapp.WebInfConfiguration</Item>
            <Item>org.eclipse.jetty.webapp.WebXmlConfiguration</Item>
            <Item>org.eclipse.jetty.webapp.MetaInfConfiguration</Item>
            <Item>org.eclipse.jetty.webapp.FragmentConfiguration</Item>
            <Item>org.eclipse.jetty.plus.webapp.EnvConfiguration</Item>
            <Item>org.eclipse.jetty.plus.webapp.PlusConfiguration</Item>
            <Item>org.eclipse.jetty.annotations.AnnotationConfiguration</Item>
            <Item>org.eclipse.jetty.webapp.JettyWebXmlConfiguration</Item>
        </Array>
    </Set>
</Configure>
```

Of course, you can also use this method to reduce the Configurations applied to a specific WebApp-Context.

Setting the list for all webapps via the Deployer

If you use the [deployer](#), you can set up the list of Configuration classes on the [WebAppProvider](#). They will then be applied to each WebAppContext deployed by the deployer:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure id="Server" class="org.eclipse.jetty.server.Server">

    <Call name="addBean">
        <Arg>
            <New id="DeploymentManager" class="org.eclipse.jetty.deploy.DeploymentManager">
                <Set name="contexts">
                    <Ref refid="Contexts" />
                </Set>
                <Call id="webappprovider" name="addAppProvider">
                    <Arg>
                        <New class="org.eclipse.jetty.deploy.providers.WebAppProvider">
                            <Set name="monitoredDirName"><Property name="jetty.home" default="." />
                        </Set>
                    </New>
                    <Set name="configurationClasses">
                        <Array type="java.lang.String">
                            <Item>org.eclipse.jetty.webapp.WebInfConfiguration</Item>
                            <Item>org.eclipse.jetty.webapp.WebXmlConfiguration</Item>
                            <Item>org.eclipse.jetty.webapp.MetaInfConfiguration</Item>
                            <Item>org.eclipse.jetty.webapp.FragmentConfiguration</Item>
                            <Item>org.eclipse.jetty.plus.webapp.EnvConfiguration</Item>
                            <Item>org.eclipse.jetty.plus.webapp.PlusConfiguration</Item>
                            <Item>org.eclipse.jetty.annotations.AnnotationConfiguration</Item>
                            <Item>org.eclipse.jetty.webapp.JettyWebXmlConfiguration</Item>
                        </Array>
                    </Set>
                </New>
            </Arg>
        </Call>
    </New>
    <Arg>
        <Call>
            <Arg>
                <Call>
                    <Arg>
                        <Call>
                            <Arg>
                                <Configure>

```

Adding or inserting to an existing list

Instead of having to enumerate the list in its entirety, you can simply nominate classes that you want to add, and indicate whereabouts in the list you want them inserted. Let's look at an example of using this method to add in Configuration support for JNDI - as usual you can either do this in an xml file, or via equivalent code. This example uses an xml file, in fact it is the \$JETTY_HOME/etc/jetty-plus.xml file from the Jetty distribution:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure id="Server" class="org.eclipse.jetty.server.Server">

    <!-- ===== -->
    <!-- Add plus Configuring classes to all webapps for this Server -->
    <!-- ===== -->
    <Call class="org.eclipse.jetty.webapp.Configuration$ClassList" name="setServerDefault">
        <Arg><Ref refid="Server" /></Arg>
        <Call name="addAfter">
            <Arg name="afterClass">org.eclipse.jetty.webapp.FragmentConfiguration</Arg>
            <Arg>
                <Array type="String">
                    <Item>org.eclipse.jetty.plus.webapp.EnvConfiguration</Item>
                    <Item>org.eclipse.jetty.plus.webapp.PlusConfiguration</Item>
                </Array>
            </Arg>
        </Call>
    </Call>

</Configure>
```

The [org.eclipse.jetty.webapp.Configuration.ClassList](#) class provides these methods for insertion:

addAfter

inserts the supplied list of Configuration class names after the given Configuration class name

addBefore

inserts the supplied list of Configuration class names before the given Configuration class name

Other Configuration

org.eclipse.jetty.server.webapp.ContainerIncludeJarPattern

This is a [context attribute](#) that can be set on [an org.eclipse.jetty.webapp.WebAppContext](#) to control which parts of the *container's* classpath should be processed for things like annotations, META-INF/resources, META-INF/web-fragment.xml etc.

Normally, nothing from the container classpath will be included for processing. However, sometimes you will need to include some. For example, you may have some libraries that are shared amongst your webapps and thus you have put them into a \$JETTY_HOME/lib directory. The libraries contain annotations and therefore must be scanned.

The value of this attribute is a regexp that defines which *jars* and *class directories* from the container's classpath should be examined.

Here's an example from a context xml file (although as always, you could have accomplished the same in code), which would match any jar whose name starts with "foo-" or "bar-", or a directory named "classes":

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">

    <Call name="setContextAttribute">
        <Arg>org.eclipse.jetty.server.webapp.ContainerIncludeJarPattern</Arg>
        <Arg>.*/foo-[^/]*\.jar$|.*/bar-[^/]*\.jar$|.*/classes/.*</Arg>
    </Call>

</Configure>
```

Note that the order of the patterns defines the ordering of the scanning of the jars or class directories.

org.eclipse.jetty.server.webapp.WebInflIncludeJarPattern

Similarly to the previous [context attribute](#), this attribute controls which jars are processed for things like annotations and META-INF resources. However, this attribute controls which jars from the *webapp's* classpath (usually WEB-INF/lib) are processed. This can be particularly useful when you have dozens of jars in WEB-INF/lib, but you know that only a few need to be scanned.

Here's an example (in an xml file, but you can do the same in code) of a pattern that matches any jar that starts with "spring-":

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">

    <Call name="setContextAttribute">
        <Arg>org.eclipse.jetty.server.webapp.WebInflIncludeJarPattern</Arg>
        <Arg>.*/spring-[^/]*\.jar$</Arg>
    </Call>

</Configure>
```

Note that the order of the patterns defines the ordering of the scanning of the jars.

Configuring Static Content Deployment

Should you want to serve purely static content, you can use the Jetty Deployment Descriptor XML concepts and the internal ResourceHandler to setup simple serving of static content. Just create a file called `scratch.xml` in the `${jetty.home} /webapps` directory and paste these file contents in it.

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN" "http://www.eclipse.org/jetty/configure.dtd">
<Configure class="org.eclipse.jetty.server.handler.ContextHandler">
  <Set name="contextPath"/>/scratch</Set>
  <Set name="handler">
    <New class="org.eclipse.jetty.server.handler.ResourceHandler">
      <Set name="resourceBase">/home/jesse/scratch</Set>
      <Set name="directoriesListed">true</Set>
    </New>
  </Set>
</Configure>
```

This is a very basic setup for serving static files, if you want advanced static file serving, use the [DefaultServlet](#).

Hot Deployment

Jetty lets you deploy an arbitrary context or web application by monitoring a directory for changes. If you add a web application or a context descriptor to the directory, Jetty's DeploymentManager (DM) deploys a new context. If you touch or update a context descriptor, the DM stops, reconfigures, and redeploys its context. If you remove a context, the DM stops it and removes it from the server.

To control this behavior, you'll want to configure some WebAppProvider properties.

monitoredDirName

The directory to scan for possible deployable Web Applications (or Deployment Descriptor XML files)

scanInterval

Number of seconds between scans of the provided monitoredDirName.

A value of 0 disables the continuous hot deployment scan, Web Applications will be deployed on startup only.

The default location for this configuration is in the `${jetty.home} /etc/jetty-deploy.xml` file.

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/configure_9_0.dtd">
<Configure id="Server" class="org.eclipse.jetty.server.Server">

  <Call name="addBean">
    <Arg>
      <New id="DeploymentManager" class="org.eclipse.jetty.deploy.DeploymentManager">
        <Set name="contexts">
          <Ref refid="Contexts" />
        </Set>
        <Call name="setContextAttribute">
          <Arg>org.eclipse.jetty.server.webapp.ContainerIncludeJarPattern</Arg>
          <Arg>.*/servlet-api-[^/]*\.\jar$</Arg>
        </Call>
      </New>
    </Arg>
  </Call>

  <Call id="webappprovider" name="addAppProvider">
```

```
<Arg>
  <New class="org.eclipse.jetty.deploy.providers.WebAppProvider">
    <Set name="monitoredDirName"><Property name="jetty.home" default="." />/webapps</Set>
    <Set name="defaultsDescriptor"><Property name="jetty.home" default="." />/etc/webdefault.xml</Set>
    <Set name="scanInterval">1</Set>
    <Set name="extractWars">true</Set>
  </New>
</Arg>
</Call>
</New>
</Arg>
</Call>
</Configure>
```

See the section called “Understanding the Default WebAppProvider” for more configuration details.

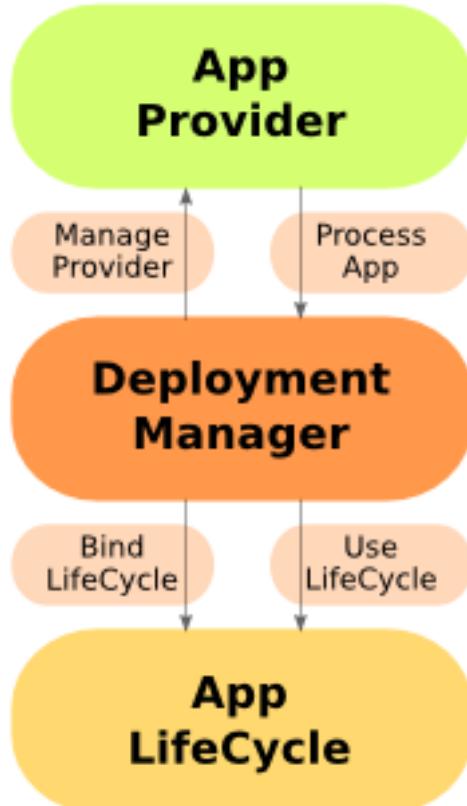
See also the section called “Deployment Architecture” for detailed conceptual information.

Deployment Architecture

Jetty is built around an extensible Deployment Manager architecture complete with formal LifeCycle for Web Applications going through it.

For Jetty to serve content (static or dynamic), you need to create a [ContextHandler](#) and add it to Jetty in the appropriate place. A pluggable DeploymentManager exists in Jetty 7 and later to make this process easier. The Jetty distribution contains example DeploymentManager configurations to deploy WAR files found in a directory to Jetty, and to deploy Jetty context .xml files into Jetty as well.

The DeploymentManager is the heart of the typical webapp deployment mechanism; it operates as a combination of an Application LifeCycle Graph, Application Providers that find and provide Applications into the Application LifeCycle Graph, and a set of bindings in the graph that control the deployment process.

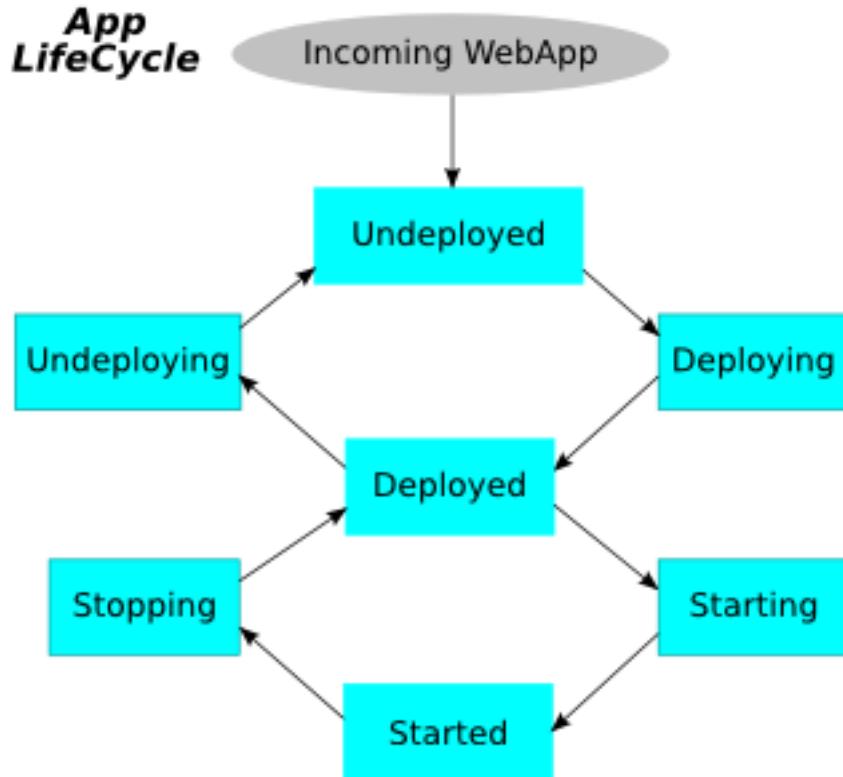


Application Providers

Before Jetty deploys an application, an [AppProvider](#) identifies the App and then provides it to the DeploymentManager. The main AppProvider with the Jetty distribution is the [WebAppProvider](#).

Application LifeCycle Graph

The core feature of the DeploymentManager is the [Application LifeCycle Graph](#).



The nodes and edges of this graph are pre-defined in Jetty along the most common actions and states found. These nodes and edges are not hardcoded; you can adjust and add to them depending on your needs (for example, any complex requirements for added workflow, approvals, staging, distribution, coordinated deploys for a cluster or cloud, etc.).

New applications enter this graph at the Undeployed node, and the [`java.lang.String DeploymentManager.requestAppGoal\(App, String\)`](#) method pushes them through the graph.

LifeCycle Bindings

A set of default [AppLifecycle.Bindings](#) defines standard behavior, and handles deploying, starting, stopping, and undeploying applications. If you choose, you can write your own `AppLifecycle.Bindings` and assign them to anywhere on the Application LifeCycle graph.

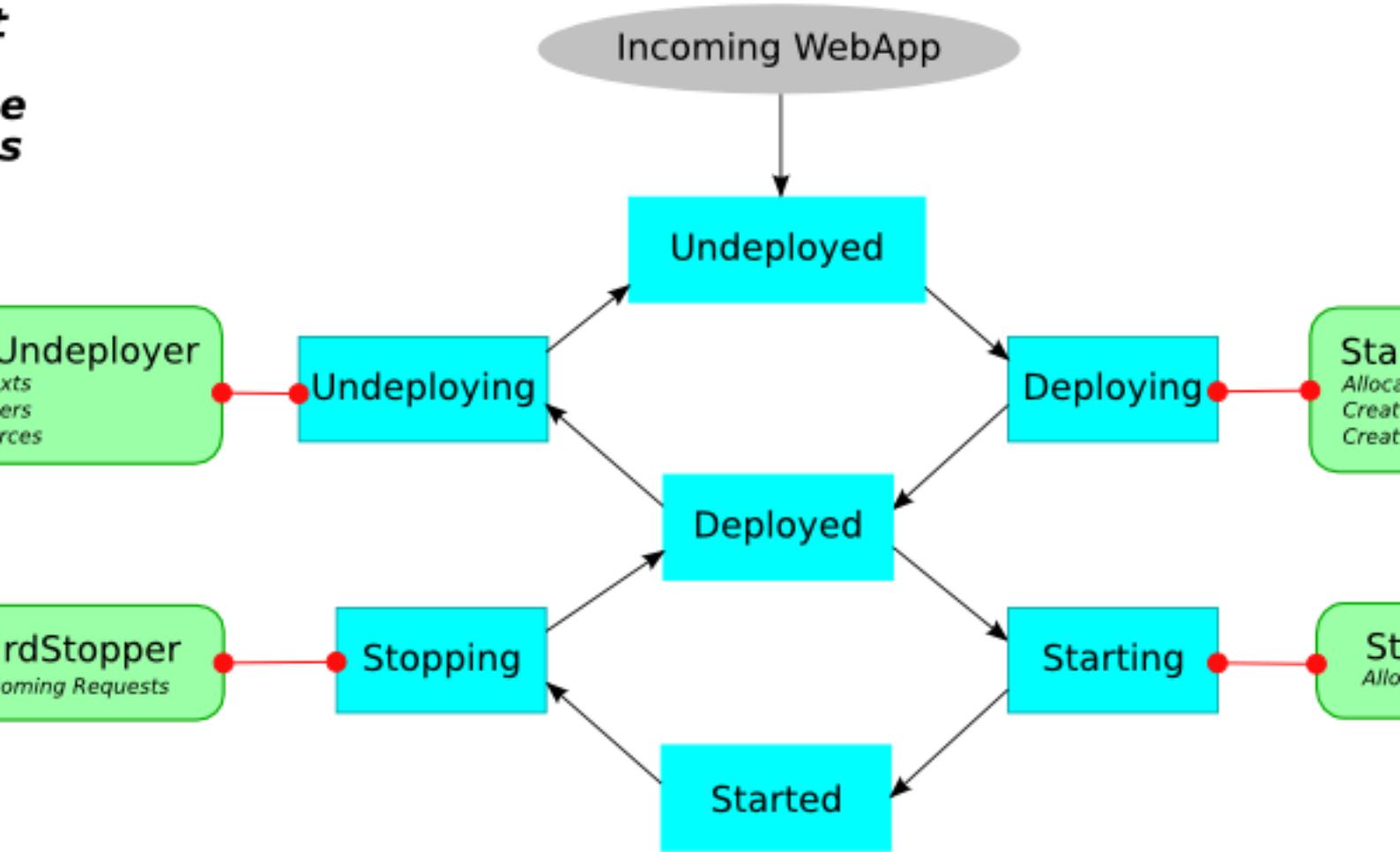
Examples of new `AppLifecycle.Binding` implementations that you can write include:

- Validating the incoming application.
- Preventing the deployment of known forbidden applications.
- Submitting the installation to an application auditing service in a corporate environment.

- Distributing the application to other nodes in the cluster or cloud.
- Emailing owner/admin of change of state of the application.

There are four default bindings:

- [StandardDeployer](#) â##Deploys the ContextHandler into Jetty in the appropriate place.
- [StandardStarter](#) â##Sets the ContextHandler to started and start accepting incoming requests.
- [StandardStopper](#) â##Stops the ContextHandler and stops accepting incoming requests.
- [StandardUndeployer](#) â##Removes the ContextHandler from Jetty.



A fifth, non-standard binding, called [Debug Binding](#), is also available for debugging reasons; It logs the various transitions through the Application LifeCycle.

Understanding the Default WebAppProvider

The [WebAppProvider](#) is for the deployment of Web Applications packaged as WAR files, expanded as a directory, or declared in a the section called “Jetty Deployable Descriptor XML File”. It supports hot (re)deployment.

The basic operation of the WebAppProvider is to periodically scan a directory for deployables. In the standard Jetty Distribution, this is configured in the `${jetty.home}/etc/jetty-deploy.xml` file.

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">
<Configure id="Server" class="org.eclipse.jetty.server.Server">

    <Call name="addBean">
        <Arg>
            <New id="DeploymentManager" class="org.eclipse.jetty.deploy.DeploymentManager">
                <Set name="contexts">
                    <Ref refid="Contexts" />
                </Set>
                <Call id="webappprovider" name="addAppProvider">
                    <Arg>
                        <New class="org.eclipse.jetty.deploy.providers.WebAppProvider">
                            <Set name="monitoredDirName"><Property name="jetty.home" default="." />
                        webapps</Set>
                            <Set name="defaultsDescriptor"><Property name="jetty.home" default="." />
                        etc/webdefault.xml</Set>
                            <Set name="scanInterval">1</Set>
                            <Set name="extractWars">true</Set>
                        </New>
                    </Arg>
                </Call>
            </New>
        </Arg>
    </Call>
</Configure>
```

The above configuration will create a DeploymentManager tracked as a Server LifeCycle Bean, with the following configuration.

contexts

A passed in reference to the HandlerContainer into which the discovered webapps are deployed. This is normally a reference that points to the id="Contexts" found in the \${jetty.home}/etc/jetty.xml file, which itself is an instance of ContextHandlerCollection.

monitoredDirName

Is a file path or URL to the directory to scan for web applications.

Scanning follows these rules:

1. Base directory must exist
2. Hidden Files (starting with ".") are ignored
3. Directories with names ending in ".d" are ignored.
4. Common CVS directories "CVS" and "CVSROOT" are ignored
5. Any *.war files are considered **automatic deployables**
6. Any *.xml files are considered **context descriptor deployables**
7. In the special case where both a WAR file and XML file exists for same base name, then the WAR file is flagged as not-deployable, and the XML file is assumed to configure and reference the WAR file. (see the section called "Configuring a Specific Web Application Deployment")
8. And directory is considered to be deployable
9. In the special case where both a Directory and WAR file of the same name exists, the directory is flagged as not-deployable, and the WAR file is assumed to be and automatic deployable.
10. In the special case where both a Directory and XML file of the same name exists, the directory is flagged as not-deployable, and the XML file is assumed to configure and reference the Directory.

11.All other directories are subject to automatic deployment.

12.If automatic deployment is used, and the special filename `root.war` or directory name `root` will result in a deployment to the " / " context path.

defaultsDescriptor

Specifies the default Servlet web descriptor to use for all Web Applications. The intent of this descriptor is to include common configuration for the Web Application before the Web Application's own /WEB-INF/web.xml is applied. The \${jetty.home}/etc/webdefault.xml that comes with the Jetty distribution controls the configuration of the JSP and Default servlets, along with mimetypes and other basic metadata.

scanInterval

Is the period in seconds between sweeps of the monitoredDirName for changes: new contexts to deploy, changed contexts to redeploy, or removed contexts to undeploy.

extractWars

If parameter is true, any packed WAR or zip files are first extracted to a temporary directory before being deployed. This is advisable if there are uncompiled JSPs in the web apps.

parentLoaderPriority

parameter is a boolean that selects whether the standard Java [parent first delegation](#) is used or the [servlet specification webapp classloading priority](#). The latter is the default.

Quickstart Webapps

The auto discovery features of the Servlet specification can make deployments slow and uncertain. Auto discovery of Web Application configuration can be useful during the development of a webapp as it allows new features and frameworks to be enabled simply by dropping in a jar file. However, for deployment, the need to scan the contents of many jars can have a significant impact of the start time of a webapp.

From Jetty release 9.2.0.v20140526, we have included the quickstart module that allows a webapp to be pre-scanned and preconfigured. This means that all the scanning is done prior to deployment and all configuration is encoded into an effective web.xml, called WEB-INF/quickstart-web.xml, which can be inspected to understand what will be deployed before deploying. Not only does the quickstart-web.xml contain all the discovered Servlets, Filters and Constraints, but it also encodes all context parameters all discovered:

- `ServletContainerInitializers`
- `HandlesTypes` classes
- Taglib Descriptors

With the quickstart mechanism, jetty is able to entirely bypass all scanning and discovery modes and start a webapp in a predictable and fast way. Tests have shown that webapps that took many seconds to scan and deploy can now be deployed in a few hundred milliseconds.

Setting up Quickstart

To use quickstart the module has to be available to your jetty instance. In a maven project this is done just by adding a dependency on the artifact ID `jetty-quickstart` or with a standard jetty distribution you can run the command:

```
$ java -jar $JETTY_HOME/start.jar --add-to-startd=quickstart
```

Also the webapps you deploy need to be instances of [`org.eclipse.jetty.quickstart.QuickStartWebApp`](#) rather than the normal

`org.eclipse.jetty.webapp.WebAppContext`. If your web application already has a `webapps/myapp.xml` file, then you can simply change the class in the `Configure` element, otherwise you can create an `webapps/myapp.xml` file as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">
<Configure class="org.eclipse.jetty.quickstart.QuickStartWebApp">
  <Set name="war"><Property name="jetty.webapps" default=". "/>/benchmark.war</Set>
  <Set name="contextPath">/benchmark</Set>
  <Set name="autoPreconfigure">true</Set>
</Configure>
```

Preconfiguring the web application

If the `QuickStateWebApp` method `setAutoPreconfigure(true)` is called (see example in `myapp.xml` above), then the first time the webapp is deployed a `WEB-INF/quickstart-web.xml` file will be generated that contains the effective `web.xml` for all the discovered configuration. On subsequent deployments, all the discovery steps are skipped and the `quickstart-web.xml` is used directly to configure the web Application.

It is also possible to preconfigure a war file manually by running the class [org.eclipse.jetty.quickstart.PreconfigureQuickStartWar](#) as a main, down simply with the jetty-all aggregate jar:

```
$ java -cp jetty-all-9.2.0.jar:servlet.jar
org.eclipse.jetty.quickstart.PreconfigureQuickStartWar myapp.war
```

This will create the `quickstart-web.xml` file before the first deployment. Note that this can also be a good debugging tool for discovered configuration and if run with debug turned on the origin of every element is included in the `quickstart-web.xml` file. Run the class with no arguments to see other options for running it.

Avoiding TLD Scans with precompiled JSPs

Of course precompiling JSPs is an excellent way to improve the start time of a web application. Since jetty 9.2.0, the apache Jasper JSP implementation has been used and has been augmented to allow the TLD scan to be skipped. This can be done by adding a context-param to the `web.xml` file (this is done automatically by the Jetty Maven JSPC plugin):

```
<context-param>
  <param-name>org.eclipse.jetty.jsp.precompiled</param-name>
  <param-value>true</param-value>
</context-param>
```

Bypassing start.jar

The jetty start.jar mechanism is a very powerful and flexible mechanism for constructing a classpath and executing a configuration encoded in jetty XML format. However, this mechanism does take some time to build the classpath. The start.jar mechanism can be bypassed by using the **--dry-run** option to generate and reuse a complete command line to start jetty at a later time:

```
$ RUN=$(java -jar $JETTY_HOME/start.jar --dry-run)
$ eval $RUN
```

Note that **--dry-run** may create a properties file in the temp directory and include it on the generated command line. If so, then a copy of the temporary properties file should be taken and the command line updated with its new persistent location.

Overlay WebApp Deployer



Note

This feature is reintroduced in Jetty 9.0.4

The Jetty Overlay Deployer allows you to overlay multiple WAR files so that you can customise, configure, and deploy a web application without unpacking, modifying and repacking the WAR file. This has the following benefits:

- You can keep the WAR file immutable, even signed, so that it is clear which version you have deployed.
- All modifications you make to customise/configure the web application are separate WARs, and thus are easily identifiable for review and migration to new versions.
- You can create a parameterised template overlay that contains common customisations and configuration that apply to many instances of the web application (for example, for multi-tenant deployment).
- Because the layered deployment clearly identifies the common and instance specific components, Jetty is able to share classloaders and static resource caches for the template, greatly reducing the memory footprint of multiple instances.

This tutorial describes how to configure Jetty to use the Overlay deployer, and how to deploy multiple instances of a web application, using the JTrac application in the example.

Overview

Customising, configuring and deploying a web application bundled as a WAR file frequently includes some or all of these steps:

- Editing the WEB-INF/web.xml file to set init parameters, add filters/servlets or to configure JNDI resources.
- Editing other application specific configuration files under WEB-INF/.
- Editing container specific configuration files under WEB-INF/ (for example, jetty-web.xml or jboss-web.xml).
- Adding/modifying static content such as images and CSS to create a style or themes for the web application.
- Adding Jars to the container classpath for Datasource and other resources.
- Modifying the container configuration to provide JNDI resources.

The result is that the customisations and configurations blend into both the container and the WAR file. If you upgrade either the container or the base WAR file to a new version, it can be a very difficult and error prone task to identify all the changes that you have made, and to reapply them to a new version.

Overlays

To solve the problems highlighted above, Jetty 7.4 introduces WAR overlays (a concept borrowed from the Maven WAR plugin). An overlay is basically just another WAR file, whose contents merge on top of the original WAR so that you can add or replace files. Jetty overlays also allow you to mix in fragments of web.xml, which means you can modify the configuration without replacing it.

JTrac Overlay Example

The JTrac issue tracking web application is a good example of a typical web application, as it uses the usual suspects of libs: spring, hibernate, dom4j, commons-*, wicket, etc. The files for this demon-

stration are available in overlays-demo.tar.gz. You can expand it on top of the jetty distribution; this tutorial expands it to /tmp and installs the components step-by-step:

```
$ cd /tmp  
$ wget http://webtide.intalio.com/wp-content/uploads/2011/05/overlays-demo.tar.gz  
$ tar xfvz overlays-demo.tar.gz  
$ export OVERLAYS=/tmp/overlays
```

Configuring Jetty for Overlays

Overlays support is included in jetty distributions from 7.4.1-SNAPSHOT onwards, so you can download a distribution from oss.sonatype.org or maven central and unpack into a directory. You need to edit the start.ini file so that it includes the overlay option and configuration file. The resulting file should look like:

```
OPTIONS=Server,jsp,jmx,resources,websocket,ext,overlay  
etc/jetty.xml  
etc/jetty-deploy.xml  
etc/jetty-overlay.xml
```

The smarts of this are in etc/jetty-deploy.xml, which installs the OverlayedAppProvider into the DeploymentManager. You can then start Jetty normally:

```
$ java -jar start.jar
```

Jetty is now listening on port 8080, but with no webapp deployed.



Important

You should conduct the rest of the tutorial in another window with the JETTY_HOME environment set to the jetty distribution directory.

Installing the WebApp

You can download and deploy the WAR file for this demo using the following commands, which essentially downloads and extracts the WAR file to the \$JETTY_HOME/overlays/webapps directory.

```
$ cd /tmp  
$ wget -O jtrac.zip http://sourceforge.net/projects/j-trac/files/jtrac/2.1.0/jtrac-2.1.0.zip/download  
$ jar xfv jtrac.zip jtrac/jtrac.war  
$ mv jtrac/jtrac.war $JETTY_HOME/overlays/webapps
```

When you have run these commands (or equivalent), you see in the Jetty server window a message saying that the OverlayedAppProvider has extracted and loaded the WAR file:

```
2011-05-06 10:31:54.678:INFO:OverlayedAppProvider:Extract jar:file:/tmp/jetty-distribution-7.4.1-SNAPSHOT/overlays/webapps/jtrac-2.1.0.war!/ to /tmp/jtrac-2.1.0_236811420856825222.extract
2011-05-06 10:31:55.235:INFO:OverlayedAppProvider:loaded jtrac-2.1.0@1304641914666
```

Unlike the normal webapps dir, loading a WAR file from the overlays/webapp dir does not deploy the web application. It simply makes it available to use as the basis for templates and overlays.

Installing a Template Overlay

A template overlay is a WAR structured directory/archive that contains just the files that you have added or modified to customize/configure the web application for all instances you plan to deploy.

You can install the demo template from the downloaded files with the command:

```
$ mv $OVERLAYS/jtracTemplate\=jtrac-2.1.0 $JETTY_HOME/overlays/templates/
```

In the Jetty server window, you should see the template loaded with a message like:

```
2011-05-06 11:00:08.716:INFO:OverlayedAppProvider:loaded
jtracTemplate=jtrac-2.1.0@1304643608715
```

The contents of the loaded template are as follows:

```
templates/jtracTemplate=jtrac-2.1.0
|__ WEB-INF
    |__ classes
    |__ jtrac-init.properties
    |__ log4j.properties
    |__ overlay.xml
    |__ template.xml
    |__ web-overlay.xml
```

name of the template directory (or WAR)

Uses the '=' character in jtracTemplate=jtrac-2.1.0 to separate the name of the template from the name of the WAR file in webapps that it applies to. If = is a problem, then you can instead use --.

WEB-INF/classes/jtrac-init.properties

Replaces the JTrac properties file with an empty file, as the properties it contains are configured elsewhere.

WEB-INF/log4j.properties

Configures the logging for all instances of the template.

WEB-INF/overlay.xml

A Jetty XML formatted IoC file that injects/configures the ContextHandler for each instance. In this case it just sets up the context path:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/configure.dtd">
```

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath"></Set>
</Configure>
```

WEB-INF/template.xml

a Jetty XML formatted IoC file that injects/configures the resource cache and classloader that all instances of the template share. It runs only once per load of the template:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure.dtd">
<Configure class="org.eclipse.jetty.overlays.TemplateContext">
  <Get name="resourceCache">
    <Set name="useFileMappedBuffer">true</Set>
    <Set name="maxCachedFileSize">10000000</Set>
    <Set name="maxCachedFiles">1000</Set>
    <Set name="maxCacheSize">64000000</Set>
  </Get>
</Configure>
```

WEB-INF/web-overlay.xml

a web.xml fragment that Jetty overlays on top of the web.xml from the base WAR file; it can set init parameters and add/modify filters and servlets. In this example it sets the application home and springs rootKey:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/
web-app_2_4.xsd"
  version="2.4">
  <context-param>
    <param-name>jtrac.home</param-name>
    <param-value>/tmp/jtrac-${overlay.instance.classifier}</param-value>
  </context-param>
  <context-param>
    <param-name>webAppRootKey</param-name>
    <param-value>jtrac-${overlay.instance.classifier}</param-value>
  </context-param>
  <filter>
</web-app>
```

Notice the parameterisation of values such as \${overlays.instance.classifier}, as this allows the configuration to be in the template, and not customised for each instance.

Without the Overlay Deployer, you would still need to have configured all of the above, but rather than being in a single clear structure the configuration elements would have been either in the server's common directory, the server's webdefaults.xml (aka server.xml), or baked into the WAR file of each application instance using copied/modified files from the original. The Overlay Deployer allows you to make all these changes in one structure; moreover it allows you to parameterise some of the configuration, which facilitates easy multi-tenant deployment.

Installing an Instance Overlay

Now that you have installed a template, you can install one or more instance overlays to deploy the actual web applications:

```
$ mv /tmp/overlays/instances/jtracTemplate\=blue $JETTY_HOME/overlays/instances/
$ mv /tmp/overlays/instances/jtracTemplate\=red $JETTY_HOME/overlays/instances/
$ mv /tmp/overlays/instances/jtracTemplate\=blue $JETTY_HOME/overlays/instances/
```

As each instance moves into place, you see the Jetty server window react and deploy that instance. Within each instance, there is the structure:

```
instances/jtracTemplate=red/
|__ WEB-INF
|   |__ overlay.xml
|__ favicon.ico
|__ resources
|   |__ jtrac.css
```

WEB-INF/overlay.xml

a Jetty XML format IoC file that injects/configures the context for the instance. In this case it sets up a virtual host for the instance:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure.dtd">
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
<Set name="virtualHosts">
  <Array type="String">
    <Item>127.0.0.2</Item>
    <Item>red.myVirtualDomain.com</Item>
  </Array>
</Set>
</Configure>
```

favicon.ico

Replaces the icon in the base WAR with one that has a theme for the instance, in this case red, blue, or green.

resources/jtrac.css

Replaces the style sheet from the base WAR with one that has a theme for the instance.

You can now view the deployed instances by pointing your browser at <http://127.0.0.1:8080>, <http://127.0.0.2:8080> and <http://127.0.0.3:8080>. The default username/password for JTrac is admin/admin.

Things to Know and Notice

- Each instance has themes with images and style sheets from the instance overlay.
- Each instance is running with its own application directory (that is, /tmp/jtrac-red), set in templates web-overlay.xml.
- A virtual host set in the instance overlay.xml distinguishes the instances.
- All instances share static content from the base WAR and template. Specifically there is a shared ResourceCache so only a single instance of each static content is loaded into memory.
- All instances share the classloader at the base WAR and template level, so that only a single instance of common classes is loaded into memory. You can configure classes with non shared statics to load in the instances classloader.

- Jetty hot deploys all overlays and tracks dependencies.
 - If an XML changes in an instance, Jetty redeploys it.
 - If an XML changes in a template, then Jetty redeploys all instances using it.
 - If a WAR file changes, then Jetty redeploys all templates and all instances dependant on it.
- You can easily deploy new versions. For example, when JTrac-2.2.0.war becomes available, you can just drop it into overlays/webapps and then rename jtracTemplate=jtrac-2.1.0 to jtracTemplate=jtrac-2.2.0
- There is a fuller version of this demo in overlays-demo-jndi.tar.gz, that uses JNDI (needs options=jndi, annotations and jetty-plus.xml in start.ini) and shows how you can add extra JARs in the overlays.

Chapter 5. Configuring Contexts

Table of Contents

Setting a Context Path	47
Configuring Virtual Hosts	48
Temporary Directories	51
Serving a WebApp from a Particular Port/Connector	54
Creating Custom Error Pages	55
Setting Max Form Size	58

This chapter discusses various options for configuring Jetty contexts.

Setting a Context Path

The context path is the prefix of a URL path that is used to select the context(s) to which an incoming request is passed. Typically a URL in a Java servlet server is of the format `http://hostname.com/contextPath/servletPath/pathInfo`, where each of the path elements can be zero or more / separated elements. If there is no context path, the context is referred to as the *root* context. The root context must be configured as "/" but is reported as the empty string by the servlet API `getContextPath()` method.

How you set the context path depends on how you deploy the web application (or `ContextHandler`):

Using Embedded Deployment

If you run Jetty from code as an embedded server (see [Embedding](#)), setting the context path is a matter of calling the `setContextPath` method on the `ContextHandler` instance (or `WebAppContext` instance).

By naming convention

If a web application is deployed using the `WebAppProvider` of the `DeploymentManager` without an XML IoC file, then the name of the WAR file is used to set the context path:

- If the WAR file is named `myapp.war`, then the context will be deployed with a context path of `/myapp`
- If the WAR file is named `ROOT.WAR` (or any case insensitive variation), then the context will be deployed with a context path of `/`
- If the WAR file is named `ROOT-foobar.war` (or any case insensitive variation), then the context will be deployed with a context path of `/` and a virtual host of "foobar"

By Deployer configuration

If a web application is deployed using the `WebAppProvider` of the `DeploymentManager` with an XML IoC file to configure the context, then the `setContextPath` method can be called within that file. For example:

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath">/test</Set>
  ...
</Configure>
```

Embedding a WEB-INF/jetty-web.xml File

You can also set the context path for webapps by embedding a WEB-INF/jetty-web.xml file in the WAR, which uses the same XML IoC format as the deployer example above. However this is not the preferred method as it requires the web application to be modified.

Configuring Virtual Hosts

A virtual host is an alternative name, registered in DNS, for an IP address such that multiple domain names will resolve to the same IP of a shared server instance. If the content to be served to the aliases names is [different](#), then a virtual host needs to be configured for each deployed [context](#) to indicate which names a context will respond to.

Virtual hosts are set on a [context](#) by calling the [setVirtualHosts](#) or [addVirtualHost](#) method which can be done either

- Using a [context XML](#) file in the webapps directory (see the example in [test.xml](#) in the jetty distribution).
- Using a WEB-INF/jetty-web.xml file (deprecated).
- Creating a [custom deployer](#) with a binding to configure virtual hosts for all contexts found in the same webapps directory.
- Calling the [API](#) directly on an [embedded](#) usage.

Virtual Host Names

Jetty supports the following styles of virtual host name:

www.hostname.com

A fully qualified host name. It is important to list all variants as a site may receive traffic from both www.hostname.com and just hostname.com

***.hostname.com**

A wildcard qualified host which will match only one level of arbitrary names. *.foo.com will match www.foo.com and m.foo.com, but not www.other.foo.com

10.0.0.2

An IP address may be given as a virtual host name to indicate that a context should handle requests received on that server port that do not have a host name specified (HTTP/0.9 or HTTP/1.0)

@ConnectorName

A connector name, which is not strictly a virtual host, but instead will only match requests that are received on connectors that have a matching name set with [Connector.setName\(String\)](#).

www.#integral.com

Non ascii and [IDN](#) domain names can be set as virtual hosts using [Puny Code](#) equivalents that may be obtained from a [Punycode/IDN converters](#). For example if the non ascii domain name www.#integral.com is given to a browser, then it will make a request that uses the domain

name `www.xn--integral-7g7d.com`, which is the name that should be added as the virtual host name.

Example Virtual Host Configuration

Virtual hosts can be used with any context that is a subclass of [ContextHandler](#). Lets look at an example where we configure a web application - represented by the [WebAppContext](#) class - with virtual hosts. You supply a list of IP addresses and names at which the web application is reachable. Suppose you have a machine with these IP addresses and these DNS resolvable names:

- `333.444.555.666`
- `127.0.0.1`
- `www.blah.com`
- `www.blah.net`
- `www.blah.org`

Suppose you have a webapp called `blah.war`, that you want all of the above names and addresses to be served at path `"/blah"`. Here's how you would configure the virtual hosts with a [context XML](#) file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
<Set name="contextPath">/blah</Set>
<Set name="war"><Property name="jetty.webapps"/>/webapps/blah.war</Set>
<Set name="virtualHosts">
<Array type="java.lang.String">
<Item>333.444.555.666</Item>
<Item>127.0.0.1</Item>
<Item>www.blah.com</Item>
<Item>www.blah.net</Item>
<Item>www.blah.org</Item>
</Array>
</Set>
</Configure>
```

Using Different Sets of Virtual Hosts to Select Different Contexts

You can configure different contexts to respond on different virtual hosts by supplying a specific list of virtual hosts for each one.

For example, suppose your imaginary machine has these DNS names:

- `www.blah.com`
- `www.blah.net`
- `www.blah.org`
- `www.other.com`
- `www.other.net`
- `www.other.org`

Suppose also you have 2 webapps, one called `blah.war` that you would like served from the `*.blah.*` names, and one called `other.war` that you want served only from the `*.other.*` names.

Using the method of preparing [context XML](#) files, one for each webapp yields the following:

For blah webapp:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
<Set name="contextPath"/>/blah</Set>
<Set name="war"><Property name="jetty.webapps"/>/webapps/blah.war</Set>
<Set name="virtualHosts">
<Array type="java.lang.String">
<Item>www.blah.com</Item>
<Item>www.blah.net</Item>
<Item>www.blah.org</Item>
</Array>
</Set>
</Configure>
```

These urls now resolve to the blah context (ie blah.war):

- <http://www.blah.com/blah>
- <http://www.blah.net/blah>
- <http://www.blah.org/blah>

For other webapp:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
<Set name="contextPath"/>/other</Set>
<Set name="war"><Property name="jetty.webapps"/>/webapps/other.war</Set>
<Set name="virtualHosts">
<Array type="java.lang.String">
<Item>www.other.com</Item>
<Item>www.other.net</Item>
<Item>www.other.org</Item>
</Array>
</Set>
</Configure>
```

These urls now resolve to the other context (ie other.war):

- <http://www.other.com/other>
- <http://www.other.net/other>
- <http://www.other.org/other>

Using Different Sets of Virtual Hosts to Select Different Contexts at the Same Context Path

In the previous section we setup 2 different contexts to be served from different virtual hosts at *different* context paths. However, there is no requirement that the context paths must be distinct: you may use the same context path for multiple contexts, and use virtual hosts to determine which one is served for a given request.

Consider an example where we have the same set of DNS names as before, and the same webapps `blah.war` and `other.war`. We still want `blah.war` to be served in response to hostnames of

`*.blah.*`, and we still want `other.war` to be served in response to `*.other.*` names. However, we would like *all* of our clients to use the " / " context path, no matter which context is being targeted.

In other words, we want all of the following urls to map to `blah.war`:

- `http://www.blah.com/`
- `http://www.blah.net/`
- `http://www.blah.org/`

Similarly, we want the following urls to map to `other.war`:

- `http://www.other.com/`
- `http://www.other.net/`
- `http://www.other.org/`

To achieve this, we simply use the same context path of "/" for each of our webapps, whilst still applying our different set of virtual host names.

For foo webapp:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath">/</Set>
  <Set name="war"><Property name="jetty.webapps"/>/webapps/foo.war</Set>
  <Set name="virtualHosts">
    <Array type="java.lang.String">
      <Item>www.blah.com</Item>
      <Item>www.blah.net</Item>
      <Item>www.blah.org</Item>
    </Array>
  </Set>
</Configure>
```

For bar webapp:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath">/</Set>
  <Set name="war"><Property name="jetty.webapps"/>/webapps/bar.war</Set>
  <Set name="virtualHosts">
    <Array type="java.lang.String">
      <Item>www.other.com</Item>
      <Item>www.other.net</Item>
      <Item>www.other.org</Item>
    </Array>
  </Set>
</Configure>
```

Temporary Directories

Jetty itself has no temporary directories, but you can assign a directory for each web application, into which the WAR is unpacked, JSPs compiled on-the-fly, etc. If you do not assign a specific temporary directory, Jetty will create one as needed when your web application starts. Whether you set the loca-

tion of the temporary directory - or you let Jetty create one for you - you also have a choice to either keep or delete the temporary directory when the web application stops.

The default temp directory

By default, Jetty will create a temporary directory for each web application. The name of the directory will be of the form:

```
"jetty-"+host+"-"+port+"-"+resourceBase+"_"+context+"_"+virtualhost+"_"+randomdigits  
+" .dir"
```

For example:

```
jetty-0.0.0.0-8080-test.war-_test-any-8900275691885214790.dir
```

Where 0.0.0.0 is the host address, 8080 is the port, test.war is the resourceBase, test is the context path (with / converted to _), any is the virtual host, and randomdigits are a string of digits guaranteed to be unique.

Once the temp directory is created, it is retrievable as the value (as a File) of the context attribute javax.servlet.context.tempdir.

The location of the temp directory

By default, jetty will create this directory inside the directory named by the java.io.tmpdir System property. You can instruct Jetty to use a different parent directory by setting the context attribute org.eclipse.jetty.webapp.basetempdir to the name of the desired parent directory. The directory named by this attribute *must* exist and be *writeable*.

As usual with Jetty, you can either set this attribute in a context xml file, or you can do it in code. Here's an example of setting it in an xml file:

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">  
  
<Set name="contextPath">/test</Set>  
<Set name="war">foo.war</Set>  
  
<Call name="setAttribute">  
<Arg>org.eclipse.jetty.webapp.basetempdir</Arg>  
<Arg>/home/my/foo</Arg>  
</Call>  
</Configure>
```

The equivalent in code is:

```
WebAppContext context = new WebAppContext();  
context.setContextPath("/test");  
context.setWar("foo.war");  
context.setAttribute("org.eclipse.jetty.webapp.basetempdir", "/tmp/foo");
```

Setting a specific temp directory

To use a particular directory as the temporary directory you can do either:

call WebAppContext.setTempDirectory(String dir)

As usual with Jetty, you can do this with an xml file or directly in code. Here's an example of setting the temp directory in xml:

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">  
  
<Set name="contextPath">/test</Set>  
<Set name="war">foo.war</Set>  
  
<Set name="tempDirectory">/some/dir/foo</Set>
```

```
</Configure>
```

Here's an example of doing it with java code:

```
WebAppContext context = new WebAppContext();
context.setContextPath("/test");
context.setWar("foo.war");
context.setTempDirectory(new File("/some/dir/foo"));
```

set the `javax.servlet.context.tempdir` context attribute

You should set this context attribute with the name of directory you want to use as the temp directory. Again, you can do this in xml:

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">

    <Set name="contextPath"/>/test</Set>
    <Set name="war">foo.war</Set>

    <Call name="setAttribute">
        <Arg>javax.servlet.context.tempdir</Arg>
        <Arg>/some/dir/foo</Arg>
    </Call>

</Configure>
```

Or in java:

```
WebAppContext context = new WebAppContext();
context.setContextPath("/test");
context.setWar("foo.war");
context.setAttribute("javax.servlet.context.tempdir", "/some/dir/foo");
```

Once a temporary directory has been created by either of these methods, a File instance for it is set as the value of the `javax.servlet.context.tempdir` attribute of the web application.



Note

Be wary of setting an explicit temp directory if you are likely to change the jars in WEB-INF/lib between redeployments. There is a JVM bug concerning caching of jar contents: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4774421

The "work" directory

Mostly for backward compatibility, from jetty-9.1.1 onwards, it will be possible to create a directory named "work" in the \${jetty.base} directory. If such a directory is found, it is assumed you want to use it as the parent directory for all of the temporary directories of the webapps in that \${jetty.base}. Moreover, as has historically been the case, these temp directories inside the work directory are not cleaned up when jetty exists (or more correctly speaking, the temp dir corresponding to a context is not cleaned up when that context stops).

When a work directory is used, the algorithm for generating the name of the context-specific temp dirs omits the random digit string. This ensures the name of the dir remains consistent across context restarts.

Persisting the temp directory

Sometimes you may find it useful to keep the contents of the temporary directory between restarts of the web application. By default, Jetty will *not* persist the temp directory. To cause Jetty to keep it, use [WebAppContext.setPersistTempDirectory\(true\)](#).

Be aware that if you call `setPersistTempDirectory(true)`, but let Jetty create a new temp directory each time (ie you do NOT set an explicit temp directory), then you will accumulate temp directories in your chosen temp directory location.

Serving a WebApp from a Particular Port/Connector

Sometimes it is required to serve different web applications from different ports/connectors. The simplest way to do this is to create multiple Server instances, however if contexts need to share resources (eg data sources, authentication), or if the mapping of ports to web applications is not cleanly divided, then the named connector mechanism can be used.

Creating Multiple Server Instances

How to create multiple server instances is simply done when writing embedded jetty code by creating multiples instances of the Server class and configuring them as needed. This is also easy to achieve if you are configuring your servers in XML. The id field in the Configure element of jetty.xml files is used to identify the instance that the configuration applies to, so to run two instances of the Server, you can copy the jetty.xml, jetty-http.xml and other jetty configuration files used and change the "Server" id to a new name. This can be done in the same style and layout as the existing jetty.xml files or the multiple XML files may be combined to a single file.

When creating new configurations for alternative server:

- Change all id="Server" to the new server name:<Configure id="OtherServer" class="org.eclipse.jetty.server.Server">
- For all connectors for the new server change the refid in the server argument: <Arg name="server"><Ref refid="OtherServer" /></Arg>
- Make sure that any references to properties like jetty.port are either renamed or replaced with absolute values
- Make sure that any deployers AppProviders refer to a different "webapps" directory so that a different set of applications are deployed.

Example Other Server XML

The following example creates another server instance and configures it with a connector and deployer:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure id="OtherServer" class="org.eclipse.jetty.server.Server">
    <Set name="handler">
        <New id="Handlers" class="org.eclipse.jetty.server.handler.HandlerCollection">
            <Set name="handlers">
                <Array type="org.eclipse.jetty.server.Handler">
                    <Item>
                        <New id="OtherContexts"
                            class="org.eclipse.jetty.server.handler.ContextHandlerCollection"/>
                    </Item>
                    <Item>
                        <New class="org.eclipse.jetty.server.handler.DefaultHandler"/>
                    </Item>
                </Array>
            </Set>
        </New>
    </Set>
</Configure>

<Call name="addConnector">
    <Arg>
        <New class="org.eclipse.jetty.server.ServerConnector">
            <Arg name="server"><Ref refid="OtherServer" /></Arg>
            <Set name="port">8888</Set>
        </New>
    </Arg>
</Call>
```

```

        </New>
    </Arg>
</Call>

<Call name="addBean">
    <Arg>
        <New id="DeploymentManager" class="org.eclipse.jetty.deploy.DeploymentManager">
            <Set name="contexts">
                <Ref refid="OtherContexts" />
            </Set>

            <Call id="webappprovider" name="addAppProvider">
                <Arg>
                    <New class="org.eclipse.jetty.deploy.providers.WebAppProvider">
                        <Set name="monitoredDirName"><Property name="jetty.home" default="." />
other-webapps</Set>
                        <Set name="defaultsDescriptor"><Property name="jetty.home" default="." />
etc/webdefault.xml</Set>
                        <Set name="configurationManager">
                            <New class="org.eclipse.jetty.deploy.PropertiesConfigurationManager"/>
                        </Set>
                    </New>
                </Arg>
            </Call>
        </New>
    </Arg>
</Call>
</Configure>

```

To run the other server, simply add the extra configuration file(s) to the command line:

```
java -jar start.jar jetty-otherserver.xml
```

Named Connectors

It is also possible to use an extension to the virtual host mechanism with named to connectors to make some web applications only accessible by specific connectors. If a connector has a name "MyConnector" set using the setName method, then this can be referenced with the special virtual host name "@MyConnector".

Creating Custom Error Pages

The following sections describe several ways to create custom error pages in Jetty.

Defining error pages in web.xml

You can use the standard webapp configuration file located in webapp/WEB-INF/web.xml to map errors to specific URLs with the error-page element. This element creates a mapping between the error-code or exception-type to the location of a resource in the web application.

- error-code - an integer value
- exception-type - a fully qualified class name of a Java Exception type
- location - location of the resource in the webapp relative to the root of the web application. Value should start with "/".

Error code example:

```

<error-page>
    <error-code>404</error-code>
    <location>/ jspnoop/ERROR/404</location>
</error-page>

```

Exception example:

```
<error-page>
<exception-type>java.io.IOException</exception-type>
<location>/jspotsnoop/IOException</location>
</error-page>
```

The error page mappings created with the `error-page` element will redirect to a normal URL within the web application and will be handled as a normal request to that location and thus may be static content, a JSP or a filter and/or servlet. When handling a request generated by an error redirection, the following request attributes are set and are available to generate dynamic content:

javax.servlet.error.exception

The exception instance that caused the error (or null)

javax.servlet.error.exception_type

The class name of the exception instance that caused the error (or null)

javax.servlet.error.message

The error message.

javax.servlet.error.request_uri

The URI of the errored request.

javax.servlet.error.servlet_name

The Servlet name of the servlet that the errored request was dispatched to/

javax.servlet.error.status_code

The status code of the error (e.g. 404, 500 etc.)

Configuring error pages context files

You can use context IoC XML files to configure the default error page mappings with more flexibility than is available with `web.xml`, specifically with the support of error code ranges. Context files are normally located in `${jetty.home} /webapps/` (see DeployerManager for more details) and an example of more flexible error mapping is below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN" "http://
jetty.eclipse.org/configure.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
<Set name="contextPath">/test</Set>
<Set name="war">
<SystemProperty name="jetty.home" default=". "/>/webapps/test
</Set>

<!-- by Code -->
<Get name="errorHandler">
<Call name="addErrorPage">
<Arg type="int">404</Arg>
<Arg type="String">/jspotsnoop/ERROR/404</Arg>
</Call>
</Get>

<!-- by Exception -->
<Get name="errorHandler">
<Call name="addErrorPage">
<Arg>
<Call class="java.lang.Class" name="forName">
<Arg type="String">java.io.IOException</Arg>
</Call>
</Arg>
<Arg type="String">/jspotsnoop/IOException</Arg>
</Get>
```

```

</Call>
</Get>

<!-- by Code Range -->
<Get name="errorHandler">
  <Call name="addErrorPage">
    <Arg type="int">500</Arg>
    <Arg type="int">599</Arg>
    <Arg type="String">/dump/errorCodeRangeMapping</Arg>
  </Call>
</Get>
</Configure>

```

Custom ErrorHandler class

If no error page mapping is defined, or if the error page resource itself has an error, then the error page will be generated by an instance of ErrorHandler configured either the Context or the Server. An ErrorHandler may extend the ErrorHandler class and may totally replace the handle method to generate an error page, or it can implement some or all of the following methods to partially modify the error pages:

```

void handle(String target, Request baseRequest, HttpServletRequest request,
HttpServletRequest response) throws IOException
void handleErrorPage(HttpServletRequest request, Writer writer, int code, String
message) throws IOException
void writeErrorPage(HttpServletRequest request, Writer writer, int code, String
message, boolean showStacks) throws IOException
void writeErrorPageHead(HttpServletRequest request, Writer writer, int code, String
message) throws IOException
void writeErrorPageBody(HttpServletRequest request, Writer writer, int code, String
message, boolean showStacks) throws IOException
void writeErrorMessage(HttpServletRequest request, Writer writer, int code, String
message, String uri) throws IOException
void writeErrorPageStacks(HttpServletRequest request, Writer writer) throws IOException

```

An ErrorHandler instance may be set on a Context by calling the ContextHandler.setErrorHandler method. This can be done by embedded code or via context IoC XML. For example:

```

<Configure class="org.eclipse.jetty.server.handler.ContextHandler">
  ...
  <Set name="errorHandler">
    <New class="com.acme.handler.MyErrorHandler"/>
  </Set>
  ...
</Configure>

```

An ErrorHandler instance may be set on the entire server by setting it as a dependent bean on the Server instance. This can be done by calling Server.addBean(Object) via embedded code or in jetty.xml IoC XML like:

```

<Configure id="Server" class="org.eclipse.jetty.server.Server">
  ...
  <Call name="addBean">
    <Arg>
      <New class="com.acme.handler.MyErrorHandler"/>
    </Arg>
  </Call>
  ...
</Configure>

```

Server level 404 error

You might get a 'page not found' when a request is made to the server for a resource that is outside of any registered contexts. As an example, you have a domain name pointing to your public server IP, yet no context is registered with jetty to serve pages for that domain. As a consequence, the server, by default, gives a listing of all contexts running on the server.

One of the quickest ways to avoid this behavior is to create a catch all context. Create a "root" web app mapped to the "/" URI. Have the index.html redirect to whatever place with a header directive.

Setting Max Form Size

Jetty limits the amount of data that can post back from a browser or other client to the server. This helps protect the server against denial of service attacks by malicious clients sending huge amounts of data. The default maximum size Jetty permits is 200000 bytes. You can change this default for a particular webapp, for all webapps on a particular Server instance, or all webapps within the same JVM.

For a Single Webapp

The method to invoke is: `ContextHandler.setMaxFormContentSize(int maxSize);`

You can do this either in a context XML deployment descriptor external to the webapp, or in a `jetty-web.xml` file in the webapp's WEB-INF directory.

In either case the syntax of the XML file is the same:

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <!-- Max Form Size -->
  <!-->
  <Set name="maxFormContentSize">200000</Set>
</Configure>
```

For All Apps on a Server

Set an attribute on the Server instance for which you want to modify the maximum form content size:

```
<Configure class="org.eclipse.jetty.server.Server">
  <Call name="setAttribute">
    <Arg>org.eclipse.jetty.server.Request.maxFormContentSize</Arg>
    <Arg>200000</Arg>
  </Call>
</Configure>
```

For All Apps in the JVM

Use the system property `org.eclipse.jetty.server.Request.maxFormContentSize`. This can be set on the commandline or in the start.ini file.

Chapter 6. Configuring Jetty Connectors

Table of Contents

Connector Configuration Overview	59
Configuring SSL	65
Setting Port 80 Access for a Non-Root User	72

This chapter discusses various options for configuring Jetty connectors.

Connector Configuration Overview

Connectors are the mechanism through which Jetty accepts network connections for various protocols. Configuring a connector is a combination of configuring the following:

- Network parameters on the connector itself (for example: the listening port).
- Services the connector uses (for example: executors, schedulers).
- Connection factories that instantiate and configure the protocol for an accepted connection.

Jetty primarily uses a single connector type called [ServerConnector](#).



Note

Prior to Jetty 9, the type of the connector specified both the protocol and the implementation used (for example, selector-based non blocking I/O vs blocking I/O, or SSL connector vs non-SSL connector). Jetty 9 has only a selector-based non blocking I/O connector, and a collection of [ConnectionFactories](#) now configure the protocol on the connector.

The standard Jetty distribution comes with the following Jetty XML files that create and configure connectors; you should examine them as you read this section:

[jetty-http.xml](#)

Instantiates a [ServerConnector](#) that accepts HTTP connections (that may be upgraded to WebSocket connections).

[jetty-https.xml](#)

Instantiates a [ServerConnector](#) that accepts SSL/TLS connections. The ConnectionFactory configured after the SSL one is a HTTP ConnectionFactories, and therefore the SSL/TLS connections will carry the HTTP protocol.

[example-jetty-spdy.xml](#)

Instantiates a [ServerConnector](#) that accepts SSL connections that carry either HTTP or SPDY traffic. Initially the SSL connection is chained to a Next Protocol Negotiation (NPN) connection, which eventually replaces itself with a connection for a protocol it negotiates with the client; this protocol may be a version of SPDY or HTTP. If the client does not support NPN, HTTP is assumed.

Typically you need to configure very little on connectors other than set the listening port (see [Network Settings](#)), and perhaps enable X-Forwarded-For customization (see [HTTP Configuration](#)). Most other settings are for expert configuration only.

Constructing a ServerConnector

The services a [ServerConnector](#) instance uses are set by constructor injection and once instantiated cannot be changed. Most of the services may be defaulted with null or 0 values so that a reasonable default is used, thus for most purposes only the Server and the connection factories need to be passed to the connector constructor. In Jetty XML (that is, in [jetty-http.xml](#)), you can do this with:

```
<New class="org.eclipse.jetty.server.ServerConnector">
  <Arg name="server"><Ref refid="Server" /></Arg>
  <Arg name="factories">
    <Array type="org.eclipse.jetty.server.ConnectionFactory">
      <!-- insert one or more factories here -->
    </Array>
  </Arg>
  <!-- set connector fields here -->
</New>
```

You can see the other arguments that can be passed when constructing a ServerConnector in the [Javadoc](#). Typically the defaults are sufficient for almost all deployments.

Network Settings.

You configure connector network settings by calling setters on the connector before it is started. For example, you can set the port with the Jetty XML:

```
<New class="org.eclipse.jetty.server.ServerConnector">
  <Arg name="server"><Ref refid="Server" /></Arg>
  <Arg name="factories"><!-- insert one or more factories here --></Arg>

  <Set name="port">8080</Set>
</New>
```

Values in Jetty XML can also be parameterized so that they may be passed from property files or set on the command line. Thus typically the port is set within Jetty XML, but uses the `Property` element to be customizable:

```
<New class="org.eclipse.jetty.server.ServerConnector">
  <Arg name="server"><Ref refid="Server" /></Arg>
  <Arg name="factories"><!-- insert one or more factories here --></Arg>

  <Set name="port"><Property name="jetty.port" default="8080"/></Set>
</New>
```

The network settings that you can set on the [ServerConnector](#) include:

Table 6.1. Connector Configuration

Field	Description
host	The network interface this connector binds to as an IP address or a hostname. If null or 0.0.0.0, bind to all interfaces.
port	The configured port for the connector or 0 a random available port may be used (selected port available via <code>getLocalPort()</code>).
idleTimeout	The time in milliseconds that the connection can be idle before it is closed.
defaultProtocol	The name of the default protocol used to select a <code>ConnectionFactory</code> instance. This defaults to the first <code>ConnectionFactory</code> added to the connector.
stopTimeout	The time in milliseconds to wait before gently stopping a connector.
acceptQueueSize	The size of the pending connection backlog. The exact interpretation is JVM and operating system specific and you can ignore it. Higher values allow more

Field	Description
	connections to wait pending an acceptor thread. Because the exact interpretation is deployment dependent, it is best to keep this value as the default unless there is a specific connection issue for a specific OS that you need to address.
reuseAddress	Allow the server socket to be rebound even if in TIME_WAIT . For servers it is typically OK to leave this as the default true.
soLingerTime	A value >=0 set the socket SO_LINGER value in milliseconds. Jetty attempts to gently close all TCP/IP connections with proper half close semantics, so a linger timeout should not be required and thus the default is -1.

HTTP Configuration

The [HttpConfiguration](#) class holds the configuration for [HTTPChannel](#)'s, which you can create 1:1 with each HTTP connection or 1:n on a multiplexed SPDY connection. Thus a [HTTPConfiguration](#) object is injected into both the HTTP and SPDY connection factories. To avoid duplicate configuration, the standard Jetty distribution creates the common [HttpConfiguration](#) instance in [jetty.xml](#), which is a Ref element then used in [jetty-http.xml](#), [jetty-https.xml](#) and [example-jetty-spdy.xml](#).

A typical configuration of [HttpConfiguration](#) is:

```
<New id="httpConfig" class="org.eclipse.jetty.server.HttpConfiguration">
  <Set name="secureScheme">https</Set>
  <Set name="securePort"><Property name="jetty.tls.port" default="8443" /></Set>
  <Set name="outputBufferSize">32768</Set>
  <Set name="requestHeaderSize">8192</Set>
  <Set name="responseHeaderSize">8192</Set>

  <Call name="addCustomizer">
    <Arg><New class="org.eclipse.jetty.server.ForwardedRequestCustomizer"/></Arg>
  </Call>
</New>
```

This example adds a [ForwardedRequestCustomizer](#) to process the [X-Forward-For](#) and related proxy headers. [jetty-https.xml](#) can, by reference, use the instance created with an ID "httpConfig":

```
<Call name="addConnector">
  <Arg>
    <New class="org.eclipse.jetty.server.ServerConnector">
      <Arg name="server"><Ref refid="Server" /></Arg>
      <Arg name="factories">
        <Array type="org.eclipse.jetty.server.ConnectionFactory">
          <Item>
            <New class="org.eclipse.jetty.server.HttpConnectionFactory">
              <Arg name="config"><Ref refid="httpConfig" /></Arg>
            </New>
          </Item>
        </Array>
      </Arg>
    <!-- ... -->
  </New>
</Arg>
</Call>
```

For SSL based connectors (in [jetty-https.xml](#) and [jetty-spdy.xml](#)), the common "httpConfig" instance is used as the basis to create an SSL specific configuration with ID "tlsHttpConfig":

```
<New id="tlsHttpConfig" class="org.eclipse.jetty.server.HttpConfiguration">
  <Arg><Ref refid="httpConfig" /></Arg>
  <Call name="addCustomizer">
    <Arg><New class="org.eclipse.jetty.server.SecureRequestCustomizer"/></Arg>
  </Call>
</New>
```

This adds a `SecureRequestCustomizer` which adds SSL Session IDs and certificate information as request attributes.

SSL Context Configuration

The SSL/TLS connectors for HTTPS and SPDY require a certificate to establish a secure connection. Jetty holds certificates in standard JVM keystores and are configured as keystore and truststores on a `SslContextFactory` instance that is injected into an `SslConnectionFactory` instance. An example using the keystore distributed with Jetty (containing a self signed test certificate) is in `jetty-https.xml` and `example-jetty-spdy.xml`. Read more about SSL keystores in [Configuring SSL](#).

Configuring Connection Factories

It is the `ConnectionFactory` instances injected into a `ServerConnector` that create the protocol handling `Connection` instances for the network endpoints the connector accepts. Thus the different instances of connectors in a Jetty setup vary mostly in the configuration of the factories for the protocols they support. Other than selecting which factories to use, there is typically very little factory configuration required other than injecting the `HTTPConfiguration` or `SslContextFactory` instances.

The simplest example in the Jetty distribution is [jetty-http.xml](#):

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<!-- ===== -->
<!-- Configure the Jetty Server instance with an ID "Server"      -->
<!-- by adding a HTTP connector.                                     -->
<!-- This configuration must be used in conjunction with jetty.xml -->
<!-- ===== -->
<Configure id="Server" class="org.eclipse.jetty.server.Server">

<!-- ===== -->
<!-- Add a HTTP Connector.                                         -->
<!-- Configure an o.e.j.server.ServerConnector with a single       -->
<!-- HttpConnectionFactory instance using the common httpConfig   -->
<!-- instance defined in jetty.xml                                -->
<!-- -->
<!-- Consult the javadoc of o.e.j.server.ServerConnector and       -->
<!-- o.e.j.server.HttpConnectionFactory for all configuration    -->
<!-- that may be set here.                                         -->
<!-- ===== -->
<Call name="addConnector">
  <Arg>
    <New class="org.eclipse.jetty.server.ServerConnector">
      <Arg name="server"><Ref refid="Server" /></Arg>
      <Arg name="factories">
        <Array type="org.eclipse.jetty.server.ConnectionFactory">
          <Item>
            <New class="org.eclipse.jetty.server.HttpConnectionFactory">
              <Arg name="config"><Ref refid="httpConfig" /></Arg>
            </New>
          </Item>
        </Array>
      </Arg>
      <Set name="host"><Property name="jetty.host" /></Set>
      <Set name="port"><Property name="jetty.port" default="80" /></Set>
      <Set name="idleTimeout"><Property name="http.timeout" default="30000"/></Set>
      <Set name="soLingerTime"><Property name="http.soLingerTime" default="-1"/></Set>
    </New>
  </Arg>
</Call>
</Configure>
```

Here the connector has only a single `ConnectionFactory`, and when a new connection is accepted, it is the `HttpConnectionFactory` that creates an `HttpConnection`.

A more complex example involving multiple connection factories is `jetty-spdy.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<Configure id="Server" class="org.eclipse.jetty.server.Server">

    <New id="sslContextFactory" class="org.eclipse.jetty.util.ssl.SslContextFactory">
        <Set name="keyStorePath">src/main/resources/keystore.jks</Set>
        <Set name="keyStorePassword">storepwd</Set>
        <Set name="trustStorePath">src/main/resources/truststore.jks</Set>
        <Set name="trustStorePassword">storepwd</Set>
        <Set name="protocol">TLSv1</Set>
    </New>

    <New id="tlsHttpConfig" class="org.eclipse.jetty.server.HttpConfiguration">
        <Arg>
            <New id="httpConfig" class="org.eclipse.jetty.server.HttpConfiguration">
                <Set name="secureScheme">https</Set>
                <Set name="securePort">
                    <Property name="jetty.tls.port" default="8443"/>
                </Set>
                <Set name="outputBufferSize">32768</Set>
                <Set name="requestHeaderSize">8192</Set>
                <Set name="responseHeaderSize">8192</Set>

                <!-- Uncomment to enable handling of X-Forwarded- style headers
                <Call name="addCustomizer">
                    <Arg><New
                        class="org.eclipse.jetty.server.ForwardedRequestCustomizer" /></Arg>
                    </Call>
                    -->
                </New>
            </Arg>
            <Call name="addCustomizer">
                <Arg>
                    <New class="org.eclipse.jetty.server.SecureRequestCustomizer" />
                </Arg>
            </Call>
        </New>
    </Arg>
    <Call name="addCustomizer">
        <Arg>
            <New id="pushStrategy"
                class="org.eclipse.jetty.spdy.server.http.ReferrerPushStrategy">
                <!-- Uncomment to blacklist browsers for this push strategy. If one of the
                blacklisted Strings occurs in the
                    user-agent header sent by the client, push will be disabled for this
                browser. This is case insensitive" -->
                <!--
                <Set name="UserAgentBlacklist">
                    <Array type="String">
                        <Item>.*\ozilla/14.*</Item>
                        <Item>.*\ozilla/15.*</Item>
                        <Item>.*\ozilla/16.*</Item>
                    </Array>
                </Set>
                -->

                <!-- Uncomment to override default file extensions to push -->
                <!--
                <Set name="PushRegexp">
                    <Array type="String">
                        <Item>.*\.css</Item>
                        <Item>.*\.js</Item>
                        <Item>.*\.png</Item>
                        <Item>.*\.jpg</Item>
                        <Item>.*\.gif</Item>
                    </Array>
                </Set>
                -->
                <Set name="referrerPushPeriod">5000</Set>
                <Set name="maxAssociatedResources">32</Set>
            </New>
        <Call id="sslConnector" name="addConnector">
            <Arg>
```

```

<New class="org.eclipse.jetty.server.ServerConnector">
    <Arg name="server"><Ref refid="Server"/></Arg>
    <Arg name="factories">
        <Array type="org.eclipse.jetty.server.ConnectionFactory">

            <!-- SSL Connection factory with NPN as next protocol -->
            <Item>
                <New class="org.eclipse.jetty.server.SslConnectionFactory">
                    <Arg name="next">npn</Arg>
                    <Arg name="sslContextFactory">
                        <Ref refid="sslContextFactory"/>
                    </Arg>
                </New>
            </Item>

            <!-- NPN Connection factory with HTTP as default protocol -->
            <Item>
                <New
                    class="org.eclipse.jetty.spdy.server.NPNServerConnectionFactory">
                    <Arg name="protocols">
                        <Array type="String">
                            <Item>spdy/3</Item>
                            <Item>spdy/2</Item>
                            <Item>http/1.1</Item>
                        </Array>
                    </Arg>
                    <Set name="defaultProtocol">http/1.1</Set>
                </New>
            </Item>

            <!-- SPDY/3 Connection factory -->
            <Item>
                <New
                    class="org.eclipse.jetty.spdy.server.http.HTTPSPDYServerConnectionFactory">
                    <Arg name="version" type="int">3</Arg>
                    <Arg name="config">
                        <Ref refid="tlsHttpConfig"/>
                    </Arg>
                    <Arg name="pushStrategy">
                        <Ref refid="pushStrategy"/>
                    </Arg>
                </New>
            </Item>

            <!-- SPDY/2 Connection factory -->
            <Item>
                <New
                    class="org.eclipse.jetty.spdy.server.http.HTTPSPDYServerConnectionFactory">
                    <Arg name="version" type="int">2</Arg>
                    <Arg name="config">
                        <Ref refid="tlsHttpConfig"/>
                    </Arg>
                </New>
            </Item>

            <!-- HTTP Connection factory -->
            <Item>
                <New class="org.eclipse.jetty.server.HttpConnectionFactory">
                    <Arg name="config">
                        <Ref refid="tlsHttpConfig"/>
                    </Arg>
                </New>
            </Item>
        </Array>
    </Arg>

    <Set name="port">8443</Set>
</New>
</Arg>
</Call>

</Configure>

```

In this case five connection factories are created and linked together by their protocol names:

"SSL-npn"

The default protocol is identified by the first connection factory, which in this case is a `SslConnectionFactory` instantiated with "npn" as the next protocol. Thus accepted endpoints are associated with an `SslConnection` instance that is chained to an `NextProtoNegoServerConnection` instance created by the "npn" connection factory.

"npn"

This is the `NPNServerConnectionFactory` chained to the `SslConnectionFactory`. The NPN connections negotiate with the client for the next protocol and then a factory of that name is looked up to create a connection to replace the NPN connection. If NPN is not supported, the defaultProtocol is configured as "http/1.1".

"spdy/3"

The factory NPN connections use if SPDY version 3 is negotiated.

"spdy/2"

The factory NPN connections use if SPDY version 2 is negotiated.

"http/1.1"

The factory NPN connections use if HTTP version 1.1 is negotiated or if NPN is not supported. Note that HTTP/1.1 can also handle HTTP/1.0.

Configuring SSL

This document provides an overview of how to configure SSL (also known as TLS) for Jetty.

Understanding Certificates and Keys

Configuring SSL can be a confusing experience of keys, certificates, protocols and formats, thus it helps to have a reasonable understanding of the basics. The following links provide some good starting points:

- Certificates:
 - [SSL Certificates HOWTO](#)
 - [Mindprod Java Glossary: Certificates](#)
- Keytool:
 - [Keytool for Unix](#)
 - [Keytool for Windows](#)
- Other tools:
 - [IBM Keyman](#)
- OpenSSL:
 - [OpenSSL HOWTO](#)
 - [OpenSSL FAQ](#)

OpenSSL vs. Keytool

For testing, the `keytool` utility bundled with the JDK provides the simplest way to generate the key and certificate you need.

You can also use the OpenSSL tools to generate keys and certificates, or to convert those that you have used with Apache or other servers. Since Apache and other servers commonly use the OpenSSL tool suite to generate and manipulate keys and certificates, you might already have some keys and certificates created by OpenSSL, or you might also prefer the formats OpenSSL produces.

If you want the option of using the same certificate with Jetty or a web server such as Apache not written in Java, you might prefer to generate your private key and certificate with OpenSSL.

Configuring Jetty for SSL

To configure Jetty for SSL, complete the tasks in the following sections:

- the section called “Generating Key Pairs and Certificates”
- the section called “Requesting a Trusted Certificate”
- the section called “Loading Keys and Certificates”
- the section called “Configuring SslContextFactory”

Generating Key Pairs and Certificates

The simplest way to generate keys and certificates is to use the `keytool` application that comes with the JDK, as it generates keys and certificates directly into the keystore. See the section called “Generating Keys and Certificates with JDK's keytool”.

If you already have keys and certificates, see the section called “Loading Keys and Certificates” to load them into a JSSE keystore. This section also applies if you have a renewal certificate to replace one that is expiring.

The examples below generate only basic keys and certificates. You should read the full manuals of the tools you are using if you want to specify:

- the key size
- the certificate expiration date
- alternate security providers

Generating Keys and Certificates with JDK's keytool

The following command generates a key pair and certificate directly into file `keystore`:

```
$ keytool -keystore keystore -alias jetty -genkey -keyalg RSA
```



Note

The DSA key algorithm certificate produces an error after loading several pages. In a browser, it displays a message "Could not establish an encrypted connection because certificate presented by localhost has an invalid signature." The solution is to use RSA for the key algorithm.

This command prompts for information about the certificate and for passwords to protect both the keystore and the keys within it. The only mandatory response is to provide the fully qualified host name of the server at the "first and last name" prompt. For example:

```
$ keytool -keystore keystore -alias jetty -genkey -keyalg RSA
Enter keystore password: password
What is your first and last name?
[Unknown]: jetty.eclipse.org
What is the name of your organizational unit?
```

```
[Unknown]: Jetty
What is the name of your organization?
[Unknown]: Mort Bay Consulting Pty. Ltd.
What is the name of your City or Locality?
[Unknown]:
What is the name of your State or Province?
[Unknown]:
What is the two-letter country code for this unit?
[Unknown]:
Is CN=jetty.eclipse.org, OU=Jetty, O=Mort Bay Consulting Pty. Ltd.,
L=Unknown, ST=Unknown, C=Unknown correct?
[no]: yes

Enter key password for <jetty>
      (RETURN if same as keystore password):
$
```

You now have the minimal requirements to run an SSL connection and could proceed directly to [configure an SSL connector](#). However the browser will not trust the certificate you have generated, and prompts the user to this effect. While what you have at this point is often sufficient for testing, most public sites need a trusted certificate, as shown in the section [generating a CSR with keytool](#).

Generating Keys and Certificates with OpenSSL

The following command generates a key pair in the file `jetty.key`:

```
$ openssl genrsa -des3 -out jetty.key
```

You might also want to use the `-rand` file argument to provide an arbitrary file that helps seed the random number generator.

The following command generates a certificate for the key into the file `jetty.crt`:

```
$ openssl req -new -x509 -key jetty.key -out jetty.crt
```

The next command prompts for information about the certificate and for passwords to protect both the keystore and the keys within it. The only mandatory response is to provide the fully qualified host name of the server at the "Common Name" prompt. For example:

```
$ openssl genrsa -des3 -out jetty.key
Generating RSA private key, 512 bit long modulus
.....+++++
e is 65537 (0x10001)
Enter pass phrase for jetty.key:
Verifying - Enter pass phrase for jetty.key:

$ openssl req -new -x509 -key jetty.key -out jetty.crt
Enter pass phrase for jetty.key:
You are about to be asked to enter information to be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there is a default value,
If you enter '.', the field is left blank.
-----
Country Name (2 letter code) [AU]::
State or Province Name (full name) [Some-State]::
Locality Name (eg, city) []::.
Organization Name (eg, company) [Internet Widgets Pty Ltd]:Mort Bay Consulting Pty. Ltd.
Organizational Unit Name (eg, section) []:Jetty
Common Name (eg, YOUR name) []:jetty.eclipse.org
Email Address []:

$
```

You now have the minimal requirements to run an SSL connection and could proceed directly to the section called "Loading Keys and Certificates" to load these keys and certificates into a JSSE keystore. However the browser will not trust the certificate you have generated, and prompts the user to this

effect. While what you have at this point is often sufficient for testing, most public sites need a trusted certificate, as shown in the section, the section called “Generating a CSR from OpenSSL” to obtain a certificate.

Using Keys and Certificates from Other Sources

If you have keys and certificates from other sources, you can proceed directly to the section called “Loading Keys and Certificates”.

Requesting a Trusted Certificate

The keys and certificates generated with JDK's `keytool` and OpenSSL are sufficient to run an SSL connector. However the browser will not trust the certificate you have generated, and it will prompt the user to this effect.

To obtain a certificate that most common browsers will trust, you need to request a well-known certificate authority (CA) to sign your key/certificate. Such trusted CAs include: AddTrust, Entrust, GeoTrust, RSA Data Security, Thawte, VISA, ValiCert, Verisign, and beTRUSTed, among others. Each CA has its own instructions (look for JSSE or OpenSSL sections), but all involve a step that generates a certificate signing request (CSR).

Generating a CSR with `keytool`

The following command generates the file `jetty.csr` using `keytool` for a key/cert already in the keystore:

```
$ keytool -certreq -alias jetty -keystore keystore -file jetty.csr
```

Generating a CSR from OpenSSL

The following command generates the file `jetty.csr` using OpenSSL for a key in the file `jetty.key`:

```
$ openssl req -new -key jetty.key -out jetty.csr
```

Notice that this command uses only the existing key from `jetty.key` file, and not a certificate in `jetty.crt` as generated with OpenSSL. You need to enter the details for the certificate again.

Loading Keys and Certificates

Once a CA has sent you a certificate, or if you generated your own certificate without `keytool`, you need to load it into a JSSE keystore.



Note

You need both the private key and the certificate in the JSSE keystore. You should load the certificate into the keystore used to generate the CSR with `keytool`. If your key pair is not already in a keystore (for example, because it has been generated with OpenSSL), you need to use the PKCS12 format to load both key and certificate (see [PKCS12 Keys & Certificates](#)).

Loading Certificates with `keytool`

You can use `keytool` to load a certificate in PEM form directly into a keystore. The PEM format is a text encoding of certificates; it is produced by OpenSSL, and is returned by some CAs. An example PEM file is:

```
jetty.crt
-----BEGIN CERTIFICATE-----
```

```

MIICSDCCAFKgAwIBAgIBADANBgkqhkiG9w0BAQQFADBUMSYwJAYDVQQKEx1Nb3J0
IEJheSBDb25zdWx0aW5nIFB0eS4gTHRkLjEOMAwGA1UECxMFSmV0dHkxGjAYBgNV
BAMTEWpldHr5Lm1vcnRiYXkub3JnMB4XDTCMDQwNjEZMTk1MFoXDTAzMDUwNjEZ
MTk1MFowVDEmMCQGA1UEChMdTW9ydcBCYXkgQ29uc3VsdGluyBQdHkuIEx0ZC4x
DjAMBgNVBAsTBUp1dHR5MRowGAYDVQQDExFqZXRo0eS5tb3J0YmF5Lm9yZzBcMA0G
CSqGS1b3DQEBAQUAA0sAMEgCQQC5V4oZeVdhhdHqa9L2/ZnKySPWUqqy81riNfAJ
7uALW0kEv/Lt1g34d00cvvt/PK8/bu4dlolnJx1SpiMzbKsFAgMBAAGjga4wgasw
HQYDVR0OBBYEFFV1gbB1XRvUx1UofmifQJS/MCYwMHwGA1UdIwR1MHOAFFV1gbB1
XRvUx1UofmifQJS/MCYwoVikVjBUMSYwJAYDVQQKEx1Nb3J0IEJheSBDb25zdWx0
aW5nIFB0eS4gTHRkLjEOMAwGA1UECxMFSmV0dHkxGjAYBgNVBAMTEWpldHr5Lm1v
cnRiYXkub3JnggEAMAwGA1UdEwQFMAMBAf8wDQYJKoZIhvCNQEEBQADQQA6NkaV
OtXzP4ayzBcgK/qSCmF44jdcARmrXhixUcXzjxsLjsJeYPJojhUdc2LQKy+p4ki8
Rcz6oCRvCGCe5kDB
-----END CERTIFICATE-----

```

The following command loads a PEM encoded certificate in the `jetty.crt` file into a JSSE keystore:

```
$ keytool -keystore keystore -import -alias jetty -file jetty.crt -trustcacerts
```

If the certificate you receive from the CA is not in a format that `keytool` understands, you can use the `openssl` command to convert formats:

```
$ openssl x509 -in jetty.der -inform DER -outform PEM -out jetty.crt
```

Loading Keys and Certificates via PKCS12

If you have a key and certificate in separate files, you need to combine them into a PKCS12 format file to load into a new keystore. The certificate can be one you generated yourself or one returned from a CA in response to your CSR.

The following OpenSSL command combines the keys in `jetty.key` and the certificate in the `jetty.crt` file into the `jetty.pkcs12` file:

```
$ openssl pkcs12 -inkey jetty.key -in jetty.crt -export -out jetty.pkcs12
```

If you have a chain of certificates, because your CA is an intermediary, build the PKCS12 file as follows:

```
$ cat example.crt intermediate.crt [intermediate2.crt] ... rootCA.crt > cert-chain.txt
$ openssl pkcs12 -export -inkey example.key -in cert-chain.txt -out example.pkcs12
```

The order of certificates must be from server to rootCA, as per RFC2246 section 7.4.2.

OpenSSL asks for an *export password*. A non-empty password is required to make the next step work. Then load the resulting PKCS12 file into a JSSE keystore with `keytool`:

```
$ keytool -importkeystore -srckeystore jetty.pkcs12 -srcstoretype PKCS12 -destkeystore
keystore
```

Configuring SslContextFactory

The generated SSL certificates held in the key store are configured on Jetty by injection an instance of [SslContextFactory](#) object and passing it to the connector's `SslConnectionFactory`, which is done in the jetty distribution by both in [jetty-https.xml](#) and [jetty-spdy.xml](#). Since SPDY is able to handle HTTPS also, typically you will configure jetty to use one of these configuration files or the other, which can be done either on the command line or by editing the `start.ini` file.

These configuration files create an `SslContextFactory` instance with the ID "sslContextFactory":

```
<New id="sslContextFactory" class="org.eclipse.jetty.util.ssl.SslContextFactory">
<Set name="KeyStorePath"><Property name="jetty.home" default="." />/etc/keystore</Set>
<Set name="KeyStorePassword">OBF:1vnvlzlolx8elvnwlvn6lx8g1zlulvn4</Set>
<Set name="KeyManagerPassword">OBF:lu2ulwm1l1z7s1z7alwnllu2g</Set>
```

```
<Set name="TrustStorePath"><Property name="jetty.home" default=". " />/etc/keystore</Set>
<Set name="TrustStorePassword">OBF:1vny1zl0lx8elvnwlvn6lx8g1zlulvn4</Set>
</New>
```

This example uses the keystore distributed with jetty. To use your own keystore you need to update at least the following settings:

KeyStorePath

You can either replace the provided keystore with your own, or change the configuration to point to a different file. Note that as a keystore is vital security information, it can be desirable to locate the file in a directory with very restricted access.

KeyStorePassword

The keystore password may be set here in plain text, or as some protection from casual observation, it may be obfuscated using the [Password](#) class.

The Truststore is used if validating client certificates and is typically set to the same keystore.

The keyManagerPassword is passed as the password arg to KeyManagerFactory.init(...). If there is no keymanagerpassword, then the keystorepassword is used instead. If there is no trustmanager set, then the keystore is used as the trust store and the keystorepassword is used as the truststore password

There is no need to create a new instance of SslContextFactory using the xml above as one exists in jetty-ssl.xml. Edit the paths and passwords in jetty-ssl.xml and ensure you add following lines to start.ini after the line jetty.dump.stop=:

```
etc/jetty-ssl.xml
etc/jetty-https.xml
```

Optionally, configure the https port either in jetty-https.xml like this:

```
<Set name="port">8443</Set>
```

or via the commandline e.g.

```
$ java -jar start.jar https.port=8443
```

Disabling/Enabling specific cipher suites

For example to avoid the BEAST attack it is necessary to configure a specific set of cipher suites. This can either be done via [SslContext.setIncludeCipherSuites\(java.lang.String...\)](#) or via [SslContext.setExcludeCipherSuites\(java.lang.String...\)](#).



Note

It's crucial that you use the exact names of the cipher suites as used/known by the JDK. You can get them by obtaining an instance of SSLEngine and call getSupportedCipherSuites(). Tools like ssllabs.com might report slightly different names which will be ignored.

Both setIncludeCipherSuites and setExcludeCipherSuites can be feed by the exact cipher suite name used in the jdk or by using regular expressions.

Here's a couple of examples on how to include only RC4 cipher suites. All of them will protect the server from the [BEAST](#) attack.

Include a preferred set of cipher suites:

```

<Set name="IncludeCipherSuites">
  <Array type="String">
    <Item>TLS_RSA_WITH_RC4_128_MD5</Item>
    <Item>TLS_RSA_WITH_RC4_128_SHA</Item>
    <Item>TLS_ECDHE_RSA_WITH_RC4_128_SHA</Item>
  </Array>
</Set>

```

Include all RC4 cipher suites by using a regex:

```

<Set name="IncludeCipherSuites">
  <Array type="String">
    <Item>.*RC4.*</Item>
  </Array>
</Set>

```

Exclude all non RC4 cipher suites:

```

<Set name="ExcludeCipherSuites">
  <Array type="String">
    <Item>SSL_RSA_WITH_DES_CBC_SHA</Item>
    <Item>SSL_DHE_RSA_WITH_DES_CBC_SHA</Item>
    <Item>SSL_DHE_DSS_WITH_DES_CBC_SHA</Item>
    <Item>SSL_RSA_EXPORT_WITH_RC4_40_MD5</Item>
    <Item>SSL_RSA_EXPORT_WITH_DES40_CBC_SHA</Item>
    <Item>SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA</Item>
    <Item>SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA</Item>
    <Item>TLS_DHE_RSA_WITH_AES_128_CBC_SHA256</Item>
    <Item>TLS_RSA_WITH_AES_128_CBC_SHA</Item>
    <Item>TLS_DHE_RSA_WITH_AES_128_CBC_SHA</Item>
    <Item>TLS_RSA_WITH_AES_128_CBC_SHA256</Item>
    <Item>TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA</Item>
    <Item>TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256</Item>
    <Item>TLS_RSA_WITH_3DES_EDE_CBC_SHA</Item>
    <Item>TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA</Item>
    <Item>TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA</Item>
    <Item>RSA_WITH_3DES_EDE_CBC_SHA</Item>
  </Array>
</Set>

```

Note



It is recommended to use the `IncludeCipherSuites` with the regex unless you've reasons you need to specify specific cipher suites. This configuration will adapt to any additions/removals of cipher suites to new versions of the JDK.

Configuring SSL Connector and Port

This `SslContextFactory` instance created above is injected into a `SslConnectionFactory` instance to be used when accepting network connections, which in turn is injected into an instance of `ServerConnector`. For example from [jetty-https.xml](#):

```

<Call id="sslConnector" name="addConnector">
  <Arg>
    <New class="org.eclipse.jetty.server.ServerConnector">
      <Arg name="server"><Ref refid="Server" /></Arg>
      <Arg name="factories">
        <Array type="org.eclipse.jetty.server.ConnectionFactory">
          <Item>
            <New class="org.eclipse.jetty.server.SslConnectionFactory">
              <Arg name="next">http/1.1</Arg>
              <Arg name="sslContextFactory"><Ref refid="sslContextFactory"/></Arg>
            </New>
          </Item>
        </Array>
      </Arg>
    </New>
  </Arg>
</Call>

```

```
<New class="org.eclipse.jetty.server.HttpConnectionFactory">
    <Arg name="config"><Ref refid="tlsHttpConfig"/></Arg>
</New>
</Item>
</Array>
</Arg>
<Set name="host"><Property name="jetty.host" /></Set>
<Set name="port"><Property name="jetty.tls.port" default="8443" /></Set>
<Set name="idleTimeout">30000</Set>
</New>
</Arg>
</Call>
```

Note also that the SSL connector port is set directly on the ServerConnector instance.

Renewing Certificates

If you are updating your configuration to use a newer certificate, as when the old one is expiring, just load the newer certificate as described in the section, the section called “Loading Keys and Certificates”. If you imported the key and certificate originally using the PKCS12 method, use an alias of “1” rather than “jetty”, because that is the alias the PKCS12 process enters into the keystore.

Setting Port 80 Access for a Non-Root User

On Unix-based systems, port 80 is protected; typically only the superuser root can open it. For security reasons, it is not desirable to run the server as root. This page presents several options to access port 80 as a non-root user, including using ipchains, iptables, Jetty’s SetUID feature, xinetd, and the Solaris 10 User Rights Management Framework.

Using ipchains

On some Linux systems you can use the *ipchains REDIRECT* mechanism to redirect from one port to another inside the kernel (if ipchains is not available, then usually iptables is):

```
# /sbin/ipchains -I input --proto TCP --dport 80 -j REDIRECT 8080
```

This command instructs the system as follows: “Insert into the kernel’s packet filtering the following as the first rule to check on incoming packets: if the protocol is TCP and the destination port is 80, redirect the packet to port 8080”. Be aware that your kernel must be compiled with support for ipchains (virtually all stock kernels are). You must also have the ipchains command-line utility installed. You can run this command at any time, preferably just once, since it inserts another copy of the rule every time you run it.

Using iptables

On many Linux systems you can use the iptables REDIRECT mechanism to redirect from one port to another inside the kernel (if iptables is not available, then usually ipchains is).

You need to add something like the following to the startup scripts or your firewall rules:

```
# /sbin/iptables -t nat -I PREROUTING -p tcp --dport 80 -j REDIRECT --to-port 8080
```

The underlying model of iptables is different from ipchains, so the forwarding normally happens only to packets originating off-box. You also need to allow incoming packets to port 8080 if you use iptables as a local firewall.

Be careful to place rules like this one early in your *input* chain. Such rules must precede any rule that accepts the packet, otherwise the redirection won’t occur. You can insert as many rules as required if your server needs to listen on multiple ports, as for HTTPS.

Configuring Jetty's SetUID Feature

[SetUID](#) is a technique that uses Unix-like file system access right to allow users to run an executable that would otherwise require higher privileges.

Jetty's SetUID module allows you to run Jetty as a normal user even when you need to run Jetty on port 80 or 443. The module is hosted as part of the Jetty ToolChain project and it is released independently from Jetty itself (and as such it has a different version than Jetty releases). You can find the source code [here](#), while the Maven coordinates are:

```
<dependency>
  <groupId>org.eclipse.jetty.toolchain.setuid</groupId>
  <artifactId>jetty-setuid-java</artifactId>
</dependency>
```

Jetty's SetUID module provides an implementation for Linux and OSX.

Since the SetUID feature requires native code, you may need to build it for your environment.

In order to use Jetty's SetUID module, you need to copy file `jetty-setuid-java-<version>.jar` into `$jetty.home/lib`, and make sure that the native library file (for Linux this file is called `libsetuid-linux.so`) is present in the native library path of the JVM (see also the section called "Configuring Jetty for SetUID").

Jetty's SetUID module also provides a default configuration file in the Jetty XML format, in file `jetty-setuid-java-<version>-config.jar`. This file can be unjarred in `$jetty.home/lib` and it will provide a `$jetty.home/etc/jetty-setuid.xml` file that you can customize. Alternatively, follow the next section that specifies how to create a Jetty XML config file for Jetty's SetUID.

Creating a Jetty Config File

Jetty SetUID module works by replacing the usual `org.eclipse.jetty.server.Server` instance with a `org.eclipse.jetty.setuid.SetUIDServer` instance.

Create a Jetty config file as follows:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN" "http://
jetty.mortbay.org/jetty/configure_9_0.dtd">
<Configure id="Server" class="org.eclipse.jetty.setuid.SetUIDServer">
  <Set name="umask">UMASK</Set>
  <Set name="uid">USERID</Set>
</Configure>
```

where you replace:

- *UMASK* with the umask setting you want the process to have.
- You must enter it in decimal. That is, if you want the effect of umask 022, you must enter

```
<Set name="umask">18</Set>
```

- If you prefer octal, enter

```
<Set name="umaskOctal">022</Set>
```

- You can remove this line if you don't want to change this the umask at runtime.
- Set it to 002 if you get an error to the effect that root does not have permission to write to the log file.

- *USERID* with the ID of the user you want the process to execute as once the ports have been opened.

Configuring Jetty for SetUID

The easiest way to do this is to edit the `$jetty.home/start.ini` file:

- uncomment `--exec`
- add `-Djava.library.path=lib/setuid`, the path where the native library can be found
- add an option for SetUID:
`OPTIONS=Server,jsp,jmx,resources,websocket,ext,jta,plus,jdbc,annotations,setuid`
- add `etc/jetty-setuid.xml` as the **first** file in the configuration file section. This allows the Server instance to be created as `org.eclipse.jetty.setuid.SetUIDServer`

Important



You must ensure that the `etc/jetty-setuid.xml` file is **first** in the list of config files.

Running Jetty as Root User

Having edited `start.ini` as advised above, to run jetty as the root user:

- Switch to the userid of your choice.
- Optionally set the umask of your choice.
- Enter the following command:

```
sudo java -jar start.jar
```

Using xinetd

With modern Linux flavours, `inetd` has a newer, better big brother `xinetd`, that you can use to redirect network traffic. Since `xinetd` is driven by text files, all you need is a text editor. For detailed information, see .

There are two ways to give `xinetd` instructions:

- Add a new service to `etc/xinetd.conf`
- Add a new file to the directory `etc/xinetd.d`

The format is the same; if you have a look at the file/directory, you will get the picture.

The following entry redirects all inward TCP traffic on port 80 to port 8888 on the local machine. You can also redirect to other machines for gimp proxying:

```
service my_redirector
{
    type = UNLISTED
    disable = no
    socket_type = stream
    protocol = tcp
    user = root
    wait = no
    port = 80
    redirect = 127.0.0.1 8888
    log_type = FILE /tmp/somefile.log
}
```

caveats

Be aware of the following:

- Include a space on either side of the '=' or it is ignored.
- type = UNLISTED means that the name of the service does not have to be listed in /etc/services, but then you have to specify port and protocol. If you want to do use an existing service name, for example, http:

```
service http
{
    disable = no
    socket_type = stream
    user = root
    wait = no
    redirect = 127.0.0.1 8888
    log_type = FILE /tmp/somefile.log
}
```

Have a browse in /etc/services and it will all become clear.

- Logging might present certain security problems, so you might want to leave that out.
- xinetd is a hugely powerful and configurable system, so expect to do some reading.

Using the Solaris 10 User Rights Management Framework

Solaris 10 provides a User Rights Management framework that can permit users and processes superuser-like abilities:

```
usermod -K defaultpriv=basic,net_privaddr myself
```

Now the myself user can bind to port 80.

Refer to the [Solaris 10](#) and [Solaris 11 Security Services documentation](#) for more information.

Chapter 7. Configuring Security

Table of Contents

Jetty 9.1: Using the \${jetty.home} and \${jetty.base} Concepts to Configure Security	76
Authentication	83
Limiting Form Content	89
Aliased Files and Symbolic links	90
Secure Password Obfuscation	91
JAAS Support	93
Spnego Support	99

Jetty 9.1: Using the \${jetty.home} and \${jetty.base} Concepts to Configure Security

Jetty 9.1 introduces \${jetty.base} and \${jetty.home}.

- \${jetty.home} is the directory location for the jetty distribution (the binaries).
- \${jetty.base} is the directory location for your customizations to the distribution.

This separation:

- Allows you to manage multiple Jetty installations.
- Makes it simple to retain your current configuration when you upgrade your Jetty distribution.

For more information, see the section called “Managing Jetty Base and Jetty Home”.

Further, Jetty 9.1 parameterizes all of the standard configuration XMLs. For SSL, parameters are now just properties in the `start.ini`, reducing to eliminating the need to edit XML files.

Jetty 9.1 also introduces modules. Instead of explicitly listing all the libraries, properties, and XML files for a feature, Jetty includes software modules, and the `start.jar` mechanism allows you to create new modules. You define a module in a `modules/*.mod` file, including the libraries, dependencies, XML, and template INI files for a Jetty feature. Thus you can use a single `--module=name` command line option as the equivalent of specifying many `--lib=location`, `feature.xml`, `name=value` arguments for a feature and all its dependencies. Modules use their dependencies to control the ordering of libraries and XML files. For more information, see the section called “Managing Startup Modules”.

Configuring SSL in Jetty 9.1

This page describes how to configure SSL in Jetty 9.1. It provides an example of using the \${jetty.home} and \${jetty.base} to maximum effect. It also includes a detailed explanation of how modules work.

This example assumes you have the jetty-distribution unpacked in `/home/user/jetty-distribution-9.1.0.RC0`.



Note

OBF passwords are not secure, just protected from casual observation.

1. Create a base directory anywhere.

```
[/home/user]$ mkdir my-base
```

```
[ /home/user]$ cd my-base
```

2. Add the modules for SSL, HTTP, and webapp deployment.

```
[my-base]$ java -jar /home/user/jetty-distribution-9.1.0.RC0/start.jar --add-to-start=ssl,http,deploy

ssl          initialised in ${jetty.base}/start.ini (appended)
ssl          enabled in      ${jetty.base}/start.ini
DOWNLOAD: http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/jetty-
server/src/main/config/etc/keystore to etc/keystore
server      initialised in ${jetty.base}/start.ini (appended)
server      enabled in      ${jetty.base}/start.ini
http        initialised in ${jetty.base}/start.ini (appended)
http        enabled in      ${jetty.base}/start.ini
server      enabled in      ${jetty.base}/start.ini
deploy     initialised in ${jetty.base}/start.ini (appended)
deploy     enabled in      ${jetty.base}/start.ini
MKDIR: ${jetty.base}/webapps
server      enabled in      ${jetty.base}/start.ini
```

3. Look at your directory.

```
[my-base]$ ls -la
total 20
drwxrwxr-x  4 user group 4096 Oct  8 06:55 ../
drwxr-xr-x 103 user group 4096 Oct  8 06:53 ../
drwxrwxr-x  2 user group 4096 Oct  8 06:55 etc/
-rw-rw-r--  1 user group  815 Oct  8 06:55 start.ini
drwxrwxr-x  2 user group 4096 Oct  8 06:55 webapps/
```

4. Copy your WAR files into webapps.

```
[my-base]$ ls -la
[my-base]$ cp ~/code/project/target/gadget.war webapps/
```

5. Copy your keystore into place.

```
[my-base]$ cp ~/code/project/keystore etc/keystore
```

6. Edit the start.ini to configure your SSL settings.

```
[my-base]$ cat start.ini
```

7. Initialize module ssl.

```
--module=ssl
```

8. Define the port to use for secure redirection.

```
jetty.secure.port=8443
```

9. Set up a demonstration keystore and truststore.

```
jetty.keystore=etc/keystore
jetty.truststore=etc/keystore
```

10. Set the demonstration passwords.

```
jetty.keystore.password=OBF:1vny1z1o1x8e1vnw1vn61x8g1zlulvn4
jetty.keymanager.password=OBF:lu2ulwm1l2z7s1z7a1wn1l1u2g
jetty.truststore.password=OBF:1vny1z1o1x8e1vnw1vn61x8g1zlulvn4
```

11. Initialize the module server.

```
--module=server
threads.min=10
threads.max=200
threads.timeout=60000
```

```
#jetty.host=myhost.com
jetty.dump.start=false
jetty.dump.stop=false
```

12. Initialize module http.

```
--module=http
jetty.port=8080
http.timeout=30000
```

13. Initialize module deploy.

```
--module=deploy
```

Look at the configuration you have at this point.

```
[my-base]$ java -jar /home/user/jetty-distribution-9.1.0.RC0/start.jar --list-config

Java Environment:
-----
java.home=/home/user/java/jdk-7u21-x64/jre
java.vm.vendor=Oracle Corporation
java.vm.version=23.21-b01
java.vm.name=Java HotSpot(TM) 64-Bit Server VM
java.vm.info=mixed mode
java.runtime.name=Java(TM) SE Runtime Environment
java.runtime.version=1.7.0_21-b11
java.io.tmpdir=/tmp

Jetty Environment:
-----
jetty.home=/home/user/jetty-distribution-9.1.0.RC0
jetty.base=/home/user/my-base
jetty.version=9.1.0.RC0

JVM Arguments:
-----
(no jvm args specified)

System Properties:
-----
jetty.base = /home/user/my-base
jetty.home = /home/user/jetty-distribution-9.1.0.RC0

Properties:
-----
http.timeout = 30000
jetty.dump.start = false
jetty.dump.stop = false
jetty.keymanager.password = OBF:lu2u1wm1lz7s1z7alwn1lu2g
jetty.keystore = etc/keystore
jetty.keystore.password = OBF:lvny1zl0lx8elvnw1vn61x8g1zlulvn4
jetty.port = 8080
jetty.secure.port = 8443
jetty.truststore = etc/keystore
jetty.truststore.password = OBF:lvny1zl0lx8elvnw1vn61x8g1zlulvn4
threads.max = 200
threads.min = 10
threads.timeout = 60000

Jetty Server Classpath:
-----
Version Information on 11 entries in the classpath.
Note: order presented here is how they would appear on the classpath.
      changes to the --module=name command line options will be reflected here.
0:          3.1.0 | ${jetty.home}/lib/servlet-api-3.1.jar
1:          3.1.RC0 | ${jetty.home}/lib/jetty-schemas-3.1.jar
2:          9.1.0.RC0 | ${jetty.home}/lib/jetty-http-9.1.0.RC0.jar
3:          9.1.0.RC0 | ${jetty.home}/lib/jetty-continuation-9.1.0.RC0.jar
4:          9.1.0.RC0 | ${jetty.home}/lib/jetty-server-9.1.0.RC0.jar
5:          9.1.0.RC0 | ${jetty.home}/lib/jetty-xml-9.1.0.RC0.jar
6:          9.1.0.RC0 | ${jetty.home}/lib/jetty-util-9.1.0.RC0.jar
7:          9.1.0.RC0 | ${jetty.home}/lib/jetty-io-9.1.0.RC0.jar
8:          9.1.0.RC0 | ${jetty.home}/lib/jetty-servlet-9.1.0.RC0.jar
9:          9.1.0.RC0 | ${jetty.home}/lib/jetty-webapp-9.1.0.RC0.jar
```

```
10:           9.1.0.RC0 | ${jetty.home}/lib/jetty-deploy-9.1.0.RC0.jar

Jetty Active XMLs:
-----
${jetty.home}/etc/jetty.xml
${jetty.home}/etc/jetty-http.xml
${jetty.home}/etc/jetty-ssl.xml
${jetty.home}/etc/jetty-deploy.xml
```

Now start Jetty.

```
[my-base]$ java -jar /home/user/jetty-distribution-9.1.0.RC0/start.jar
2013-10-08 07:06:55.837:INFO:oejs.Server:main: jetty-9.1.0.RC0
2013-10-08 07:06:55.853:INFO:oejdp.ScanningAppProvider:main: Deployment monitor [file:/home/joakim/my-base/webapps/] at interval 1
2013-10-08 07:06:55.872:INFO:oejs.ServerConnector:main: Started
ServerConnector@72974691{HTTP/1.1}{0.0.0.0:8080}
```

Reviewing the Configuration

The following sections review this configuration.

\${jetty.base} and \${jetty.home}

First notice the separation of \${jetty.base} and \${jetty.home}.

- \${jetty.home} is where your distribution lies, unchanged, unedited.
- \${jetty.base} is where your customizations are.

Modules

Notice that you have --module=<name> here and there; you have wrapped up the goal of a module (libs, configuration XMLs, and properties) into a single unit, with dependencies on other modules.

You can see the list of modules:

```
[my-base]$ java -jar /home/user/jetty-distribution-9.1.0.RC0/start.jar --list-modules

Jetty All Available Modules:
-----

Module: annotations
    LIB: lib/jetty-annotations-${jetty.version}.jar
    LIB: lib/annotations/*.jar
    XML: etc/jetty-annotations.xml
depends: [plus]

Module: client
    LIB: lib/jetty-client-${jetty.version}.jar
depends: []

Module: debug
    XML: etc/jetty-debug.xml
depends: [server]

Module: deploy
    LIB: lib/jetty-deploy-${jetty.version}.jar
    XML: etc/jetty-deploy.xml
depends: [webapp]
enabled: ${jetty.base}/start.ini

Module: ext
    LIB: lib/ext/*.jar
depends: []

Module: http
    XML: etc/jetty-http.xml
depends: [server]
enabled: ${jetty.base}/start.ini
```

```

Module: https
    XML: etc/jetty-https.xml
depends: [ssl]

Module: ipaccess
    XML: etc/jetty-ipaccess.xml
depends: [server]

Module: jaas
    LIB: lib/jetty-jaas-${jetty.version}.jar
    XML: etc/jetty-jaas.xml
depends: [server]

Module: jaspi
    LIB: lib/jetty-jaspi-${jetty.version}.jar
    LIB: lib/jaspi/*.jar
depends: [security]

Module: jmx
    LIB: lib/jetty-jmx-${jetty.version}.jar
    XML: etc/jetty-jmx.xml
depends: []

Module: jndi
    LIB: lib/jetty-jndi-${jetty.version}.jar
    LIB: lib/jndi/*.jar
depends: [server]

Module: jsp
    LIB: lib/jsp/*.jar
depends: [servlet]

Module: jvm
depends: []

Module: logging
    XML: etc/jetty-logging.xml
depends: []

Module: lowresources
    XML: etc/jetty-lowresources.xml
depends: [server]

Module: monitor
    LIB: lib/jetty-monitor-${jetty.version}.jar
    XML: etc/jetty-monitor.xml
depends: [client, server]

Module: npn
depends: []

Module: plus
    LIB: lib/jetty-plus-${jetty.version}.jar
    XML: etc/jetty-plus.xml
depends: [server, security, jndi]

Module: proxy
    LIB: lib/jetty-proxy-${jetty.version}.jar
    XML: etc/jetty-proxy.xml
depends: [client, server]

Module: requestlog
    XML: etc/jetty-requestlog.xml
depends: [server]

Module: resources
    LIB: resources
depends: []

Module: rewrite
    LIB: lib/jetty-rewrite-${jetty.version}.jar
    XML: etc/jetty-rewrite.xml
depends: [server]

Module: security
    LIB: lib/jetty-security-${jetty.version}.jar
depends: [server]

```

```

Module: server
    LIB: lib/servlet-api-3.1.jar
    LIB: lib/jetty-schemas-3.1.jar
    LIB: lib/jetty-http-${jetty.version}.jar
    LIB: lib/jetty-continuation-${jetty.version}.jar
    LIB: lib/jetty-server-${jetty.version}.jar
    LIB: lib/jetty-xml-${jetty.version}.jar
    LIB: lib/jetty-util-${jetty.version}.jar
    LIB: lib/jetty-io-${jetty.version}.jar
    XML: etc/jetty.xml
depends: []
enabled: ${jetty.base}/start.ini

Module: servlet
    LIB: lib/jetty-servlet-${jetty.version}.jar
depends: [server]

Module: servlets
    LIB: lib/jetty-servlets-${jetty.version}.jar
depends: [servlet]

Module: setuid
    LIB: lib/setuid/jetty-setuid-java-1.0.1.jar
    XML: etc/jetty-setuid.xml
depends: [server]

Module: spdy
    LIB: lib/spdy/*.jar
    XML: etc/jetty-ssl.xml
    XML: etc/jetty-spdy.xml
depends: [ssl, npn]

Module: ssl
    XML: etc/jetty-ssl.xml
depends: [server]
enabled: ${jetty.base}/start.ini

Module: stats
    XML: etc/jetty-stats.xml
depends: [server]

Module: webapp
    LIB: lib/jetty-webapp-${jetty.version}.jar
depends: [servlet]

Module: websocket
    LIB: lib/websocket/*.jar
depends: [annotations]

Module: xinetd
    XML: etc/jetty-xinetd.xml
depends: [server]

Jetty Active Module Tree:
-----
+ Module: server [enabled]
  + Module: http [enabled]
  + Module: servlet [transitive]
  + Module: ssl [enabled]
    + Module: webapp [transitive]
      + Module: deploy [enabled]

```

These are the modules by name, the libraries they bring in, the XML configurations they use, the other modules they depend on (even optional ones), and if the module is in use, where it was enabled.

While you can manage the list of active modules yourself, it is much easier to edit the \${jetty.base}/start.ini.

If you want to start using a new module:

```
[my-base] $ java -jar ../jetty-distribution-9.1.0.RC0/start.jar --add-to-start=https
```

This adds the `--module=` lines and associated properties (the parameterized values mentioned above), to your `start.ini`.



Important

Leave the modules and XML files alone in the `${jetty.home}` directory; there is no need to be moving or copying them unless you want to make your own modules or override the behavior of an existing module.

Notice that your `${jetty.base} /start.ini` has no references to the XML files. That's because the module system and its graph of dependencies now dictate all of the XML files, and their load order.

Parameters

Next is parameterizing all of the standard configuration XMLs. In this example all of the SSL parameters are now just properties in the `start.ini`, reducing or eliminating the need to edit XML files.

Overriding `${jetty.home}` in `${jetty.base}`

Finally, you can override anything you see in `${jetty.home}` in `${jetty.base}` , even XML configurations and libraries.

For more information on the `start.jar` in 9.1, see the section called “Using start.jar”.

Summary of Configuring SSL in Jetty 9.1

1. Download and unpack Jetty 9.1 into `/home/user/jetty-distribution-9.1.0.RC1`.
2. Go to your base directory and just use the distribution, no editing.

```
[my-base]$ java -jar /home/user/jetty-distribution-9.1.0.RC1/start.jar
```

- The Jetty 9.1 distribution provides, out of the box, the XML configuration files, in this case `jetty-http.xml` and `jetty-ssl.xml`. You can find them in `${jetty.home} /etc/` directory.
- We have parameterized all of the configurable values in those XMLs. You can now set the values using simple properties, either on the command line, or within the `${jetty.base} /start.ini`.
- When you activate the module for HTTP or HTTPS, Jetty automatically adds the appropriate libraries and XML to start Jetty. Unless you have a highly custom setup (such as listening on two different ports, using SSL on each, each with its own keystore and configuration), you should have no need to be mucking around in XML files.

3. Use modules to configure HTTPS:

- http -> server
- https -> ssl -> server

You can find the details about the modules in `${jetty.home} /modules/`. For SSL they include `modules/http.mod`, `modules/https.mod`, `modules/ssl.mod`, and `modules/server.mod`.

Ideally, this level of detail is not important to you. What is important is that you want to use HTTPS and want to configure it. You accomplish that by adding the `--module=https` to your `start.ini`. By default, the module system keeps things sane, and transitively includes all dependent modules as well.

You can see what the configuration looks like, after all of the modules are resolved, without starting Jetty via:

```
[my-base] $ java -jar ../jetty-distribution-9.1.0.RC0/start.jar --list-config
```

Just because the JARs exist on disk does not mean that they are in use. The configuration controls what is used.

Use the `--list-config` to see the configuration. Notice that only a subset of the JARs from the distribution are in use. The modules you have enabled determine that subset.

```
[my-base]$ java -jar ~/jetty-distribution-9.1.0.RC0/start.jar --list-config
```

Authentication

There are two aspects to securing a web application(or context) within the Jetty server:

Authentication

The web application can be configured with a mechanism to determine the identity of the user. This is configured by a mix of standard declarations and jetty specific mechanisms and is covered in this section.

Authorization

Once the identify of the user is known (or not known), the web application can be configured via standard descriptors with security constraints that declare what resources that user may access.

Configuring an Authentication mechanism

The jetty server supports several standard authentication mechanisms: [BASIC](#); [DIGEST](#); [FORM](#); [CLIENT-CERT](#); and other mechanisms can be plugged in using the extensible [JASPI](#) or [SPNEGO](#) mechanisms.

Internally, configuring an authentication mechanism is done by setting an instance of a the [Authenticator](#) interface onto the [SecurityHandler](#) of the context, but in most cases it is done by declaring a `<login-config>` element in the standard web.xml descriptor or via annotations.

Below is an example taken from the [jetty-test-webapp web.xml](#) that configures BASIC authentication:

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Test Realm</realm-name>
</login-config>
```

The [jetty-test-webapp web.xml](#) also includes commented out examples of other DIGEST and FORM configuration:

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>Test Realm</realm-name>
  <form-login-config>
    <form-login-page>/logon.html?param=test</form-login-page>
    <form-error-page>/logonError.html?param=test</form-error-page>
  </form-login-config>
</login-config>
```

With FORM Authentication, you must also configure URLs of pages to generate a login form and handle errors. Below is a simple HTML form from the [test webapp logon.html](#):

```

<HTML>
<H1>FORM Authentication demo</H1>
<form method="POST" action="j_security_check">
<table border="0" cellspacing="2" cellpadding="1">
<tr>
  <td>Username:</td>
  <td><input size="12" value="" name="j_username" maxlength="25" type="text"></td>
</tr>
<tr>
  <td>Password:</td>
  <td><input size="12" value="" name="j_password" maxlength="25" type="password"></td>
</tr>
<tr>
  <td colspan="2" align="center">
    <input name="submit" type="submit" value="Login">
  </td>
</tr>
</table>
</form>
</HTML>

```

The Authentication mechanism declared for a context / web application defines how the server obtain authentication credentials from the client, but it does not define how the server checks if those credentials are valid. To check credentials, the server and/or context also need to be configured with a [LoginService](#) instance, which may be matched by the declared realm-name.

Security Realms

Security realms allow you to secure your web applications against unauthorized access. Protection is based on authentication that identifies who is requesting access to the webapp and access control that restricts what can be accessed and how it is accessed within the webapp.

A webapp statically declares its security requirements in its web.xml file. Authentication is controlled by the <login-config> element. Access controls are specified by <security-constraint> and <security-role-ref> elements. When a request is received for a protected resource, the web container checks if the user performing the request is authenticated, and if the user has a role assignment that permits access to the requested resource.

The Servlet Specification does not address how the static security information in the WEB-INF/web.xml file is mapped to the runtime environment of the container. For Jetty, the [LoginService](#) performs this function.

A LoginService has a unique name, and gives access to information about a set of users. Each user has authentication information (e.g. a password) and a set of roles associated with him/herself.

You may configure one or many different LoginServices depending on your needs. A single realm would indicate that you wish to share common security information across all of your web applications. Distinct realms allow you to partition your security information webapp by webapp.

When a request to a web application requires authentication or authorization, Jetty will use the <realm-name> sub-element inside <login-config> element in the web.xml file to perform an *exact match* to a LoginService.

Scoping Security Realms

A LoginService has a unique name, and is composed of a set of users. Each user has authentication information (for example, a password) and a set of roles associated with him/herself. You can configure one or many different realms depending on your needs.

- Configure a single LoginService to share common security information across all of your web applications.

- Configure distinct LoginServices to partition your security information webapp by webapp.

Globally Scoped

A LoginService is available to all web applications on a Server instance if you define it in a Jetty configuration file, for example \${jetty.home}/etc/jetty.xml. Here's an example of defining an in-memory type of LoginService called the [HashLoginService](#):

```
<Configure id="Server" class="org.eclipse.jetty.server.Server">
  <Call name="addBean">
    <Arg>
      <New class="org.eclipse.jetty.security.HashLoginService">
        <Set name="name">Test Realm</Set>
        <Set name="config"><SystemProperty name="jetty.home" default=". "/>/etc/
realm.properties</Set>
        <Set name="refreshInterval">0</Set>
      </New>
    </Arg>
  </Call>
</Configure>
```

If you define more than one LoginService on a Server, you will need to specify which one you want used for each context. You can do that by telling the context the name of the LoginService, or passing it the LoginService instance. Here's an example of doing both of these, using a [context xml file](#):

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Get name="securityHandler">
    <!-- Either: -->
    <Set name="loginService">
      <New class="org.eclipse.jetty.security.HashLoginService">
        <Set name="name">Test Realm</Set>
      </New>
    </Set>

    <!-- or if you defined a LoginService called "Test Realm" in jetty.xml : -->
    <Set name="realmName">Test Realm</Set>
  </Get>
```

Per-Webapp Scoped

Alternatively, you can define a LoginService for just a single web application. Here's how to define the same HashLoginService, but inside a [context xml file](#):

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath"/>/test</Set>
  <Set name="war"><SystemProperty name="jetty.home" default=". "/>/webapps/test</Set>
  <Get name="securityHandler">
    <Set name="loginService">
      <New class="org.eclipse.jetty.security.HashLoginService">
        <Set name="name">Test Realm</Set>
        <Set name="config"><SystemProperty name="jetty.home" default=". "/>/etc/
realm.properties</Set>
      </New>
    </Set>
  </Get>
</Configure>
```

Jetty provides a number of different LoginService types which can be seen in the next section.

Configuring a LoginService

A [LoginService](#) instance is required by each context/webapp that has a authentication mechanism, which is used to check the validity of the username and credentials collected by the authentication mechanism. Jetty provides the following implementations of LoginService:

[HashLoginService](#)

A user realm that is backed by a hash map that is filled either programatically or from a java properties file.

[JDBCLoginService](#)

Uses a JDBC connection to an SQL database for authentication

[DataSourceLoginService](#)

Uses a JNDI defined [DataSource](#) for authentication

[JAASLoginService](#)

Uses a [JAAS](#) provider for authentication, See the section on [JAAS support](#) for more information.

[SpnegoLoginService](#)

[SPNEGO](#) Authentication, See the section on [SPNEGO support](#) for more information.

An instance of a LoginService can be matched to a context/webapp either by:

- A LoginService instance may be set directly on the SecurityHandler instance via embedded code or IoC XML
- Matching the realm-name defined in web.xml with the name of a LoginService instance that has been added to the Server instance as a dependent bean.
- If only a single LoginService instance has been set on the Server then it is used as the login service for the context.

HashLoginService

The HashLoginService is a simple and efficient login service that loads usernames, credentials and roles from a java properties file in the format:

```
username: password[,rolename ...]
```

where:

username

is the user's unique identity

password

is the user's (possibly obfuscated or MD5 encrypted) password;

rolename

is a role of the user

For example:

```
admin: CRYPT:ad1ks..kc.1Ug,server-administrator,content-administrator,admin
```

```
other: OBF:1xmk1w26lu9rlwlclxmq
guest: guest,read-only
```

You configure the HashLoginService with a name and a reference to the location of the properties file:

```
<Item>
<New class="org.eclipse.jetty.security.HashLoginService">
  <Set name="name">Test Realm</Set>
  <Set name="config"><SystemProperty name="jetty.home" default=". "/>/etc/
realm.properties</Set>
</New>
</Item>
```

You can also configure it to check the properties file regularly for changes and reload when changes are detected. The reloadInterval is in seconds:

```
<New class="org.eclipse.jetty.security.HashLoginService">
  <Set name="name">Test Realm</Set>
  <Set name="config"><SystemProperty name="jetty.home" default=". "/>/etc/
realm.properties</Set>
  <Set name="reloadInterval">5</Set>
  <Call name="start"></Call>
</New>
```

JDBCLoginService

In this implementation, authentication and role information is stored in a database accessed via JDBC. A properties file defines the JDBC connection and database table information. Here is an example of a properties file for this realm implementation:

```
jdbcdriver = org.gjt.mm.mysql.Driver
url = jdbc:mysql://localhost/jetty
username = jetty
password = jetty
usertable = users
usertablekey = id
usertableuserfield = username
usertablepasswordfield = pwd
roletable = roles
roletablekey = id
roletablerolefield = role
userroletable = user_roles
userroletableuserkey = user_id
userroletablerolekey = role_id
cachetime = 300
```

The format of the database tables is (pseudo-sql):

```
users
(
  id integer PRIMARY KEY,
  username varchar(100) NOT NULL UNIQUE KEY,
  pwd varchar(50) NOT NULL
);
user_roles
```

```

(
    user_id integer NOT NULL,
    role_id integer NOT NULL,
    UNIQUE KEY (user_id, role_id),
    INDEX(user_id)
);
roles
(
    id integer PRIMARY KEY,
    role varchar(100) NOT NULL UNIQUE KEY
);

```

Where:

- **users** is a table containing one entry for every user consisting of:

id

the unique identity of a user

user

the name of the user

pwd

the user's password (possibly obfuscated or MD5 encrypted)

- **user-roles** is a table containing one row for every role granted to a user:

user_id

the unique identity of the user

role_id

the role for a user

- **roles** is a table containing one role for every role in the system:

id

the unique identifier of a role

role

a human-readable name for a role

If you want to use obfuscated, MD5 hashed or encrypted passwords the 'pwd' column of the 'users' table must be large enough to hold the obfuscated, hashed or encrypted password text plus the appropriate prefix.

You define a JDBCLoginService with the name of the realm and the location of the properties file describing the database:

```

<New class="org.eclipse.jetty.security.JDBCLoginService">
    <Set name="name">Test JDBC Realm</Set>
    <Set name="config">etc/jdbcRealm.properties</Set>
</New>

```

Authorization

As far as the [Servlet Specification](#) is concerned, authorization is based on roles. As we have seen, a LoginService associates a user with a set of roles. When a user requests a resource that is access protected, the LoginService will be asked to authenticate the user if they are not already, and then asked to confirm if that user possesses one of the roles permitted access to the resource.

Until Servlet 3.1, role-based authorization could define:

- access granted to a set of named roles
- access totally forbidden, regardless of role
- access granted to a user in any of the roles defined in the effective web.xml. This is indicated by the special value of "*" for the <role-name> of a <auth-constraint> in the <security-constraint>

With the advent of Servlet 3.1, there is now another authorization:

- access granted to any user who is authenticated, regardless of roles. This is indicated by the special value of "***" for the <role-name> of a <auth-constraint> in the <security-constraint>

Limiting Form Content

Form content sent to the server is processed by Jetty into a map of parameters to be used by the web application. This can be vulnerable to denial of service (DOS) attacks since significant memory and CPU can be consumed if a malicious clients sends very large form content or large number of form keys. Thus Jetty limits the amount of data and keys that can be in a form posted to Jetty.

The default maximum size Jetty permits is 200000 bytes and 1000 keys. You can change this default for a particular webapp or for all webapps on a particular Server instance.

Configuring Form Limits for a Webapp

To configure the form limits for a single webapplication, the context handler (or webappContext) instance must be configured using the following methods:

```
ContextHandler.setMaxFormContentSize(int maxSizeInBytes);
ContextHandler.setMaxFormKeys(int formKeys);
```

These methods may be called directly when embedding jetty, but more commonly are configured from a context XML file or WEB-INF/jetty-web.xml file:

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  ...
  <Set name="maxFormContentSize">200000</Set>
  <Set name="maxFormKeys">200</Set>
</Configure>
```

Configuring Form Limits for the Server

If a context does not have specific form limits configured, then the server attributes are inspected to see if a server wide limit has been set on the size or keys. The following XML shows how these attributes can be set in jetty.xml:

```
<configure class="org.eclipse.jetty.server.Server">
  ...
  <Call name="setAttribute">
    <Arg>org.eclipse.jetty.server.Request.maxFormContentSize</Arg>
    <Arg>100000</Arg>
  </Call>
<Call name="setAttribute">
```

```
<Arg>org.eclipse.jetty.server.Request.maxFormKeys</Arg>
<Arg>2000</Arg>
</Call>
</configure>
```

Aliased Files and Symbolic links

Web applications will often serve static content from the file system provided by the operating system running underneath the JVM. However because file systems often implement multiple aliased names for the same file, then security constraints and other servlet URI space mappings may inadvertently be bypassed by aliases.

A key example of this is case insensitivity and 8.3 names implemented by the Windows File system. If a file within a webapplication called /mysecretfile.txt is protected by a security constraint on the URI /mysecretfile.txt, then a request to /MySecretFile.TXT will not match the URI constraint because URIs are case sensitive, but the windows file system will report that a file does exist at that name and it will be served despite the security constraint. Less well known than case insensitivity is that windows file systems also support [8.3 filenames](#) for compatibility with legacy programs. Thus a request to a URI like /MYSECR~1.TXT will again not match the security constraint, but will be reported as an existing file by the file system and served.

There are many examples of aliases, not just on windows:

- NTFS Alternate stream names like c:\test\file.txt:\$DATA:name
- OpenVMS support file versioning so that /mysecret.txt;N refers to version N of /mysecret.txt and is essentially an alias.
- The clearcase software configuration management system provides a file system where @@ in a file name is an alias to a specific version.
- The unix file system supports ./foo.txt as an alias for /foo.txt
- Many JVM implementations incorrectly assume the null character is a string terminator, so that a file name resulting from /foobar.txt%00 is an alias for /foobar.txt
- Unix symbolic links and hard links are a form of aliases that allow the same file or directory to have multiple names.

In addition, it is not just URI security constraints that can be bypassed. For example the mapping of the URI pattern *.jsp to the JSP Servlet may be bypassed by an request to an alias like /foobar.jsp%00, thus rather than execute the JSP, the source code of the JSP is returned by the file system.

Good Security Practise

Part of the problem with aliases is that the standard web application security model is to allow all requests except the ones that are specifically denied by security constraints. A best practise for security is to deny all requests and to permit only those that are specifically identified as allowable. While it is possible to design web application security constraints in this style, it can be difficult in all circumstances and it is not the default. Thus it is important for Jetty to be able to detect and deny requests to aliased static content.

Alias detection

It is impossible for Jetty to know of all the aliases that may be implemented by the file system running beneath it, thus it does not attempt to make any specific checks for any known aliases. Instead jetty detects aliases by using the canonical path of a file. If a file resource handled by jetty has a canonical

name that differs from the name used to request the resource, then Jetty determines that the resource is an aliased request and it will not be returned by the `ServletContext.getResource(String)` method (or similar) and thus will not be served as static content nor used as the basis of a JSP.

This if Jetty is running on a windows operation system, then a file called `/MySecret.TXT` will have a canonical name that exactly matches that case. So while a request to `/mysecret.txt` or `/MYSECR~1.TXT` will result in a File Resource that matches the file, the different canonical name will indicate that those requests are aliases and they will not be served as static content and instead a 404 response returned.

Unfortunately this approach denies all aliases, including symbolic links, which can be useful in assembling complex web applications.

Serving Aliases and Symbolic Links

Not all aliases are bad nor should be seen as attempts to subvert security constraints. Specifically symbolic links can be very useful when assembling complex web applications, yet by default Jetty will not serve them. Thus Jetty contexts support an extensible AliasCheck mechanism to allow aliases resources to be inspected and conditionally served. In this way, "good" aliases can be detected and served. Jetty provides several utility implementations of the AliasCheck interface as nested classes with ContextHandler:

ApproveAliases

Approve all aliases (USE WITH CAUTION!).

AllowSymLinkAliasChecker

Approve Aliases using the `java-7 Files.readSymbolicLink(path)` and `Path.toRealPath(...)` APIs to check that alias are valid symbolic links.

An application is free to implement its own Alias checking. Alias Checkers can be installed in a context via the following XML used in a context deployer file or `WEB-INF/jetty-web.xml`:

```
<!-- Allow symbolic links -->
<Call name="addAliasCheck">
  <Arg><New class="org.eclipse.jetty.server.handler.AllowSymLinkAliasChecker"/></Arg>
</Call>
```

Secure Password Obfuscation

There are many places where you might want to use and store a password, for example for the SSL connectors and user passwords in realms.

Passwords can be stored in clear text, obfuscated, checksummed or encrypted in order of increasing security. The choice of method to secure a password depends on where you are using the password. In some cases such as keystore passwords and digest authentication, the system must retrieve the original password, which requires the obfuscation method. The drawback of the obfuscation algorithm is that it protects passwords from casual viewing only.

When the stored password is compared to one a user enters, the handling code can apply the same algorithm that secures the stored password to the user input and compare results, making password authentication more secure.

The class `org.eclipse.jetty.http.security.Password` can be used to generate all varieties of passwords.

Run it without arguments to see usage instructions:

```
$ export JETTY_VERSION=9.0.0-SNAPSHOT
$ java -cp lib/jetty-util-$JETTY_VERSION.jar org.eclipse.jetty.util.security.Password

Usage - java org.eclipse.jetty.util.security.Password [<user>] <password>
If the password is ?, the user will be prompted for the password
```

For example, to generate a secured version of the password "blah" for the user "me", do:

```
$ export JETTY_VERSION=9.0.0.RC0
$ java -cp lib/jetty-util-$JETTY_VERSION.jar org.eclipse.jetty.util.security.Password me
blah
blah
OBF:20771x1b206z
MD5:639bae9ac6b3ela84cebb7b403297b79
CRYPT:me/ks90E221EY
```

You can now cut and paste whichever secure version you choose into your configuration file or java code.

For example, the last line below shows how you would cut and paste the encrypted password generated above into the properties file for a LoginService:

```
admin: CRYPT:ad1ks..kc.1Ug,server-administrator,content-administrator,admin
other: OBF:1xmk1w261u9r1wlclxmq
guest: guest,read-only
me:CRYPT:me/ks90E221EY
```



Tip

Don't forget to also copy the OBF:, MD5: or CRYPT: prefix on the generated password. It will not be usable by Jetty without it.

You can also use obfuscated passwords in jetty xml files where a plain text password is usually needed. Here's an example setting the password for a JDBC Datasource with obfuscation:

```
<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
  <Arg></Arg>
  <Arg>jdbc/DSTest</Arg>
  <Arg>
    <New class="com.jolbox.bonecp.BoneCPDataSource">
      <Set name="driverClass">com.mysql.jdbc.Driver</Set>
      <Set name="jdbcUrl">jdbc:mysql://localhost:3306/foo</Set>
      <Set name="username">dbuser</Set>
      <Set name="password">
        <Call class="org.eclipse.jetty.util.security.Password" name="deobfuscate">
          <Arg>OBF:1ri71v1rlv2n1ri71shqlri71shs1ri71v1rlv2n1ri7</Arg>
        </Call>
      </Set>
      <Set name="minConnectionsPerPartition">5</Set>
      <Set name="maxConnectionsPerPartition">50</Set>
      <Set name="acquireIncrement">5</Set>
      <Set name="idleConnectionTestPeriod">30</Set>
    </New>
  </Arg>
</New>
```

JAAS Support

JAAS implements a Java version of the standard Pluggable Authentication Module (PAM) framework.

JAAS can be used for two purposes:

- for authentication of users, to reliably and securely determine who is currently executing Java code, regardless of whether the code is running as an application, an applet, a bean, or a servlet; and
- for authorization of users to ensure they have the access control rights (permissions) required to do the actions performed.

JAAS authentication is performed in a pluggable fashion. This permits applications to remain independent from underlying authentication technologies. New or updated authentication technologies can be plugged under an application without requiring modifications to the application itself. Applications enable the authentication process by instantiating a LoginContext object, which in turn references a Configuration to determine the authentication technology(ies), or LoginModule(s), to be used in performing the authentication. Typical LoginModules may prompt for and verify a username and password. Others may read and verify a voice or fingerprint sample.

See Java Authentication and Authorization Service (JAAS) [Reference Guide](#) for more information about JAAS.

Jetty and JAAS

Many application servers support JAAS as a means of bringing greater flexibility to the declarative security models of the J2EE (now known as the JavaEE) [specification](#). Jetty support for JAAS provides greater alternatives for servlet security, and increases the portability of web applications.

The JAAS support aims to dictate as little as possible whilst providing a sufficiently flexible infrastructure to allow users to drop in their own custom [LoginModules](#).

Configuration

Using JAAS with jetty is very simply a matter of declaring a `org.eclipse.jetty.jaas.JAASLoginService`, creating a jaas login module configuration file and specifying it on the jetty run line. Let's look at an example.

Step 1

Configure a jetty `org.eclipse.jetty.jaas.JAASLoginService` to match the <realm-name> in your web.xml file. For example, if the web.xml contains a realm called "xyz" like so:

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>xyz</realm-name>
  <form-login-config>
    <form-login-page>/login/login</form-login-page>
    <form-error-page>/login/error</form-error-page>
  </form-login-config>
</login-config>
```

Then you need to create a JAASLoginService with the matching name of "xyz":

```
<New class="org.eclipse.jetty.jaas.JAASLoginService">
  <Set name="Name">Test JAAS Realm</Set>
  <Set name="LoginModuleName">xyz</Set>
</New>
```

Caution



The name of the realm-name that you declare in `web.xml` must match exactly the name of your JAASLoginService.

You can declare your JAASLoginService in a couple of different ways:

1. If you have more than one webapp that you would like to use the same security infrastructure, then you can declare your JAASLoginService in a top-level jetty xml file as a bean that is added to the `org.eclipse.jetty.server.Server`. Here's an example:

```
<Configure id="Server" class="org.eclipse.jetty.server.Server">

    <Call name="addBean">
        <Arg>
            <New class="org.eclipse.jetty.jaas.JAASLoginService">
                <Set name="name">Test JAAS Realm</Set>
                <Set name="LoginModuleName">xyz</Set>
            </New>
        </Arg>
    </Call>

</Configure>
```

2. Alternatively, you can use a JAASLoginService with just a specific webapp by creating a [context xml](#) file for the webapp, and specifying the JAASLoginService in it:

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">

    <Set name="securityHandler">
        <New class="org.eclipse.jetty.security.ConstraintSecurityHandler">
            <Set name="loginService">
                <New class="org.eclipse.jetty.jaas.JAASLoginService">
                    <Set name="name">Test JAAS Realm</Set>
                    <Set name="loginModuleName">xyz</Set>
                </New>
            </Set>
        </New>
    </Set>
</Configure>
```

Step 2

Set up your LoginModule in a configuration file, following the [syntax rules](#) :

```
xyz {
    com.acme.SomeLoginModule required debug=true;
};
```

Caution



It is imperative that the application name on the first line is exactly the same as the `LoginModuleName` of your JAASLoginService.

You may find it convenient to name this configuration file as `etc/login.conf` because, as we will see below, some of the wiring up for jaas has been done for you.

Step 3

You now need to invoke jetty with support for jaas. There are 2 aspects to this:

- adding jaas-related jars to the jetty container classpath
- setting the System property `java.security.auth.login.config`

To accomplish the above, use the jetty [startup modules mechanism](#) to add the jaas module:

```
java -jar start.jar --add-to-startd=jaas
```



Note

The top level of the distribution does not have the jaas module enabled by default. However, there are several [demo webapps](#) - including a jaas webapp - available in the [demo-base](#) directory of the distribution which has pre-enabled the jaas module.

Now you will have a file named `start.d/jaas.ini`, which contains:

```
--module=jaas
jaas.login.conf=etc/login.conf
```

The `jaas.login.conf` property refers to the location of your LoginModule configuration file that you established in [Step 2](#). If you called it `etc/login.conf`, then your work is done. Otherwise, change the value of the `jaas.login.conf` property to be the location of your LoginModule configuration file. Jetty will automatically use this property to set the value of the System property `java.security.auth.login.config`.

A Closer Look at JAASLoginService

To allow the greatest degree of flexibility in using JAAS with web applications, the JAASLoginService supports a couple of configuration options. Note that you don't ordinarily need to set these explicitly, as jetty has defaults which will work in 99% of cases. However, should you need to, you can configure:

- a policy for role-based authorization (Default: [org.eclipse.jetty.jaas.StrictRoleCheckPolicy](#))
- a CallbackHandler (Default: [org.eclipse.jetty.jaas.callback.DefaultCallbackHandler](#))
- a list ofclassnames for the Principal implementation that equate to a user role (Default: [org.eclipse.jetty.jaas.JAASRole](#))

Here's an example of setting each of these (to their default values):

```
<New class="org.eclipse.jetty.jaas.JAASLoginService">
  <Set name="Name">Test JAAS Realm</Set>
  <Set name="LoginModuleName">xyz</Set>
  <Set name="RoleCheckPolicy">
    <New class="org.eclipse.jetty.jaas.StrictRoleCheckPolicy"/>
  </Set>
  <Set name="CallbackHandlerClass">
    org.eclipse.jetty.jaas.callback.DefaultCallbackHandler
  </Set>
  <Set name="roleClassNames">
    <Array type="java.lang.String">
      <Item>org.eclipse.jetty.jaas.JAASRole</Item>
    </Array>
  </Set>
</New>
```

RoleCheckPolicy

The RoleCheckPolicy must be an implementation of the [org.eclipse.jetty.jaas.RoleCheckPolicy](#) interface and its purpose is to help answer the question "is User X in Role Y" for role-based authorization requests. The default implementation distributed with jetty is the [org.eclipse.jetty.jaas.StrictRoleCheckPolicy](#), which will assess a user as having a particular role iff that role is at the top of the stack of roles that have been

temporarily pushed onto the user or if the user has no temporarily assigned roles, the role is amongst those configured for the user.

Roles can be temporarily assigned to a user programmatically by using the pushRole(String rolename) method of the [org.eclipse.jetty.jaas.JAASUserPrincipal](#) class.

For the majority of webapps, the default StrictRoleCheckPolicy will be quite adequate, however you may provide your own implementation and set it on your JAASLoginService instance.

CallbackHandler

A CallbackHandler is responsible for interfacing with the user to obtain usernames and credentials to be authenticated.

Jetty ships with the [org.eclipse.jetty.jaas.DefaultCallbackHandler](#) which interfaces the information contained in the request to the Callbacks that are requested by LoginModules. You can replace this default with your own implementation if you have specific requirements not covered by the default.

Role Principal Implementation Class

When LoginModules authenticate a user, they usually also gather all of the roles that a user has and place them inside the JAAS Subject. As LoginModules are free to use their own implementation of the JAAS Principal to put into the Subject, jetty needs to know which Principals represent the user and which represent his/her roles when performing authorization checks on <security-constraint>s. The example LoginModules that ship with jetty all use the [org.eclipse.jetty.jaas.JAASRole](#) class. However, if you have plugged in some other LoginModules, you must configure theclassnames of their role Principal implementations.

Sample LoginModules

- [org.eclipse.jetty.jaas.spi.JDBCLoginModule](#)
- [org.eclipse.jetty.jaas.spi.PropertyFileLoginModule](#)
- [org.eclipse.jetty.jaas.spi.DataSourceLoginModule](#)
- [org.eclipse.jetty.jaas.ldap.LdapLoginModule](#)



Passwords/Credentials

Passwords can be stored in clear text, obfuscated or checksummed. The class [org.eclipse.util.security.Password](#) should be used to generate all varieties of passwords, the output from which can be cut and pasted into property files or entered into database tables.

See more on this under the Configuration section on [securing passwords](#).

JDBCLoginModule

The JDBCLoginModule stores user passwords and roles in a database that are accessed via JDBC calls. You can configure the JDBC connection information, as well as the names of the table and columns storing the username and credential, and the name of the table and columns storing the roles.

Here is an example login module configuration file entry for it using an HSQLDB driver:

```
jdbc {  
    org.eclipse.jetty.jaas.spi.JDBCLoginModule required  
    debug="true"
```

```

dbUrl="jdbc:hsqldb:."
dbUserName="sa"
dbDriver="org.hsqldb.jdbcDriver"
userTable="myusers"
userField="myuser"
credentialField="mypassword"
userRoleTable="myuserroles"
userRoleUserField="myuser"
userRoleRoleField="myrole";
};

```

There is no particular schema required for the database tables storing the authentication and role information. The properties userTable, userField, credentialField, userRoleTable, userRoleUserField, userRoleRoleField configure the names of the tables and the columns within them that are used to format the following queries:

- select <credentialField> from <userTable> where <userField> =?
- select <userRoleRoleField> from <userRoleTable> where <userRoleUserField> =?

Credential and role information is lazily read from the database when a previously unauthenticated user requests authentication. Note that this information is only cached for the length of the authenticated session. When the user logs out or the session expires, the information is flushed from memory.

Note that passwords can be stored in the database in plain text or encoded formats - see "Passwords/Credentials" note above.

DataSourceLoginModule

Similar to the JDBCLoginModule, but this LoginModule uses a DataSource to connect to the database instead of a jdbc driver. The DataSource is obtained by doing a jndi lookup on `java:comp/env/$ {dnJNDIName}`

Here is a sample login module configuration for it:

```

ds {
    org.eclipse.jetty.jaas.spi.DataSourceLoginModule required
    debug="true"
    dbJNDIName="ds"
    userTable="myusers"
    userField="myuser"
    credentialField="mypassword"
    userRoleTable="myuserroles"
    userRoleUserField="myuser"
    userRoleRoleField="myrole";
};

```

PropertyFileLoginModule

With this login module implementation, the authentication and role information is read from a property file.

```

props {
    org.eclipse.jetty.jaas.spi.PropertyFileLoginModule required
    debug="true"
    file="/somewhere/somefile.props";
};

```

The file parameter is the location of a properties file of the same format as the etc/realm.properties example file. The format is:

```
<username>: <password>[,<rolename> ...]
```

Here's an example:

```
fred: OBF:1xmk1w261u9r1wlclxmq,user,admin
harry: changeme,user,developer
tom: MD5:164c88b302622e17050af52c89945d44,user
dick: CRYPT:adpxezg3FUZAk,admin
```

The contents of the file are fully read in and cached in memory the first time a user requests authentication.

LdapLoginModule

Here's an example:

```
ldaploginmodule {
    org.eclipse.jetty.jaas.spi.LdapLoginModule required
    debug="true"
    contextFactory="com.sun.jndi.ldap.LdapCtxFactory"
    hostname="ldap.example.com"
    port="389"
    bindDn="cn=Directory Manager"
    bindPassword="directory"
    authenticationMethod="simple"
    forceBindingLogin="false"
    userBaseDn="ou=people,dc=alcatel"
    userRdnAttribute="uid"
    userIdAttribute="uid"
    userPasswordAttribute="userPassword"
    userObjectClass="inetOrgPerson"
    roleBaseDn="ou=groups,dc=example,dc=com"
    roleNameAttribute="cn"
    roleMemberAttribute="uniqueMember"
    roleObjectClass="groupOfUniqueNames";
};
```

Writing your Own LoginModule

If you want to implement your own custom LoginModule, there are two classes to be familiar with [org.eclipse.jetty.jaas.spi.AbstractLoginModule](#) and [org.eclipse.jetty.jaas.spi.UserInfo](#).

The [org.eclipse.jetty.jaas.spi.AbstractLoginModule](#) implements all of the [javax.security.auth.spi.LoginModule](#) methods. All you need to do is to implement the `getUserInfo` method to return a [org.eclipse.jetty.jaas.UserInfo](#) instance which encapsulates the username, password and role names (note: as `java.lang.Strings`) for a user.

The `AbstractLoginModule` does not support any caching, so if you want to cache `Userinfo` (eg as does the [org.eclipse.jetty.jaas.spi.PropertyFileLoginModule](#)) then you must provide this yourself.

Other Goodies

RequestParameterCallback

As all servlet containers intercept and process a form submission with action `j_security_check`, it is usually not possible to insert any extra input fields onto a login form with

which to perform authentication: you may only pass `j_username` and `j_password`. For those rare occasions when this is not good enough, and you require more information from the user in order to authenticate them, you can use the JAAS callback handler `org.eclipse.jetty.jaas.callback.RequestParamerCallback`. This callback handler gives you access to all parameters that were passed in the form submission. To use it, in the `login()` method of your custom login module, add the `RequestParamerCallback` to the list of callback handlers the login module uses, tell it which params you are interested in, and then get the value of the parameter back. Here's an example:

```
public class FooLoginModule extends AbstractLoginModule
{

    public boolean login()
        throws LoginException
    {

        Callback[] callbacks = new Callback[3];
        callbacks[0] = new NameCallback();
        callbacks[1] = new ObjectCallback();

        //as an example, look for a param named "extrainfo" in the request
        //use one RequestParameterCallback() instance for each param you want to access
        callbacks[2] = new RequestParameterCallback ();
        ((RequestParameterCallback)callbacks[2]).setParameterName ("extrainfo");

        callbackHandler.handle(callbacks);
        String userName = ((NameCallback)callbacks[0]).getName();
        Object pwd = ((ObjectCallback)callbacks[1]).getObject();
        List paramValues = ((RequestParameterCallback)callbacks[2]).getParameterValues();

        //use the userName, pwd and the value(s) of the parameter named "extrainfo" to
        //authenticate the user
    }
}
```

Example JAAS WebApp

An example webapp using jaas can be found in our git repo:

- <http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/tree/tests/test-webapps/test-jaas-webapp>

Spnego Support

Spnego or Simple and Protected GSSAPI Negotiation Mechanism is a way for users to be seamlessly authenticated when running on a Windows or Active Directory based network. Jetty supports this type of authentication and authorization through the JDK so you must be using a JDK that supports it, which recent versions of Java 6 and 7 do. Also important to note is that this is an incredibly fragile setup where everything needs to be configured just right for things to work, otherwise it can fail in fun and exciting, not to mention obscure ways.

There is a substantial amount of configuration and testing required to enable this feature as well as knowledge and access to central systems on a Windows network such as the Active Domain Controller and the ability to create and maintain service users.

Configuring Jetty and Spnego

To run with spnego enabled the following command line options are required:

```
-Djava.security.krb5.conf=/path/to/jetty/etc/krb5.ini \
-Djava.security.auth.login.config=/path/to/jetty/etc/spnego.conf \
-Djavax.security.auth.useSubjectCredsOnly=false
```

For debugging the spnego authentication the following options are very helpful:

```
-Dorg.eclipse.jetty.LEVEL=debug \
-Dsun.security.spnego.debug=all
```

Spnego Authentication must be enabled in the webapp in the following way. The name of the role will be different for your network.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Secure Area</web-resource-name>
    <url-pattern>/secure/me/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <!-- this is the domain that the user is a member of -->
    <role-name>MORTBAY.ORG</role-name>
  </auth-constraint>
</security-constraint>
<login-config>
  <auth-method>SPNEGO</auth-method>
  <realm-name>Test Realm</realm-name>
  <!-- optionally to add custom error page -->
  <spnego-login-config>
    <spnego-error-page>/loginError.html?param=foo</spnego-error-page>
  </spnego-login-config>
</login-config>
```

A corresponding UserRealm needs to be created either programmatically if embedded, via the jetty.xml or in a context file for the webapp.

This is what the configuration within a jetty xml file would look like.

```
<Call name="addBean">
  <Arg>
    <New class="org.eclipse.jetty.security.SpnegoLoginService">
      <Set name="name">Test Realm</Set>
      <Set name="config"><Property name="jetty.home" default=". "/>/etc/
      spnego.properties</Set>
    </New>
  </Arg>
</Call>
```

This is what the configuration within a context xml file would look like.

```
<Get name="securityHandler">
  <Set name="loginService">
    <New class="org.eclipse.jetty.security.SpnegoLoginService">
      <Set name="name">Test Realm</Set>
      <Set name="config">
        <SystemProperty name="jetty.home" default=". "/>/etc/spnego.properties
      </Set>
    </New>
  </Set>
</Get>
```

```
</Set>
</New>
</Set>
<Set name="checkWelcomeFiles">true</Set>
</Get>
```

There are a number of important configuration files with spnego that are required. The default values for these configuration files from this test example are found in the jetty-distribution.

spnego.properties

configures the user realm with runtime properties

krb5.ini

configures the underlying kerberos setup

spnego.conf

configures the glue between gssapi and kerberos

It is important to note that the keytab file referenced in the krb5.ini and the spnego.conf files needs to contain the keytab for the targetName for the http server. To do this use a process similar to this:

On the windows active domain controller run:

```
$ setspn -A HTTP/linux.mortbay.org ADUser
```

To create the keytab file use the following process:

```
$ ktpass -out c:\dir\krb5.keytab -princ HTTP/linux.mortbay.org@MORTBAY.ORG -mapUser
ADUser -mapOp set -pass ADUserPWD -crypto RC4-HMAC-NT -pType KRB5_NT_PRINCIPAL
```

This step should give you the keytab file which should then be copied over to the machine running this http server and referenced from the configuration files. For our testing we put the keytab into the etc directory of jetty and referenced it from there.

Configuring Firefox

The follows steps have been required to inform Firefox that it should use a negotiation dialog to authenticate.

1. browse to about:config and agree to the warnings
2. search through to find the 'network' settings
3. set network.negotiate-auth.delegation-uris to http://,https://
4. set network.negotiate-auth.trusted-uris to http://,https://

Configuring Internet Explorer

The follows steps have been required to inform Internet Explorer that it should use a negotiation dialog to authenticate.

1. Tools -> Options -> Security -> Local Intranet -> Sites (everything should be checked here)

2. Tools -> Options -> Security -> Local Intranet -> Sites -> Advanced (add url to server (http:// and/ or https:// use the hostname!)
3. Tools -> Options -> Security -> Local Intranet -> Sites -> Advanced -> Close
4. Tools -> Options -> Security -> Local Intranet -> Sites -> Ok
5. Tools -> Options -> Advanced -> Security (in the checkbox list)
6. locate and check 'Enable Integrated Windows Authentication'
7. Tools -> Options -> Advanced -> Security -> Ok
8. close IE then reopen and browse to your spengo protected resource



Note

Note: You must go to the hostname and not the IP, if you go to the IP it will default to NTLM authentication...the following conditions must be true for Spnego authentication to work.

- You must be within the Intranet Zone of the network
- Accessing the server using a Hostname rather than IP
- Integrated Windows Authentication in IE is enabled and the host is trusted in Firefox
- The server is not local to the browser, it can't be running on localhost.
- The client's Kerberos system is authenticated to a domain controller

Chapter 8. Configuring JSP Support

Table of Contents

Configuring JSP 103

Configuring JSP

This document provides information about configuring Java Server Pages for Jetty.

Which JSP Implementation

From jetty-9.2 onwards, we are using Jasper from [Apache](#) as the default JSP container implementation. In previous versions we used Jasper from [Glassfish](#), and if you wish to, you can continue to do so.

The jetty-distribution by default enables the jsp [module](#), and by default, this [module](#) is set to Apache Jasper. To change to use Glassfish Jasper instead, edit the `$JETTY_HOME/start.d/jsp.mod` file and change the line indicated:

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/jetty-distribution/  
src/main/resources/modules/jsp.mod
```

Note that the availability of some JSP features may depend on which JSP container implementation you are using. Note also that it may not be possible to precompile your jsps with one container and deploy to the other.

Precompiling JSRs

You can either follow the instructions on precompilation provided by the JSP container of your choice (either [Glassfish](#) or [Apache](#)), or if you are using maven for your builds, you can use the [jetty-jspc-maven](#) plugin to do it for you.

If you have precompiled your jsps, and have customized the output package prefix (which is `org.apache.jsp` by default), you should configure your webapp context to tell Jetty about this custom package name. You can do this using a servlet context init-param called `org.eclipse.jetty.servlet.jspPackagePrefix`.

For example, suppose you have precompiled your jsps with the custom package prefix of `com.acme`, then you would add the following lines to your `web.xml` file:

```
<context-param>  
  <param-name>org.eclipse.jetty.servlet.jspPackagePrefix</param-name>  
  <param-value>com.acme</param-value>  
</context-param>
```



Note

both jetty maven plugins - [jetty-jspc-maven-plugin](#) and the [jetty-maven-plugin](#) - will only use Apache Jasper

Runtime Compiling JSRs

Depending on which JSP container you elect to use, the configuration and compilation characteristics will be different.

Apache JSP Container

By default, the Apache JSP container will look for the Eclipse Java Compiler (jdt). The jetty distribution ships a copy of this in \$JETTY_HOME/lib/apache-jsp. If you wish to use a different compiler, you will need to configure the `compilerClassName` init-param on the `JspServlet` with the name of the class.

Table 8.1. Understanding Apache JspServlet Parameters

init param	Description	Default	webdefault.xml
classpath	Classpath used for jsp compilation. Only used if org.apache.catalina.jsp_classpath context attribute is not set, which it is in Jetty.	-	-
classdebuginfo	Include debugging info in class file.	TRUE	-
checkInterval	Interval in seconds between background recompile checks. Only relevant if development=false.	0	-
development	development=true, recompilation checks occur on each request. See also modification-TestInterval.	TRUE	-
displaySourceFragment	Should a source fragment be included in exception messages	TRUE	-
errorOnUseBeanInvalidClassAttribute	Should Jasper issue an error when the value of the class attribute in an useBean action is not a valid bean class	TRUE	-
fork	Should Ant fork its Java compiles of JSP pages?	TRUE	FALSE
keepgenerated	Do you want to keep the generated Java files around?	TRUE	-
trimSpaces	Should white spaces between directives or actions be trimmed?	FALSE	-
enablePooling	Determines whether tag handler pooling is enabled.	TRUE	-
engineOptionsClass	Allows specifying the Options class used to configure Jasper. If	-	

init param	Description	Default	webdefault.xml
	not present, the default EmbeddedServletOptions will be used.		
mappedFile	Support for mapped Files. Generates a servlet that has a print statement per line of the JSP file	TRUE	-
suppressSmap	Generation of SMAP info for JSR45 debugging.	FALSE	-
dumpSmap	Dump SMAP JSR45 info to a file.	FALSE	-
genStrAsCharArray	Option for generating Strings.	FALSE	-
ieClassId	The class-id value to be sent to Internet Explorer when using <jsp:plugin> tags.	clsid:8AD9C840-044E-14D1-B3E9-00805F499D93	
maxLoadedJsp	The maximum number of JSPs that will be loaded for a web application. If more than this number of JSPs are loaded, the least recently used JSPs will be unloaded so that the number of JSPs loaded at any one time does not exceed this limit. A value of zero or less indicates no limit.	-1	-
jspIdleTimeout	The amount of time in seconds a JSP can be idle before it is unloaded. A value of zero or less indicates never unload.	-1	-
scratchDir	Directory where servlets are generated. See	-	-
compilerClassName	If not set, defaults to the Eclipse jdt compiler.	-	
compiler	Used if the Eclipse jdt compiler cannot be found on the classpath. It is the classname of a compiler that Ant should invoke.	-	-
compilerTargetVM	Target vm to compile for.	1.7	-

init param	Description	Default	webdefault.xml
compilerSourceVM	Sets source compliance level for the jdt compiler.	1.7	—
javaEncoding	Pass through the encoding to use for the compilation.	UTF8	—
modificationTestInterval	If development=true, interval between recompilation checks, triggered by a request.	4	—
xpoweredBy	Generate an X-Powered-By response header.	FALSE	FALSE
recompileOnFail	If a JSP compilation fails should the modificationTestInterval be ignored and the next access trigger a re-compilation attempt? Used in development mode only and is disabled by default as compilation may be expensive and could lead to excessive resource usage.	-	—

Glassfish JSP Container

To compile .jsp files into Java classes, you need a Java compiler. You can acquire a Java compiler from the JVM if you are using a full JDK, or from a third party Jar.

By default, the Glassfish JSP container tries to use the compiler that is part of the JDK. **NOTE:** when using the JDK compiler, the system does *not* save your class files to disk unless you use the saveBytecode init-param as described below.

If you do not have a full JDK, you can configure the Eclipse Java Compiler that Jetty ships in the distro in \$JETTY_HOME/lib/jsp/. You need to define a SystemProperty that prevents the Glassfish JSP engine from defaulting to the in-JVM compiler.

This is best done when using the standalone distro by uncommenting the System property org.apache.jasper.compiler.disablejsr199 in the jsp module:

```
-Dorg.apache.jasper.compiler.disablejsr199=true
```

Or for embedded usages, simply define this as a normal System property.

Configuration

The JSP engine has many configuration parameters. Some parameters affect only precompilation, and some affect runtime recompilation checking. Parameters also differ among the various versions of the JSP engine. This page lists the configuration parameters, their meanings, and their default

settings. Set all parameters on the `org.apache.jasper.JspServlet` instance defined in the `webdefault.xml` file.



Note

Be careful: for all of these parameters, if the value you set doesn't take effect, try using all lower case instead of camel case, or capitalizing only some of the words in the name, as JSP is inconsistent in its parameter naming strategy.

Table 8.2. Understanding Glassfish JSP Parameters

init param	Description	Default	webdefault.xml
development	development=true, recompilation checks occur on each request. See also modification-TestInterval.	TRUE	—
fork	Should Ant fork its Java compiles of JSP pages?	TRUE	FALSE
keepgenerated	Do you want to keep the generated Java files around?	FALSE	—
saveBytecode	If class files are generated as byte arrays, should they be saved to disk at the end of compilations?	FALSE	—
trimSpaces	Should white spaces between directives or actions be trimmed?	FALSE	—
enablePooling	Determines whether tag handler pooling is enabled.	TRUE	—
mappedFile	Support for mapped Files. Generates a servlet that has a print statement per line of the JSP file./	TRUE	—
sendErrorToClient	If false, stack traces, etc., are sent to std error instead of the client's browser.	FALSE	—
classdebuginfo	Include debugging info in class file.	TRUE	—
checkInterval	Interval in seconds between background recompile checks. Only relevant if development=false.	0	—
suppressSmap	Generation of SMAP info for JSR45 debugging.	FALSE	—
dumpSmap	Dump SMAP JSR45 info to a file.	FALSE	—

init param	Description	Default	webdefault.xml
genStrAsCharArray	Option for generating Strings.	FALSE	–
genStrAsByteArray	Option for generating Strings.	TRUE	–
defaultBufferNone		FALSE	–
errorOnUseBeanInvalidClassAttribute		FALSE	–
scratchDir	Directory where servlets are generated. Jetty sets this value according to the [/display/JETTY/Temporary+Directories work dir] settings for the webapp.	–	–
compiler	Determined at runtime. For Jetty this is the Eclipse jdt compiler.	–	–
compilerTargetVM	Target vm to compile for.	1.5	–
compilerSourceVM	Sets source compliance level for the jdt compiler.	1.5	–
javaEncoding	Pass through the encoding to use for the compilation.	UTF8	–
modificationTestInterval	If development=true, interval between recompilation checks, triggered by a request.	0	–
xpoweredBy	Generate an X-Powered-By response header.	FALSE	FALSE
usePrecompiled/use-precompiled		FALSE	–
validating/enableTld-Validation	Whether or not to validate tag files against the schema.	FALSE	–
reload-interval	If reload-interval=0, then no runtime checking of JSP, otherwise sets the checking interval for both development=true and development=false.	–	–
initial-capacity/initial-Capacity	The initial capacity of the hash maps mapping the name of the JSP to class and JSP file.	–	–

Much confusion generally ensues about the development, checkInterval and modificationTestInterval parameters and JSP runtime recompilation. Here is a factoring out of the various options:

- Check the JSP files for possible recompilation on every request:

```
<init-param>
    <param-name>development</param-name>
    <param-value>true</param-value>
</init-param>
```

- Only check approximately every N seconds, where a request triggers the time-lapse calculation. This example checks every 60 seconds:

```
<init-param>
    <param-name>development</param-name>
    <param-value>true</param-value>
</init-param>
<init-param>
    <param-name>modificationTestInterval</param-name>
    <param-value>60</param-value>
</init-param>
```

- Do no checking whatsoever, but still compile the JSP on the very first hit. (Be aware that this "reload-interval" parameter is shorthand for a "development=false" and "checkInterval=0" combination.):

```
<init-param>
    <param-name>reload-interval</param-name>
    <param-value>-1</param-value>
</init-param>
```

- Don't do any request-time checking, but instead start a background thread to do checks every N seconds. This example checks every 60 seconds:

```
<init-param>
    <param-name>development</param-name>
    <param-value>false</param-value>
</init-param>
<init-param>
    <param-name>checkInterval</param-name>
    <param-value>60</param-value>
</init-param>
```

Modifying Configuration

Regardless of which JSP container you are using, there are several options for modifying the JspServlet configuration.

Overriding webdefault.xml

You can make a copy of the [\\$JETTY_HOME/etc/webdefault.xml](#) that ships with Jetty, apply your changes, and use it instead of the shipped version. The example below shows how to do this when using the Jetty Maven plugin.

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <configuration>
    <webApp>
      <defaultsDescriptor>src/main/resources/webdefault.xml</defaultsDescriptor>
    </webApp>
  </configuration>
</plugin>
```

If you are using the Jetty distro, and you want to change the JSP settings for just one or a few of your webapps, copy the [\\$JETTY_HOME/etc/webdefault.xml](#) file somewhere, modify it, and then use a [context xml](#) file to set this file as the defaultsDescriptor for your webapp. Here's a snippet:

```
<Configure class=>"org.eclipse.jetty.webapp.WebAppContext">

  <Set name=>"contextPath">/foo</Set>
  <Set name=>"war"><SystemProperty name=>"jetty.home" >default=>". . ."/>/webapps/
foobar.war</Set>
  <Set name=>"defaultsDescriptor">/home smith/dev/webdefault.xml</Set>

</Configure>
```

If you want to change the JSP settings for all webapps, edit the [\\$JETTY_HOME/etc/webdefaults.xml](#) file directly instead.

Configuring the JSP Servlet in web.xml

Another option is to add an entry for the JSPServlet to the [WEB-INF/web.xml](#) file of your webapp and change or add init-params. You may also add (but not remove) servlet-mappings. You can use the entry in [\\$JETTY_HOME/etc/webdefault.xml](#) as a starting point.

```
<servlet id=>"jsp">
  <servlet-name>jsp</servlet-name>
  <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
  <init-param>
    <param-name>logVerbosityLevel</param-name>
    <param-value>DEBUG</param-value>
  </init-param>
  <init-param>
    <param-name>fork</param-name>
    <param-value>>false</param-value>
  </init-param>
  <init-param>
    <param-name>keepgenerated</param-name>
    <param-value>>true</param-value>
  </init-param>
  ...
  <load-on-startup>0</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>jsp</servlet-name>
  <url-pattern>*.jsp</url-pattern>
```

```
<url-pattern>*.jspx</url-pattern>
<url-pattern>*.xsp</url-pattern>
<url-pattern>*.xsp</url-pattern>
<url-pattern>*.JSP</url-pattern>
<url-pattern>*.JSF</url-pattern>
<url-pattern>*.JSF</url-pattern>
<url-pattern>*.XSP</url-pattern>
</servlet-mapping>

<servlet id=>"my-servlet">
  <servlet-name>myServlet</servlet-name>
  <servlet-class>com.acme.servlet.MyServlet</servlet-class>
  ...

```

Using JSTL Taglibs

The JavaServer Pages Standard Tag Library (JSTL) is part of the Jetty distribution (in \$JETTY_HOME/lib/jsp) and is automatically on the classpath.

Using JSF Taglibs

The following sections provide information about using JSF taglibs with Jetty Standalone and the Jetty Maven Plugin.

Using JSF Taglibs with Jetty Distribution

If you want to use JSF with your webapp, you need to copy the JSF implementation Jar (whichever Jar contains the META-INF/*.tld files from your chosen JSF implementation) into Jetty's shared container lib directory. You can either put them into the lib directory matching your JSP container of choice (either \$JETTY_HOME/lib/jsp for Glassfish JSP, or \$JETTY_HOME/lib/apache-jsp for Apache JSP), or put them into \$JETTY_HOME/lib/ext.

Using JSF Taglibs with Jetty Maven Plugin

You should make your JSF jars dependencies of the plugin and *not* the webapp itself. For example:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <configuration>
    <webApp>
      <contextPath>/${artifactId}</contextPath>
    </webApp>
    <scanIntervalSeconds>5</scanIntervalSeconds>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>com.sun.faces</groupId>
      <artifactId>jsf-api</artifactId>
      <version>2.0.8</version>
    </dependency>
    <dependency>
      <groupId>com.sun.faces</groupId>
      <artifactId>jsf-impl</artifactId>
      <version>2.0.8</version>
    </dependency>
  </dependencies>
</plugin>
```

Part III. Jetty Administration Guide

Table of Contents

9. Starting Jetty	115
Startup Overview	115
Managing XML Based Startup Configuration	118
Managing Server Classpath	118
Managing Startup Modules	120
Managing Jetty Base and Jetty Home	124
Using start.jar	128
Startup a Unix Service using jetty.sh	131
Startup via Windows Service	135
10. Session Management	141
Setting Session Characteristics	141
Using Persistent Sessions	144
Session Clustering with a Database	146
Session Clustering with MongoDB	149
11. Configuring JNDI	154
Quick Setup	154
Detailed Setup	154
Working with Jetty JNDI	156
Configuring JNDI	159
Using JNDI with Jetty Embedded	163
Datasource Examples	165
12. Annotations	172
Quick Setup	172
Working with Annotations	173
Using Annotations with Jetty Embedded	175
13. JMX	179
Using Java Management Extensions (JMX)	179
Jetty JConsole	182
Jetty JMX Annotations	184
14. SPDY	187
Introducing SPDY	187
Configuring SPDY	187
Configuring SPDY Proxy	188
Configuring SPDY push	194
Implement a custom SPDY PushStrategy	197
15. ALPN	199
.....	199
16. NPN	203
.....	203
17. FastCGI Support	208
FastCGI Introduction	208
Configuring Jetty for FastCGI	208
18. Bundled Servlets, Filters, and Handlers	212
Default Servlet	212
Proxy Servlet	213
Balancer Servlet	214
CGI Servlet	215
Quality of Service Filter	215
Denial of Service Filter	218
Gzip Filter	220
Cross Origin Filter	222
Resource Handler	223
Debug Handler	225
Statistics Handler	226
IP Access Handler	227

Moved Context Handler	229
Shutdown Handler	230
Default Handler	231
Error Handler	231
19. Jetty Runner	233
Use Jetty without an installed distribution	233
20. Setuid	239
Configuring Setuid	239
21. Optimizing Jetty	241
Garbage Collection	241
High Load	242
Limiting Load	244
22. Jetty Logging	246
Configuring Jetty Logging	246
Default Logging with Jetty's StdErrLog	247
Configuring Jetty Request Logs	249
Example: Logging with Apache Log4j	251
Example: Logging with Java's java.util.logging via Slf4j	252
Example: Logging with Java's java.util.logging via JavaUtilLog	254
Example: Logging with Logback	255
Example: Capturing Multiple Logging Frameworks with Slf4j	256
Example: Centralized Logging with Logback	259
Jetty Dump Tool	262

Chapter 9. Starting Jetty

Table of Contents

Startup Overview	115
Managing XML Based Startup Configuration	118
Managing Server Classpath	118
Managing Startup Modules	120
Managing Jetty Base and Jetty Home	124
Using start.jar	128
Startup a Unix Service using jetty.sh	131
Startup via Windows Service	135

Startup Overview

The `start.jar` bootstrap manages the startup of standalone Jetty. It is responsible for:

Building the classpath

The `start.jar` bootstrap builds a classpath for all the required Jetty features and their dependencies. It builds the classpath using either the `--lib` option to `start.jar` to add an individual classpath entry, or with the `--module` option that includes all the libs and their dependencies for a module (a named Jetty feature).

Instantiating the Server Components

The server and its components are instantiated using either Jetty IoC XML or spring. The Jetty server is just a collection of java POJOs for the server, connectors, session managers and others. These are instantiated, injected, and wired up together in XML files, commonly one per module/feature, that are passed as arguments to `start.jar`.

Resolving Server Filesystem Locations

The `start.jar` mechanism resolves canonical locations for the `${jetty.home}` and the `${jetty.base}` directories.

The `${jetty.home}` directory is the location of the standard distribution of Jetty.

The `${jetty.base}` directory is the location of the local server customization and configurations.

If you want to modify the Jetty distribution, base and home can be the same directory. Separating the base and home directories allows the distribution to remain unmodified, with all customizations in the the home directory, and thus simplifies subsequent server version upgrades.

Parameterizing the Server Configuration

XML files primarily determine the server configuration. Many of these files are parameterized to allow simple injection of host names, ports, passwords and more. The `start.jar` mechanism allows you to set parameters on the command line or in properties files.

To achieve these start up mechanisms, the `start.jar` uses:

Command line arguments

You can configure the entire server with command line arguments that specify libraries, properties and XML files. However in practice the INI and modules mechanisms (below) reduce the verbosity of the command line.

INI files

The `start.jar` mechanism uses the contents of the `${jetty.base}/start.ini` and `${jetty.base}/start.d/*.ini` files with each line equivalent to a `start.jar` command line argument. This means that either a global `start.ini` file or multiple `start.d/feature.ini` files control the configuration of the server.

Modules

Instead of explicitly listing all the libraries, properties and XML files for a feature, the `start.jar` mechanism allows you to create modules. You define a module in a `modules/*.mod` file, including the libraries, dependencies, XML, and template INI files for a Jetty feature. Thus you can use a single `--module=name` command line option as the equivalent of specifying many `--lib=location, feature.xml, name=value` arguments for a feature and all its dependencies. Modules also use their dependencies to control the ordering of libraries and XML files.

XML Files

XML files in either Jetty IoC or Spring format instantiate the actual POJO components of the server. This includes all major components such as connectors, keystores, session managers, and data sources. Typically there are one or more XML files per module, and these are defined and activated in the corresponding module.

Startup Example

The simplest way to start Jetty is via the `start.jar` mechanism using the following Java command line:

```
[user]$ cd jetty-distribution-9.2.1.v20140609  
[jetty-distribution-9.2.1.v20140609]$ java -jar start.jar --module=http jetty.port=8080
```

This command uses the `start.jar` mechanism to bootstrap the classpath, properties, and XML files with the metadata obtained from the `http` module. Specifically the `http` module is defined in the `${jetty.home}/modules/http.mod` file, and includes the following:

```
[jetty-distribution-9.2.1.v20140609]$ cat modules/http.mod  
[depend]  
server  
  
[xml]  
etc/jetty-http.xml  
  
[ini-template]  
jetty.port=8080  
http.timeout=30000
```

The `http` module declares that `http` depends on the `server` module, uses the `jetty-http.xml` file, and can be parameterized with `jetty.port` and `http.timeout` parameters. The INI-template section is not actually used by the command above, so the `jetty.port` must still be defined on the command line.

Following the `server` dependency, the `${jetty.home}/modules/server.mod` file includes:

```
[jetty-distribution-9.2.1.v20140609]$ cat modules/server.mod  
[lib]  
lib/servlet-api-3.1.jar  
lib/jetty-http-${jetty.version}.jar  
lib/jetty-server-${jetty.version}.jar  
lib/jetty-xml-${jetty.version}.jar  
lib/jetty-util-${jetty.version}.jar  
lib/jetty-io-${jetty.version}.jar  
  
[xml]  
etc/jetty.xml  
  
[ini-template]  
threads.min=10
```

```
threads.max=200
```

The `server` module declares the libraries the server needs and to use `jetty.xml` file. The combined metadata of the `http` and `server` modules results in `start.jar` generating the effective Java command line required to start Jetty.

Another way to see this is by asking Jetty what its configuration looks like:

```
[jetty-distribution-9.1.0-DEMO]$ java -jar start.jar --module=http jetty.port=9099 --list-config

Java Environment:
-----
java.home=/user/lib/jvm/jdk-7u21-x64/jre
java.vm.vendor=Oracle Corporation
java.vm.version=23.25-b01
java.vm.name=Java HotSpot(TM) 64-Bit Server VM
java.vm.info=mixed mode
java.runtime.name=Java(TM) SE Runtime Environment
java.runtime.version=1.7.0_25-b15
java.io.tmpdir=/tmp

Jetty Environment:
-----
jetty.home=/opt/jetty/jetty-distribution-9.1.0-DEMO
jetty.base=/opt/jetty/jetty-distribution-9.1.0-DEMO
jetty.version=9.1.0-DEMO

JVM Arguments:
-----
(no jvm args specified)

System Properties:
-----
jetty.home = /opt/jetty/jetty-distribution-9.1.0-DEMO
jetty.base = /opt/jetty/jetty-distribution-9.1.0-DEMO

Properties:
-----
jetty.port = 9099

Jetty Server Classpath:
-----
Version Information on 7 entries in the classpath.
Note: order presented here is how they would appear on the classpath.
      changes to the --module=name command line options will be reflected here.
0:          3.1.0 | ${jetty.home}/lib/servlet-api-3.1.jar
1:          3.1.RC0 | ${jetty.home}/lib/jetty-schemas-3.1.jar
2: 9.1.0-DEMO | ${jetty.home}/lib/jetty-http-9.1.0-DEMO.jar
3: 9.1.0-DEMO | ${jetty.home}/lib/jetty-server-9.1.0-DEMO.jar
4: 9.1.0-DEMO | ${jetty.home}/lib/jetty-xml-9.1.0-DEMO.jar
5: 9.1.0-DEMO | ${jetty.home}/lib/jetty-util-9.1.0-DEMO.jar
6: 9.1.0-DEMO | ${jetty.home}/lib/jetty-io-9.1.0-DEMO.jar

Jetty Active XMLs:
-----
${jetty.home}/etc/jetty.xml
${jetty.home}/etc/jetty-http.xml
```

This represents the entirety of the configuration that is applied to start Jetty.

If you don't want to use the `start.jar` bootstrap, you can start Jetty using a traditional Java command line.

The following is the equivalent `java` command line for what the `start.jar` bootstrap above performs.

```
[user]$ cd jetty-distribution-9.2.1.v20140609
[jetty-distribution-9.2.1.v20140609]$ echo jetty.port=8080 > /tmp/jetty.properties
[jetty-distribution-9.2.1.v20140609]$ export JETTY_HOME=`pwd` 
[jetty-distribution-9.2.1.v20140609]$ export JETTY_BASE=`pwd` 
[jetty-distribution-9.2.1.v20140609]$ export JETTY_VERSION="${project.version}" 
[jetty-distribution-9.2.1.v20140609]$ java -Djetty.home=$JETTY_HOME \
```

```
-Djetty.base=$JETTY_BASE \
-cp \
:$JETTY_HOME/lib/servlet-api-3.1.jar\
:$JETTY_HOME/lib/jetty-schemas-3.1.jar\
:$JETTY_HOME/lib/jetty-http-$JETTY_VERSION.jar\
:$JETTY_HOME/lib/jetty-server-$JETTY_VERSION.jar \
:$JETTY_HOME/lib/jetty-xml-$JETTY_VERSION.jar\
:$JETTY_HOME/lib/jetty-util-$JETTY_VERSION.jar\
:$JETTY_HOME/lib/jetty-io-$JETTY_VERSION.jar\
org.eclipse.jetty.xml.XmlConfiguration \
/tmp/jetty.properties \
$JETTY_HOME/etc/jetty.xml \
$JETTY_HOME/etc/jetty-http.xml
```

The java command line sets up the classpath with the core Jetty jars and the servlet API, executes the XmlConfiguration class, and passes it some XML files that define the server and an HTTP connector running on the port defined in the `jetty.properties` file.

You can further simplify the startup of this server by using the INI template defined by the modules to create a `start.ini` file with the command:

```
[user]$ cd jetty-distribution-9.2.1.v20140609
[jetty-distribution-9.2.1.v20140609]$ mkdir example-base
[example-base]$ cd example-base
[example-base]$ ls -la
total 8
drwxrwxr-x  2 user webgroup 4096 Oct  4 11:49 .
drwxrwxr-x 12 user webgroup 4096 Oct  4 11:49 ..
[example-base]$ java -jar ./start.jar --add-to-start=http
WARNING: http          initialised in ${jetty.base}/start.ini (appended)
WARNING: http          enabled in   ${jetty.base}/start.ini
WARNING: server        initialised in ${jetty.base}/start.ini (appended)
WARNING: server        enabled in    ${jetty.base}/start.ini
[example-base]$ ls -la
total 12
drwxrwxr-x  2 user webgroup 4096 Oct  4 11:55 .
drwxrwxr-x 12 user webgroup 4096 Oct  4 11:49 ..
-rw-rw-r--  1 user webgroup  250 Oct  4 11:55 start.ini
```

Once complete, you can edit the `start.ini` file to modify any parameters and you can run the server with the simple command:

```
[example-base]$ java -jar ./start.jar
```

Managing XML Based Startup Configuration

When you see XML files on the command line for startup of Jetty, they are always part of the Jetty IoC Configuration mechanism.

Internally, Jetty uses these XML files to build up Jetty with the features that you want to use.

The module mechanism present in Jetty determines the load order of the XML files.

The Jetty Base and Jetty Home resolution logic also applies, which allows you to override a XML file declared by a module with your XML by simply having the same named XML in your `${jetty.base}` directory location.

Managing Server Classpath

Jetty Server Classpath is determined by a combination of factors.

The `java.class.path` System Property

If you start Jetty with a JVM specified classpath, then Jetty will use the `java.class.path` System Property to populate the initial classpath.

Module specified Libraries

With Jetty 9.1+ the module system declares various libraries that are required for that module to operate, these module defined libraries are added to the Jetty Server classpath when any module is activated with library declarations.

Command Line Libraries

The command line option `--lib=<path>` can be used as a final means to add arbitrary entries to the Jetty Server classpath.

Of special note, there are 2 structural modules defined to ease some of this for you.

`--module=ext`

The `ext` module will enable the `lib/ext/*.jar` logic.

If this module is activated, then all jar files found in the `lib/ext/` paths will be automatically added to the Jetty Server Classpath.

`--module=resources`

The `resources` module will add the `resources/` directory to the classpath.

If you have 3rd party libraries that lookup resources from the classpath, put your files in here.

Logging libraries often have classpath lookup of their configuration files (eg: `log4j.properties`, `log4j.xml`, `logging.properties`, and `logback.xml`), so this would be the ideal setup for this sort of configuration demand.



Note

Both the `ext` and `resources` modules declare relative paths that follow [Jetty Base](#) and [Jetty Home path resolution rules](#).

Interrogating the Server Classpath

The Jetty start.jar has the ability to resolve the classpath from the command line + modules + configuration, and list the classpath entries it will use to start jetty.

The `--list-classpath` command line option is used as such.

(Demonstrated with the [demo-base from the Jetty Distribution](#))

```
[demo-base]$ java -jar ../start.jar --list-classpath

Jetty Server Classpath:
-----
Version Information on 42 entries in the classpath.
Note: order presented here is how they would appear on the classpath.
      changes to the --module=name command line options will be reflected here.
  0:         9.1.0-DEMO | ${jetty.home}/lib/jetty-client-9.1.0-DEMO.jar
  1:     1.4.1.v201005082020 | ${jetty.base}/lib/ext/
javax.mail.glassfish-1.4.1.v201005082020.jar
  2:         9.1.0-DEMO | ${jetty.base}/lib/ext/test-mock-resources-9.1.0-DEMO.jar
  3:             (dir) | ${jetty.home}/resources
  4:         3.1.0 | ${jetty.home}/lib/servlet-api-3.1.jar
  5:     3.1.RC0 | ${jetty.home}/lib/jetty-schemas-3.1.jar
  6:         9.1.0-DEMO | ${jetty.home}/lib/jetty-http-9.1.0-DEMO.jar
  7:     9.1.0-DEMO | ${jetty.home}/lib/jetty-continuation-9.1.0-DEMO.jar
  8:     9.1.0-DEMO | ${jetty.home}/lib/jetty-server-9.1.0-DEMO.jar
  9:     9.1.0-DEMO | ${jetty.home}/lib/jetty-xml-9.1.0-DEMO.jar
 10:    9.1.0-DEMO | ${jetty.home}/lib/jetty-util-9.1.0-DEMO.jar
 11:    9.1.0-DEMO | ${jetty.home}/lib/jetty-io-9.1.0-DEMO.jar
 12:    9.1.0-DEMO | ${jetty.home}/lib/jetty-jaas-9.1.0-DEMO.jar
 13:     9.1.0-DEMO | ${jetty.home}/lib/jetty-jndi-9.1.0-DEMO.jar
 14: 1.1.0.v201105071233 | ${jetty.home}/lib/jndi/
javax.activation-1.1.0.v201105071233.jar
 15: 1.4.1.v201005082020 | ${jetty.home}/lib/jndi/
javax.mail.glassfish-1.4.1.v201005082020.jar
 16:             1.2 | ${jetty.home}/lib/jndi/javax.transaction-api-1.2.jar
```

```
17:           9.1.0-DEMO | ${jetty.home}/lib/jetty-rewrite-9.1.0-DEMO.jar
18:           9.1.0-DEMO | ${jetty.home}/lib/jetty-security-9.1.0-DEMO.jar
19:           9.1.0-DEMO | ${jetty.home}/lib/jetty-servlet-9.1.0-DEMO.jar
20:           3.0.0 | ${jetty.home}/lib/jsp/javax.el-3.0.0.jar
21:           1.2.0.v201105211821 | ${jetty.home}/lib/jsp/
javax.servlet.jsp.jstl-1.2.0.v201105211821.jar
22:           2.3.2 | ${jetty.home}/lib/jsp/javax.servlet.jsp-2.3.2.jar
23:           2.3.1 | ${jetty.home}/lib/jsp/javax.servlet.jsp-api-2.3.1.jar
24:           2.3.3 | ${jetty.home}/lib/jsp/jetty-jsp-jdt-2.3.3.jar
25:           1.2.0.v201112081803 | ${jetty.home}/lib/jsp/
org.apache.taglibs.standard.glassfish-1.2.0.v201112081803.jar
26:           3.8.2.v20130121-145325 | ${jetty.home}/lib/jsp/
org.eclipse.jdt.core-3.8.2.v20130121.jar
27:           9.1.0-DEMO | ${jetty.home}/lib/jetty-plus-9.1.0-DEMO.jar
28:           9.1.0-DEMO | ${jetty.home}/lib/jetty-webapp-9.1.0-DEMO.jar
29:           9.1.0-DEMO | ${jetty.home}/lib/jetty-annotations-9.1.0-DEMO.jar
30:           4.1 | ${jetty.home}/lib/annotations/asm-4.1.jar
31:           4.1 | ${jetty.home}/lib/annotations/asm-commons-4.1.jar
32:           1.2 | ${jetty.home}/lib/annotations/javax.annotation-api-1.2.jar
33:           9.1.0-DEMO | ${jetty.home}/lib/jetty-deploy-9.1.0-DEMO.jar
34:           1.0 | ${jetty.home}/lib/websocket/javax.websocket-api-1.0.jar
35:           9.1.0-DEMO | ${jetty.home}/lib/websocket/javax.websocket-client-
impl-9.1.0-DEMO.jar
36:           9.1.0-DEMO | ${jetty.home}/lib/websocket/javax.websocket-server-
impl-9.1.0-DEMO.jar
37:           9.1.0-DEMO | ${jetty.home}/lib/websocket/websocket-api-9.1.0-DEMO.jar
38:           9.1.0-DEMO | ${jetty.home}/lib/websocket/websocket-client-9.1.0-
DEMO.jar
39:           9.1.0-DEMO | ${jetty.home}/lib/websocket/websocket-common-9.1.0-
DEMO.jar
40:           9.1.0-DEMO | ${jetty.home}/lib/websocket/websocket-server-9.1.0-
DEMO.jar
41:           9.1.0-DEMO | ${jetty.home}/lib/websocket/websocket-servlet-9.1.0-
DEMO.jar
```

Of note is that an attempt is made to list the internally declared version of each artifact on the Server Classpath, of potential help when diagnosing classpath issues.

Managing Startup Modules

Starting with Jetty 9.1, a new Module system was introduced. (It replaced the old `start.config` + `OPTIONS` techniques from past Jetty Distributions).

The standard Jetty Distribution ships with several modules defined in `${jetty.home}/modules/`

What a Jetty Startup Module Defines:

A Module Name

The name of the module is the keyword used by the `--module=<name>` command line argument to activate/enable modules, and also find dependant modules.

The filename of the module defines its name. (eg: `server.mod` becomes the module named "server")

List of Dependant Modules

All modules can declare that they depend on other modules with the `[depend]` section.

The list of dependencies is used to transitively resolve other modules that are deemed to be required based on the modules that you activate.

The order of modules defined in the graph of active modules is used to determine various execution order for configuration, such as Jetty IoC XML configurations, and to resolve conflicting property declarations.

Of note: there is a special section `[optional]` used to describe structurally dependant modules that are not technically required, but might be of use to your specific configuration.

List of Libraries

Module can optionally declare that they have libraries that they need to function properly.

The [lib] section declares a set of pathnames that follow the [Jetty Base and Jetty Home path resolution rules](#).

List of Jetty IoC XML Configurations

A Module can optionally declare a list of Jetty IoC XML configurations used to wire up the functionality that this module defines.

The [xml] section declares a set of pathnames that follow the [Jetty Base and Jetty Home path resolution rules](#).

Ideally, all XML files are parameterized to accept properties to configure the various elements of the standard configuration. Allowing for a simplified configuration of Jetty for the vast majority of deployments.

The execution order of the Jetty IoC XML configurations is determined by the graph of active module dependencies resolved via the [depend] sections.

If the default XML is not sufficient to satisfy your needs, you can override this XML by making your own in the \${jetty.base}/etc/ directory, with the same name. The resolution steps for Jetty Base and Jetty Home will ensure that your copy from \${jetty.base} will be picked up over the default one in \${jetty.home}.

Jetty INI Template

Each module can optionally declare a startup ini template that is used to insert/append/inject sample configuration elements into the start.ini or start.d/*.ini files when using the --add-to-start=<name> or --add-to-startd=<name> command line arguments in start.jar.

Commonly used to present some of the parameterized property options from the Jetty IoC XML configuration files also referenced in the same module.

The [ini-template] section declares this section of sample configuration.

Required Files and Directories

If the activation of a module requires some paths to exist, the [files] section defines them.

There are 2 modes of operation of the entries in this section.

Ensure Directory Exists

If you add a pathname that ends in "/" (slash), such as "webapps/", then that directory will be created if it does not yet exist in \${jetty.base} / <pathname> (eg: "webapps/" will result in \${jetty.base}/webapps/ being created)

Download File

There is a special syntax to allow you to download a file into a specific location if it doesn't exist yet.

<url>:<pathname>

Currently, the <url> must be a http:// scheme URL ([file a bug](#) if you want more schemes supported). The <pathname> portion follows the [Jetty Base and Jetty Home path resolution rules](#).

Example:

```
http://repo.corp.com/maven/corp-security-policy-1.0.jar:lib/corp-security-policy.jar
```

This will check for the existence of lib/corp-security-policy.jar, and if it doesn't exist, it will download the jar file from http://repo.corp.com/maven/corp-security-policy-1.0.jar

Enabling Modules

Jetty ships with many modules defined, and a small subset predefined in the `start.ini` found in the jetty distribution.



Tip

The default distribution has a commingled `${jetty.home}` and `${jetty.base}`. Where the directories for `${jetty.home}` and `${jetty.base}` point to the same location.

It is highly encouraged that you learn about the differences in [Jetty Base vs Jetty Home](#) and take full advantage of this setup.

When you want enable a module, you can use the `--module=<name>` syntax on the command line (or `start.ini`) to enable it + all of its dependant modules.

An example of this, with a new, empty, base directory.

```
[~/home/user]$ mkdir my-base
[~/home/user]$ cd my-base
[my-base]$ java -jar ../jetty-distribution/start.jar
WARNING: Nothing to start, exiting ...

Usage: java -jar start.jar [options] [properties] [configs]
       java -jar start.jar --help # for more information
[my-base]$ java -jar ../jetty-distribution/start.jar --list-config

Java Environment:
-----
java.home=/home/joakim/java/jdk-7u21-x64/jre
java.vm.vendor=Oracle Corporation
java.vm.version=23.21-b01
java.vm.name=Java HotSpot(TM) 64-Bit Server VM
java.vm.info=mixed mode
java.runtime.name=Java(TM) SE Runtime Environment
java.runtime.version=1.7.0_21-b11
java.io.tmpdir=/tmp

Jetty Environment:
-----
jetty.home=/home/joakim/code/intalio/distros/jetty-distribution-9.1.0-DEMO
jetty.base=/home/joakim/code/intalio/distros/jetty-distribution-9.1.0-DEMO/my-base
jetty.version=9.1.0-DEMO

JVM Arguments:
-----
(no jvm args specified)

System Properties:
-----
jetty.base = /home/joakim/code/intalio/distros/jetty-distribution-9.1.0-DEMO/my-base
jetty.home = /home/joakim/code/intalio/distros/jetty-distribution-9.1.0-DEMO

Properties:
-----
(no properties specified)

Jetty Server Classpath:
-----
No classpath entries and/or version information available show.

Jetty Active XMLs:
-----
(no xml files specified)
```

Can't start, as there is no configuration, yet. Lets use a simple one that will start up support for webapps and hot deployment.

```
[my-base]$ java -jar ../jetty-distribution/start.jar --module=webapp,deploy
```

Starting Jetty

```
2013-10-16 13:30:45.636:INFO:oejs.Server:main: jetty-9.1.0-DEMO
2013-10-16 13:30:45.656:INFO:oejdp.ScanningAppProvider:main: Deployment monitor [file:/home/user/my-base/webapps] at interval 1
[Ctrl+C]
[my-base]$ java -jar ./jetty-distribution/start.jar --module=webapp,deploy --list-config
Java Environment:
-----
java.home=/home/joakim/java/jdk-7u21-x64/jre
java.vm.vendor=Oracle Corporation
java.vm.version=23.21-b01
java.vm.name=Java HotSpot(TM) 64-Bit Server VM
java.vm.info=mixed mode
java.runtime.name=Java(TM) SE Runtime Environment
java.runtime.version=1.7.0_21-b11
java.io.tmpdir=/tmp

Jetty Environment:
-----
jetty.home=/home/joakim/code/intalio/distros/jetty-distribution-9.1.0-DEMO
jetty.base=/home/joakim/code/intalio/distros/jetty-distribution-9.1.0-DEMO/my-base
jetty.version=9.1.0-DEMO

JVM Arguments:
-----
(no jvm args specified)

System Properties:
-----
jetty.base = /home/joakim/code/intalio/distros/jetty-distribution-9.1.0-DEMO/my-base
jetty.home = /home/joakim/code/intalio/distros/jetty-distribution-9.1.0-DEMO

Properties:
-----
(no properties specified)

Jetty Server Classpath:
-----
Version Information on 11 entries in the classpath.
Note: order presented here is how they would appear on the classpath.
      changes to the --module=name command line options will be reflected here.
0:          3.1.0 | ${jetty.home}/lib/servlet-api-3.1.jar
1:          3.1.RC0 | ${jetty.home}/lib/jetty-schemas-3.1.jar
2:          9.1.0-DEMO | ${jetty.home}/lib/jetty-http-9.1.0-DEMO.jar
3:          9.1.0-DEMO | ${jetty.home}/lib/jetty-continuation-9.1.0-DEMO.jar
4:          9.1.0-DEMO | ${jetty.home}/lib/jetty-server-9.1.0-DEMO.jar
5:          9.1.0-DEMO | ${jetty.home}/lib/jetty-xml-9.1.0-DEMO.jar
6:          9.1.0-DEMO | ${jetty.home}/lib/jetty-util-9.1.0-DEMO.jar
7:          9.1.0-DEMO | ${jetty.home}/lib/jetty-io-9.1.0-DEMO.jar
8:          9.1.0-DEMO | ${jetty.home}/lib/jetty-servlet-9.1.0-DEMO.jar
9:          9.1.0-DEMO | ${jetty.home}/lib/jetty-webapp-9.1.0-DEMO.jar
10:         9.1.0-DEMO | ${jetty.home}/lib/jetty-deploy-9.1.0-DEMO.jar

Jetty Active XMLs:
-----
${jetty.home}/etc/jetty.xml
${jetty.home}/etc/jetty-deploy.xml
```

From here, we can see that the Jetty now starts up with basic WebApp support and automatic deployment.

Just create a `my-base/webapps/` directory and toss in a simple war file and you now have a functional server.

To see what modules are active, you can use the `--list-modules` command line argument.

```
[my-base]$ java -jar ./jetty-distribution/start.jar --module=webapp,deploy --list-
modules
...(snip)...
Jetty Active Module Tree:
-----
+ Module: server [transitive]
  + Module: servlet [transitive]
    + Module: webapp [enabled]
      + Module: deploy [enabled]
```

We declared webapp and deploy modules on the command line with the servlet and server modules being brought into the active module tree transitively.

Managing Jetty Base and Jetty Home

Starting with Jetty 9.1, it is now possible to maintain a separation between the binary installation of the standalone Jetty (known as `-${jetty.home}`), and the customizations for your specific environment (known as `-${jetty.base}`).

Jetty Base

Also known as the `-${jetty.base}` property

This is the location for your configurations and customizations to the Jetty distribution.

Jetty Home

Also known as the `-${jetty.home}` property.

This is the location for the Jetty distribution binaries, default XML IoC configurations, and default module definitions.

Potential configuration is resolved from these 2 directory locations.

Configuration Resolution Rules

Check Jetty Base

If the referenced configuration exists, relative to the defined Jetty Base, use it.

Check Jetty Home

If the referenced configuration exists, relative to the defined Jetty Home, use it.

Use `java.io.File(String pathname)` Logic

Lastly, use the reference as a `java.io.File(String pathname)` [http://docs.oracle.com/javase/7/docs/api/java/io/File.html#File(java.lang.String)] reference, following the default resolution rules outlined by that constructor.

In brief, the reference will be used as-is, be it relative (to current working directory, aka `-${user.dir}`) or absolute path, or even network reference (such as on Windows and use of UNC paths).

For more details on how startup with start.jar works, see [Using start.jar: Executing](#)

Demo-Base in the Jetty Distribution

The Jetty Distribution comes with an example `-${jetty.base}` which enables the various demonstration webapps and server configurations.

How to use the demo-base directory as a Jetty Base directory.

```
[jetty-distribution-9.2.1.v20140609]$ ls -la
total 496
drwxrwxr-x 11 user group 4096 Oct  8 15:23 .
drwxr-xr-x 14 user group 4096 Oct  8 13:04 ..
drwxrwxr-x  2 user group 4096 Oct  8 06:54 bin/
drwxrwxr-x  6 user group 4096 Oct  8 06:54 demo-base/
drwxrwxr-x  2 user group 4096 Oct 11 15:14 etc/
drwxrwxr-x 11 user group 4096 Oct  8 06:54 lib/
-rw-rw-r--  1 user group 30012 Sep 30 19:55 license-eplv10-aslv20.html
drwxrwxr-x  2 user group 4096 Oct  8 06:54 logs/
drwxrwxr-x  2 user group 4096 Oct  8 06:54 modules/
-rw-rw-r--  1 user group 6262 Sep 30 19:55 notice.html
-rw-rw-r--  1 user group 1249 Sep 30 19:55 README.TXT
```

```

drwxrwxr-x  2 user group  4096 Oct  8 06:54 resources/
drwxrwxr-x  2 user group  4096 Oct  8 06:54 start.d/
-rw-rw-r--  1 user group  1780 Sep 30 19:55 start.ini
-rw-rw-r--  1 user group  71921 Sep 30 19:55 start.jar
-rw-rw-r--  1 user group 336468 Sep 30 19:55 VERSION.txt
drwxrwxr-x  2 user group  4096 Oct  8 06:54 webapps/
[jetty-distribution-9.2.1.v20140609]$ cd demo-base
[demo-base]$ java -jar ../start.jar
2013-10-16 09:08:47.800:WARN::main: demo test-realm is deployed. DO NOT USE IN
PRODUCTION!
2013-10-16 09:08:47.802:INFO:oejs.Server:main: jetty-${project.version}
2013-10-16 09:08:47.817:INFO:oejdp.ScanningAppProvider:main: Deployment monitor [file:/home/user/jetty-distribution-9.2.1.v20140609/demo-base/webapps/] at interval 1
2013-10-16 09:08:48.072:WARN::main: async-rest webapp is deployed. DO NOT USE IN
PRODUCTION!
...

```

As you can see above, you are executing the demo-base configuration using the Jetty Base concepts.

If you want to see what the Jetty Base looks like without executing Jetty, you can simply list the configuration

```

[my-base]$ java -jar ../start.jar --list-config

Java Environment:
-----
java.home=/usr/lib/jvm/jdk-7u21-x64/jre
java.vm.vendor=Oracle Corporation
java.vm.version=23.21-b01
java.vm.name=Java HotSpot(TM) 64-Bit Server VM
java.vm.info=mixed mode
java.runtime.name=Java(TM) SE Runtime Environment
java.runtime.version=1.7.0_21-b11
java.io.tmpdir=/tmp

Jetty Environment:
-----
jetty.home=/home/user/jetty-distribution-9.1.0-DEMO
jetty.base=/home/user/jetty-distribution-9.1.0-DEMO/demo-base
jetty.version=9.1.0-DEMO

JVM Arguments:
-----
(no jvm args specified)

System Properties:
-----
jetty.base = /home/user/jetty-distribution-9.1.0-DEMO/demo-base
jetty.home = /home/user/jetty-distribution-9.1.0-DEMO

Properties:
-----
demo.realm = etc/realm.properties
https.port = 8443
https.timeout = 30000
jaas.login.conf = etc/login.conf
jetty.dump.start = false
jetty.dump.stop = false
jetty.keymanager.password = OBF:lu2ulwm1lz7s1z7alwnllu2g
jetty.keystore = etc/keystore
jetty.keystore.password = OBF:lvny1zlolx8elvnwlvn6lx8glzlulvn4
jetty.port = 8080
jetty.secure.port = 8443
jetty.truststore = etc/keystore
jetty.truststore.password = OBF:lvny1zlolx8elvnwlvn6lx8glzlulvn4
org.eclipse.jetty.websocket.jsr356 = false
threads.max = 200
threads.min = 10
threads.timeout = 60000

Jetty Server Classpath:
-----
Version Information on 42 entries in the classpath.
Note: order presented here is how they would appear on the classpath.
      changes to the --module=name command line options will be reflected here.
0:          9.1.0-DEMO | ${jetty.home}/lib/jetty-client-9.1.0-DEMO.jar

```

```

1:      1.4.1.v201005082020 | ${jetty.base}/lib/ext/
javax.mail.glassfish-1.4.1.v201005082020.jar
2:          9.1.0-DEMO | ${jetty.base}/lib/ext/test-mock-resources-9.1.0-DEMO.jar
3:              (dir) | ${jetty.home}/resources
4:                  3.1.0 | ${jetty.home}/lib/servlet-api-3.1.jar
5:                  3.1.RC0 | ${jetty.home}/lib/jetty-schemas-3.1.jar
6:                  9.1.0-DEMO | ${jetty.home}/lib/jetty-http-9.1.0-DEMO.jar
7:                  9.1.0-DEMO | ${jetty.home}/lib/jetty-continuation-9.1.0-DEMO.jar
8:                  9.1.0-DEMO | ${jetty.home}/lib/jetty-server-9.1.0-DEMO.jar
9:                  9.1.0-DEMO | ${jetty.home}/lib/jetty-xml-9.1.0-DEMO.jar
10:                 9.1.0-DEMO | ${jetty.home}/lib/jetty-util-9.1.0-DEMO.jar
11:                 9.1.0-DEMO | ${jetty.home}/lib/jetty-io-9.1.0-DEMO.jar
12:                 9.1.0-DEMO | ${jetty.home}/lib/jetty-jaas-9.1.0-DEMO.jar
13:                 9.1.0-DEMO | ${jetty.home}/lib/jetty-jndi-9.1.0-DEMO.jar
14:             1.1.0.v201105071233 | ${jetty.home}/lib/jndi/
javax.activation-1.1.0.v201105071233.jar
15:             1.4.1.v201005082020 | ${jetty.home}/lib/jndi/
javax.mail.glassfish-1.4.1.v201005082020.jar
16:                 1.2 | ${jetty.home}/lib/jndi/javax.transaction-api-1.2.jar
17:                 9.1.0-DEMO | ${jetty.home}/lib/jetty-rewrite-9.1.0-DEMO.jar
18:                 9.1.0-DEMO | ${jetty.home}/lib/jetty-security-9.1.0-DEMO.jar
19:                 9.1.0-DEMO | ${jetty.home}/lib/jetty-servlet-9.1.0-DEMO.jar
20:                     3.0.0 | ${jetty.home}/lib/jsp/javax.el-3.0.0.jar
21:             1.2.0.v201105211821 | ${jetty.home}/lib/jsp/
javax.servlet.jsp.jstl-1.2.0.v201105211821.jar
22:                 2.3.2 | ${jetty.home}/lib/jsp/javax.servlet.jsp-2.3.2.jar
23:                 2.3.1 | ${jetty.home}/lib/jsp/javax.servlet.jsp-api-2.3.1.jar
24:                 2.3.3 | ${jetty.home}/lib/jsp/jetty-jsp-jdt-2.3.3.jar
25:             1.2.0.v201112081803 | ${jetty.home}/lib/jsp/
org.apache.taglibs.standard.glassfish-1.2.0.v201112081803.jar
26:             3.8.2.v20130121-14325 | ${jetty.home}/lib/jsp/
org.eclipse.jdt.core-3.8.2.v20130121.jar
27:                 9.1.0-DEMO | ${jetty.home}/lib/jetty-plus-9.1.0-DEMO.jar
28:                 9.1.0-DEMO | ${jetty.home}/lib/jetty-webapp-9.1.0-DEMO.jar
29:                 9.1.0-DEMO | ${jetty.home}/lib/jetty-annotations-9.1.0-DEMO.jar
30:                     4.1 | ${jetty.home}/lib/annotations/asm-4.1.jar
31:                     4.1 | ${jetty.home}/lib/annotations/asm-commons-4.1.jar
32:                     1.2 | ${jetty.home}/lib/annotations/javax.annotation-api-1.2.jar
33:                 9.1.0-DEMO | ${jetty.home}/lib/jetty-deploy-9.1.0-DEMO.jar
34:                     1.0 | ${jetty.home}/lib/websocket/javax.websocket-api-1.0.jar
35:                 9.1.0-DEMO | ${jetty.home}/lib/websocket/javax.websocket-client-
impl-9.1.0-DEMO.jar
36:                 9.1.0-DEMO | ${jetty.home}/lib/websocket/javax.websocket-server-
impl-9.1.0-DEMO.jar
37:                     9.1.0-DEMO | ${jetty.home}/lib/websocket/websocket-api-9.1.0-DEMO.jar
38:                     9.1.0-DEMO | ${jetty.home}/lib/websocket/websocket-client-9.1.0-
DEMO.jar
39:                     9.1.0-DEMO | ${jetty.home}/lib/websocket/websocket-common-9.1.0-
DEMO.jar
40:                     9.1.0-DEMO | ${jetty.home}/lib/websocket/websocket-server-9.1.0-
DEMO.jar
41:                     9.1.0-DEMO | ${jetty.home}/lib/websocket/websocket-servlet-9.1.0-
DEMO.jar

Jetty Active XMLs:
-----
${jetty.home}/etc/jetty.xml
${jetty.home}/etc/jetty-http.xml
${jetty.home}/etc/jetty-jaas.xml
${jetty.home}/etc/jetty-rewrite.xml
${jetty.home}/etc/jetty-ssl.xml
${jetty.home}/etc/jetty-https.xml
${jetty.home}/etc/jetty-plus.xml
${jetty.home}/etc/jetty-annotations.xml
${jetty.home}/etc/jetty-deploy.xml
${jetty.base}/etc/demo-rewrite-rules.xml
${jetty.base}/etc/test-realm.xml

```

This demonstrates the powerful `--list-config` command line option and how you can use it to see what the configuration will look like when starting Jetty. From the Java environment, to the system properties, to the classpath, and finally the Active Jetty IoC XML used to build up your Jetty server configuration.

Of note, is that the output will make it known where the configuration elements came from, be it in either in `${jetty.home}` or `${jetty.base}`.

If you look at the \${jetty.base}/start.ini you will see something like the following.

```
[my-base]$ cat start.ini
# Enable security via jaas, and configure it
--module=jaas
jaas.login.conf=etc/login.conf

# Enable rewrite examples
--module=rewrite
etc/demo-rewrite-rules.xml

# WebSocket chat examples needs websocket enabled
# Don't start for all contexts (set to true in test.xml context)
org.eclipse.jetty.websocket.jsr356=false
--module=websocket

# Create and configure the test realm
etc/test-realm.xml
demo.realm=etc/realm.properties

# Initialize module server
--module=server
threads.min=10
threads.max=200
threads.timeout=60000
jetty.dump.start=false
jetty.dump.stop=false

--module=deploy
--module=jsp
--module=ext
--module=resources
--module=client
--module=annotations
```

The \${jetty.base}/start.ini is the main startup configuration entry point for Jetty. In this example you will see that we are enabling a few modules for Jetty, specifying some properties, and also referencing some Jetty IoC XML files (namely the etc/demo-rewrite-rules.xml and etc/test-realm.xml files)

When Jetty's start.jar resolves the entries in the start.ini, it will follow the [resolution rules above](#).

For example, the reference to etc/demo-rewrite-rules.xml was found in \${jetty.base}/etc/demo-rewrite-rules.xml.

Declaring Jetty Base

The Jetty Distribution's start.jar is the component that manages the behavior of this separation.

The Jetty start.jar and XML files always assume that both \${jetty.home} and \${jetty.base} are defined when starting Jetty.

You can opt to manually define the \${jetty.home} and \${jetty.base} directories, such as this:

```
[jetty-distribution-9.2.1.v20140609]$ pwd
/home/user/jetty-distribution-9.2.1.v20140609
[jetty-distribution-9.2.1.v20140609]$ java -jar start.jar \
    jetty.home=/home/user/jetty-distribution-9.2.1.v20140609 \
    jetty.base=/home/user/my-base
2013-10-16 09:08:47.802:INFO:oejs.Server:main: jetty-${project.version}
2013-10-16 09:08:47.817:INFO:oejdp.ScanningAppProvider:main: Deployment monitor [file:/home/user/my-base/webapps/] at interval 1
...
```

Or you can declare one directory and let the other one be discovered.

The following example uses default discovery of \${jetty.home} by using the parent directory of wherever start.jar itself is, and a manual declaration of \${jetty.base}.

```
[jetty-distribution-9.2.1.v20140609]$ pwd  
/home/user/jetty-distribution-9.2.1.v20140609  
[jetty-distribution-9.2.1.v20140609]$ java -jar start.jar jetty.base=/home/user/my-base  
2013-10-16 09:08:47.802:INFO:oejs.Server:main: jetty-${project.version}  
2013-10-16 09:08:47.817:INFO:oejdp.ScanningAppProvider:main: Deployment monitor [file:/  
home/user/my-base/webapps/] at interval 1  
...
```

But Jetty recommends that you always start Jetty by sitting in the directory that is your \${jetty.base} and starting Jetty by referencing the start.jar remotely.

The following demonstrates this by allowing default discovery of \${jetty.home} via locating the start.jar, and using the user.dir System Property for \${jetty.base}.

```
[jetty-distribution-9.2.1.v20140609]$ pwd  
/home/user/jetty-distribution-9.2.1.v20140609  
[jetty-distribution-9.2.1.v20140609]$ cd /home/user/my-base  
[my-base]$ java -jar /home/user/jetty-distribution-9.2.1.v20140609/start.jar  
2013-10-16 09:08:47.802:INFO:oejs.Server:main: jetty-${project.version}  
2013-10-16 09:08:47.817:INFO:oejdp.ScanningAppProvider:main: Deployment monitor [file:/  
home/user/my-base/webapps/] at interval 1  
...
```

Important



Be aware of the user.dir system property, it can only be safely set when the JVM starts, and many 3rd party libraries (especially logging) use this system property.

It is strongly recommended that you sit in the directory that is your desired \${jetty.base} when starting Jetty to have consistent behavior and use of the user.dir system property.

Using start.jar

The most basic way of starting the Jetty standalone server is to execute the start.jar, which is a bootstrap for starting Jetty with the configuration you want.

```
[jetty-distribution-9.2.1.v20140609]$ java -jar start.jar  
2013-09-23 11:27:06.654:INFO:oejs.Server:jetty-${project.version}  
...
```

Jetty is a highly modularized web server container. Very little is mandatory and required, and most components are optional; you enable or disable them according to the needs of your environment.

At its most basic, you configure Jetty from two elements:

1. A set of libraries and directories that make up the server classpath.
2. A set of Jetty XML configuration files (IoC style) that establish how to build the Jetty server and its components.

Starting with Jetty 9.1 you have more options on how to configure Jetty (these are merely syntactic sugar that eventually resolve into the two basic configuration components).

Jetty 9.1 Startup Features include:

- A separation of the Jetty distribution binaries in \${jetty.home} and the environment specific configurations (and binaries) found in \${jetty.base} (detailed in [Managing Jetty Base and Jetty Home](#).)
- You can enable a set of libraries and XML configuration files via the newly introduced [module system](#).

- All of the pre-built XML configuration files shipped in Jetty are now parameterized with properties that you can specify in your \${jetty.base}/start.ini (demonstrated in [Quick Start Configuration](#)).

There is no longer a

These are powerful new features, made to support a variety of styles of configuring Jetty, from a simple property based configuration, to handling multiple installations on a server, to customized stacks of technology on top of Jetty, and even the classic, custom XML configurations of old.

For example, if you use the \${jetty.base} concepts properly, you can upgrade the Jetty distribution without having to remake your entire tree of modifications to Jetty. Simply separate out your specific modifications to the \${jetty.base}, and in the future, just upgrade your \${jetty.home} directory with a new Jetty distribution.

Executing start.jar

When you execute `start.jar` the system:

- Loads and parses all INIs found in \${jetty.base}/start.d/*.ini as command line arguments.
- Loads and parses \${jetty.base}/start.ini as command line arguments.
- Parses actual command line arguments used to execute `start.jar` itself.
- Resolves any XML configuration files, modules, and libraries using base vs. home resolution steps:
 1. Checks whether file exists as relative reference to \${jetty.base}.
 2. Checks whether file exists as relative reference to \${jetty.home}.
 3. Uses default behavior of `java.io.File`.

(Relative to `System.getProperty("user.dir")` and then as absolute file system path)

- Loads any dependent modules (merges XXNK, library, and properties results with active command line).
- Builds out server classpath.
- Determines run mode:
 - Shows informational command line options and exit.
 - Executes Jetty normally, waits for Jetty to stop.
 - Executes a forked JVM to run Jetty in, waits for forked JVM to exit.
- If you want to start Jetty:
 - Load each XML configuration (in the order determined by the INIs and module system).
 - Let XML configuration start Jetty.

start.jar Command Line Options

Basic command line options:

--help

Obtains the current list of command line options and some basic usage help.

--version

Shows the list of server classpath entries, and prints version information found for each entry.

--list-classpath

Similar to --version, shows the server classpath.

--list-config

Lists the resolved configuration that will start Jetty.

Output includes:

- Java environment
- Jetty environment
- JVM arguments
- Properties
- Server classpath
- Server XML configuration files

--dry-run

Prints the resolved command line that `start.jar` should use to start a forked instance of Jetty.

--exec

Starts a forked instance of Jetty.

Debug and Start Logging Options:

--debug

Enables debugging output of the startup procedure.

Note: This does not set up debug logging for Jetty itself.

--start-log-file=<filename>

Sends all startup output to the filename specified.

Filename is relative to `#{jetty.base}`.

This is useful for capturing startup issues where the Jetty-specific logger has not yet kicked in due to a possible startup configuration error.

Module Management:

--list-modules

Lists all the modules defined by the system.

Looks for module files using the [normal \\${jetty.base} and \\${jetty.home} resolution logic](#).

Also lists enabled state based on information present on the command line, and all active startup INI files.

--module=<name>,(<name>)*

Enables one or more modules by name (use --list-modules to see the list of available modules).

This enables all transitive (dependent) modules from the module system as well.

If you use this from the shell command line, it is considered a temporary effect, useful for testing out a scenario. If you want this module to always be enabled, add this command to your `#{jetty.base}/start.ini`.

--add-to-start=<name>,(<name>)*

Enables a module by appending lines to the \${jetty.base}/start.ini file.

The lines that are added are provided by the module-defined INI templates.

Note: Transitive modules are also appended.

--add-to-startd=<name>,(<name>)*

Enables a module via creation of a module-specific INI file in the \${jetty.base}/start.d/ directory.

The content of the new INI is provided by the module-defined ini templates.

Note: Transitive modules are also created in the same directory as their own INI files.

--write-module-graph=<filename>

Advanced feature: Creates a graphviz [dot file](#) of the module graph as it exists for the active \${jetty.base}.

```
# generate module.dot
$ java -jar start.jar --module=websocket --write-module-graph=modules.dot

# post process to a PNG file
$ dot -Tpng -o modules.png modules.dot
```

See [graphviz.org](#) for details on [how to post-process this dotty file](#) into the output best suited for your needs.

Command Line Stop Option:

--stop

Sends a stop signal to the running Jetty instance.

Note: The server must have been started with various stop properties for this to work.

Properties:

STOP.PORT=<number>

The port to use to stop the running Jetty server. This is an internal port, opened on localhost, used solely for stopping the running Jetty server. Choose a port that you do not use to serve web traffic.

Required for --stop to function.

STOP.KEY=<alphanumeric>

The passphrase defined to stop the server.

Required for --stop to function.

STOP.WAIT=<number>

The time (in seconds) to wait for confirmation that the running Jetty server has stopped. If not specified, the stopper waits indefinitely for the server to stop.

If the time specified elapses, without a confirmation of server stop, then the --stop command exits with a non-zero return code.

Startup a Unix Service using jetty.sh

The Standalone Jetty distribution ships with a bin/jetty.sh script that can be used by various Unix (including OSX) to manage jetty startup.

This script is suitable for setting up Jetty as a service in Unix.

Quick-Start a Jetty Service

The minimum steps to get Jetty to run as a Service

```
[ /opt/jetty]# tar -zxf /home/user/downloads/jetty-distribution-9.2.1.v20140609.tar.gz
[ /opt/jetty]# cd jetty-distribution-9.2.1.v20140609/
[ /opt/jetty/jetty-distribution-9.2.1.v20140609]# ls
bin          lib           modules      resources  start.jar
demo-base    license-eplv10-aslv20.html notice.html  start.d   VERSION.txt
etc          logs          README.TXT  start.ini  webapps

[ /opt/jetty/jetty-distribution-9.2.1.v20140609]# cp bin/jetty.sh /etc/init.d/jetty
[ /opt/jetty/jetty-distribution-9.2.1.v20140609]# cd /opt/jetty

[ /opt/jetty]# echo JETTY_HOME=`pwd` > /etc/default/jetty
[ /opt/jetty]# cat /etc/default/jetty
JETTY_HOME=/opt/jetty/jetty-distribution-9.2.1.v20140609

[ /opt/jetty]# service jetty start
Starting Jetty: OK Wed Nov 20 10:26:53 MST 2013
```

From this simple demonstration we can see that Jetty started successfully as a Unix Service from the `/opt/jetty/jetty-distribution-9.2.1.v20140609` directory.

This looks all fine and dandy, however you are running a default Jetty on the root user id.

Practical Setup of a Jetty Service

There are various ways this can be accomplished, mostly depending on your Unix environment (and possibly corporate policies)

The techniques outlined here assume an installation on Linux (demonstrated on Ubuntu 12.04.3 LTS).

Prepare System

Prepare some empty directories to work with.

```
# mkdir -p /opt/jetty
# mkdir -p /opt/web/mybase
# mkdir -p /opt/jetty/temp
```

The directory purposes are as follows:

/opt/jetty

Where the Jetty Distribution will be unpacked into

/opt/web/mybase

Where your specific set of webapps will be located, including all of the configuration required of the server to make them operational.

/opt/jetty/temp

This is the temporary directory assigned to Java by the Service Layer (this is what Java sees as the `java.io.tmpdir` System Property)

This is intentionally kept separate from the standard temp directory of `/tmp`, as this location doubles as the Servlet Spec work directory. (It is our experience that the standard temp directory is often managed by various cleanup scripts that wreak havoc on a long running Jetty server)

Make sure you have Java 7 Installed

Jetty \${project.version} requires Java 7 (or greater) to run. Make sure you have it installed.

```
# apt-get install openjdk-7-jdk
```

Or download Java 7 from: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

```
# java -version
java version "1.6.0_27"
OpenJDK Runtime Environment (IcedTea6 1.12.6) (6b27-1.12.6-1ubuntu0.12.04.2)
OpenJDK 64-Bit Server VM (build 20.0-b12, mixed mode)

# update-alternatives --list java
/usr/lib/jvm/java-6-openjdk-amd64/jre/bin/java
/usr/lib/jvm/java-7-openjdk-amd64/jre/bin/java

# update-alternatives --config java
There are 2 choices for the alternative java (providing /usr/bin/java).

      Selection    Path                      Priority   Status
-----+-----+-----+-----+-----+-----+-----+
*   0       /usr/lib/jvm/java-6-openjdk-amd64/jre/bin/java  1061      auto mode
      1       /usr/lib/jvm/java-6-openjdk-amd64/jre/bin/java  1061      manual mode
      2       /usr/lib/jvm/java-7-openjdk-amd64/jre/bin/java  1051      manual mode

Press enter to keep the current choice[*], or type selection number: 2
update-alternatives: using /usr/lib/jvm/java-7-openjdk-amd64/jre/bin/java to provide /usr/bin/java (java) in manual mode.

# java -version
java version "1.7.0_25"
OpenJDK Runtime Environment (IcedTea 2.3.10) (7u25-2.3.10-1ubuntu0.12.04.2)
OpenJDK 64-Bit Server VM (build 23.7-b01, mixed mode)
```

Create a User to Run Jetty

It is recommended that you create a user to specifically run Jetty. This user should have the minimum set of privileges needed to run Jetty.

```
# useradd --user-group --shell /bin/false --home-dir /opt/jetty/temp jetty
```

This will create a user called `jetty`, belonging to the group called `jetty`, with no shell access (aka `/bin/false`), and home directory at `/opt/jetty/temp`.

Download and Unpack Your Distribution

Grab a copy of the distribution from the [Official Eclipse Download Site](#)

Unpack it into place.

```
[/opt/jetty]# tar -zxf /home/user/Downloads/jetty-distribution-9.1.0.v20131115.tar.gz
[/opt/jetty]# ls -F
jetty-distribution-9.1.0.v20131115/
[/opt/jetty]# mkdir /opt/jetty/temp
```

It might seem strange or undesirable to unpack the first portion of the `jetty-distribution` directory name too. But starting with Jetty 9.1 the split between `${jetty.home}` and `${jetty.base}` allows for easier upgrades of Jetty itself while isolating your webapp specific configuration.

The `/opt/jetty/temp` directory is created as a durable place for Jetty to use for temp and working directories. Many Unix systems will periodically clean out the `/tmp` directory, this behavior is undesired in a Servlet container and has been known to cause problems. This durable directory at `/opt/jetty/temp` solves for that behavior.

Configure your WebApp base

The directory at `/opt/web/mybase` is going to be a `${jetty.base}`, so lets configure it to hold your webapp and its configuration.

Tip

 In past versions of Jetty, you would configure / modify / add to the jetty-distribution directory directly. While this is still supported, we encourage you to setup a proper \${jetty.base} directory, as it will benefit you with easier jetty-distribution upgrades in the future.

```
# cd /opt/web/mybase/
[/opt/web/mybase]# ls
[/opt/web/mybase]# java -jar /opt/jetty/jetty-distribution-9.1.0.v20131115/start.jar \
--add-to-start=deploy,http,logging
WARNING: deploy      initialised in ${jetty.base}/start.ini (appended)
WARNING: deploy      enabled in    ${jetty.base}/start.ini
WARNING: server     initialised in ${jetty.base}/start.ini (appended)
WARNING: server     enabled in    ${jetty.base}/start.ini
WARNING: http       initialised in ${jetty.base}/start.ini (appended)
WARNING: http       enabled in    ${jetty.base}/start.ini
WARNING: server     enabled in    ${jetty.base}/start.ini
WARNING: logging    initialised in ${jetty.base}/start.ini (appended)
WARNING: logging    enabled in    ${jetty.base}/start.ini
[/opt/web/mybase]# ls -F
start.ini  webapps/
```

At this point you have configured your /opt/web/mybase to enable the following modules:

deploy

This is the module that will perform deployment of web applications (WAR files or exploded directories), or Jetty IoC XML context deployables, from the /opt/web/mybase/webapps directory.

http

This sets up a single Connector that listens for basic HTTP requests.

See the created start.ini for configuring this connector

logging

When running Jetty as a service it is very important to have logging enabled. This module will enable the basic STDOUT and STDERR capture logging to the /opt/web/mybase/logs/ directory.

See [Using start.jar](#) for more details and options on setting up and configuring a \${jetty.base} directory.

Copy your war file into place.

```
# cp /home/user/projects/mywebsite.war /opt/web/mybase/webapps/
```

Most service installations will want jetty to run on port 80, now is your opportunity to change this from the default value of 8080 to 80.

Edit the /opt/web/mybase/start.ini and change the jetty.port value.

```
# grep jetty.port /opt/web/mybase/start.ini
jetty.port=80
```

Change Permissions

Change the permissions on the Jetty distribution, and your webapp directories so that the user you created can access it.

```
# chown --recursive jetty /opt/jetty
# chown --recursive jetty /opt/web/mybase
```

Configure the Service Layer

Next we need to make the Unix System aware that we have a new Jetty Service that can be managed by the standard service calls.

```
# cp /opt/jetty/jetty-distribution-9.1.0.v20131115/bin/jetty.sh /etc/init.d/jetty
# echo "JETTY_HOME=/opt/jetty/jetty-distribution-9.1.0.v20131115" > /etc/default/jetty
# echo "JETTY_BASE=/opt/web/mybase" >> /etc/default/jetty
# echo "TMPDIR=/opt/jetty/temp" >> /etc/default/jetty
```

Test out the configuration

```
# service jetty status
Checking arguments to Jetty:
START_INI      = /opt/web/mybase/start.ini
JETTY_HOME     = /opt/jetty/jetty-distribution-9.1.0.v20131115
JETTY_BASE     = /opt/web/mybase
JETTY_CONF     = /opt/jetty/jetty-distribution-9.1.0.v20131115/etc/jetty.conf
JETTY_PID      = /var/run/jetty.pid
JETTY_START    = /opt/jetty/jetty-distribution-9.1.0.v20131115/start.jar
JETTY_LOGS     = /opt/web/mybase/logs
CLASSPATH      =
JAVA           = /usr/bin/java
JAVA_OPTIONS   = -Djetty.state=/opt/web/mybase/jetty.state
                  -Djetty.logs=/opt/web/mybase/logs
                  -Djetty.home=/opt/jetty/jetty-distribution-9.1.0.v20131115
                  -Djetty.base=/opt/web/mybase
                  -Djava.io.tmpdir=/opt/jetty/temp
JETTY_ARGS     = jetty-logging.xml jetty-started.xml
RUN_CMD        = /usr/bin/java
                  -Djetty.state=/opt/web/mybase/jetty.state
                  -Djetty.logs=/opt/web/mybase/logs
                  -Djetty.home=/opt/jetty/jetty-distribution-9.1.0.v20131115
                  -Djetty.base=/opt/web/mybase
                  -Djava.io.tmpdir=/opt/jetty/temp
                  -jar /opt/jetty/jetty-distribution-9.1.0.v20131115/start.jar
jetty-logging.xml
jetty-started.xml
```

Start Your Service

You now have a configured \${jetty.base} in /opt/web/mybase and a jetty-distribution in /opt/jetty/jetty-distribution-9.2.1.v20140609, along with the service level files necessary to start the service.

Go ahead, start it.

```
# service jetty start
Starting Jetty: OK Wed Nov 20 12:35:28 MST 2013

# service jetty check
...(snip)..
Jetty running pid=2958

[/opt/web/mybase]# ps u 2958
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
jetty     2958  5.3  0.1 11179176 53984 ?        S1   12:46   0:00 /usr/bin/java -Djetty...
```

You should now have your server running. Try it out

Startup via Windows Service

There are no components that ship with the Jetty Distribution to make it a formal Windows Service.

However, we recommend the use of [Apache ProcRun's Daemon](#).

The techniques outlined here are based on Windows 7 (64-bit), using JDK 7 (64-bit), running on an Intel i7 architecture machine.

Prepare System

Prepare some empty directories to work with.

```
C:\> mkdir opt
C:\> cd opt
C:\opt> mkdir jetty
C:\opt> mkdir logs
C:\opt> mkdir myappbase
C:\opt> mkdir temp
C:\opt> dir
Volume in drive C has no label.
Volume Serial Number is DEAD-BEEF

Directory of C:\opt

11/21/2013  04:06 PM    <DIR>          .
11/21/2013  04:06 PM    <DIR>          ..
11/21/2013  04:06 PM    <DIR>          jetty
11/21/2013  04:06 PM    <DIR>          logs
11/21/2013  04:06 PM    <DIR>          myappbase
11/21/2013  04:06 PM    <DIR>          temp
11/21/2013  04:06 PM          0 File(s)      0 bytes
```

The directory purposes are as follows:

C:\opt

Where the service layer utilities, scripts, and binaries will eventually be

C:\opt\logs

Where the logs for the service layer will put its own logs.

Typically you will see the audit logs (install/update/delete), StdOutput, and StdError logs here.

C:\opt\jetty

Where the Jetty Distribution will be unpacked into

C:\opt\myappbase

Where your specific set of webapps will be located, including all of the configuration required of the server to make them operational.

C:\opt\temp

This is the temporary directory assigned to Java by the Service Layer (this is what Java sees as the `java.io.tmpdir` System Property)

This is intentionally kept separate from the standard temp directories of Windows, as this location doubles as the Servlet Spec work directory.

Make sure you have Java 7 Installed

Or download Java 7 from: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

```
C:\opt>java -version
java version "1.7.0_45"
Java(TM) SE Runtime Environment (build 1.7.0_45-b18)
Java HotSpot(TM) 64-Bit Server VM (build 24.45-b08, mixed mode)
```

Download and Unpack Distribution

Grab a copy of the ZIP distribution from the [Official Eclipse Download Site](#)

Open it up the downloaded Zip in Windows Explorer and drag the contents of the `jetty-distribution-9.2.1.v20140609` directory into place at `C:\opt\jetty`

Once you are complete, the contents of the C:\opt\jetty directory should look like this:

```
C:\opt\jetty>dir
Volume in drive C has no label.
Volume Serial Number is C8CF-820B

Directory of C:\opt\jetty

11/21/2013  12:13 PM    <DIR>      .
11/21/2013  12:13 PM    <DIR>      ..
11/21/2013  12:13 PM    <DIR>      bin
11/21/2013  12:13 PM    <DIR>      demo-base
11/21/2013  12:13 PM    <DIR>      etc
11/21/2013  12:13 PM    <DIR>      lib
11/21/2013  12:13 PM          30,012 license-eplv10-aslv20.html
11/21/2013  12:13 PM    <DIR>      logs
11/21/2013  12:13 PM    <DIR>      modules
11/21/2013  12:13 PM          6,262 notice.html
11/21/2013  12:13 PM          1,249 README.TXT
11/21/2013  12:13 PM    <DIR>      resources
11/21/2013  12:13 PM    <DIR>      start.d
11/21/2013  12:13 PM          2,126 start.ini
11/21/2013  12:13 PM          72,226 start.jar
11/21/2013  12:13 PM          341,784 VERSION.txt
11/21/2013  12:13 PM    <DIR>      webapps
               6 File(s)       453,659 bytes
11 Dir(s)   306,711,420,928 bytes free
```

Download and Unpack ProcRun

Download a copy of the [native binaries](#) of [Apache ProcRun](#).

You should have downloaded a file named commons-daemon-1.0.15-bin-windows.zip (version might be different). Open this ZIP file in Windows Explorer and drag prunmgr.exe and prunsrv.exe files into the C:\opt directory.

Once you are complete, the contents of C:\opt directory should look like this:

```
C:\opt> dir
Volume in drive C has no label.
Volume Serial Number is DEAD-BEEF

Directory of C:\opt

11/21/2013  04:06 PM    <DIR>      .
11/21/2013  04:06 PM    <DIR>      ..
11/21/2013  04:06 PM    <DIR>      jetty
11/21/2013  04:06 PM    <DIR>      logs
11/21/2013  04:06 PM    <DIR>      myappbase
11/21/2013  04:06 PM    <DIR>      temp
11/21/2013  04:11 PM          104,448 prunmgr.exe
11/21/2013  04:11 PM          80,896 prunsrv.exe
               2 File(s)       185,344 bytes
```

Configure your WebApp / Jetty Base

Now it's time to setup your new \${jetty.base} directory to have all of your WebApps and the configurations that they need.

We'll start by specifying which modules we want to use (this will create a start.ini file and also create a few empty directories for you)

```
C:\opt\myappbase>java -jar ..\jetty\start.jar --add-to-start=deploy,http,logging

WARNING: deploy      initialised in ${jetty.base}\start.ini (appended)
WARNING: deploy      enabled in      ${jetty.base}\start.ini
MKDIR: ${jetty.base}\webapps
WARNING: server      initialised in ${jetty.base}\start.ini (appended)
WARNING: server      enabled in      ${jetty.base}\start.ini
```

```

WARNING: http          initialised in ${jetty.base}\start.ini (appended)
WARNING: http          enabled in    ${jetty.base}\start.ini
WARNING: server         enabled in    ${jetty.base}\start.ini
WARNING: logging        initialised in ${jetty.base}\start.ini (appended)
WARNING: logging        enabled in    ${jetty.base}\start.ini
MKDIR: ${jetty.base}\logs

C:\opt\myappbase>dir
Volume in drive C has no label.
Volume Serial Number is C8CF-820B

Directory of C:\opt\myappbase

11/21/2013  12:49 PM    <DIR>          .
11/21/2013  12:49 PM    <DIR>          ..
11/21/2013  12:49 PM    <DIR>          logs
11/21/2013  12:49 PM            1,355 start.ini
11/21/2013  12:49 PM    <DIR>          webapps
               1 File(s)       1,355 bytes
               4 Dir(s)   306,711,064,576 bytes free

```

At this point you have configured your C:\opt\myappbase to enable the following modules:

deploy

This is the module that will perform deployment of web applications (WAR files or exploded directories), or Jetty IoC XML context deployables, from the C:\opt\myappbase\webapps directory.

http

This sets up a single Connector that listens for basic HTTP requests.

See the created `start.ini` for configuring this connector

logging

When running Jetty as a service it is very important to have logging enabled. This module will enable the basic STDOUT and STDERR capture logging to the C:\opt\myappbase\logs directory.

See [Using start.jar](#) for more details and options on setting up and configuring a \${jetty.base} directory.

At this point you merely have to copy your WAR files into the webapps directory.

```
c:\opt\myappbase> copy C:\projects\mywebsite.war webapps\
```

Configure Service Layer

At this point you should have your directories, Java, the Jetty distribution, and your webapp specifics setup and ready for operation.

We will use the [Apache ProcRun's prunsrv.exe](#) to install a Jetty Service.

The basic command line syntax is outlined in the link above.

A example `install-jetty-service.bat` is provided here as an example, based on the above directories.

```

@echo off
set SERVICE_NAME=JettyService
set JETTY_HOME=C:\opt\jetty
set JETTY_BASE=C:\opt\myappbase
set STOPKEY=secret
set STOPPORT=50001

set PR_INSTALL=C:\opt\prunsrv.exe

@REM Service Log Configuration

```

```
set PR_LOGPREFIX=%SERVICE_NAME%
set PR_LOGPATH=C:\opt\logs
set PR_STDOUTPUT=auto
set PR_STDError=auto
set PR_LOGLEVEL=Debug

@REM Path to Java Installation
set JAVA_HOME=C:\Program Files\Java\jdk1.7.0_45
set PR_JVM=%JAVA_HOME%\jre\bin\server\jvm.dll
set PR_CLASSPATH=%JETTY_HOME%\start.jar;%JAVA_HOME%\lib\tools.jar

@REM JVM Configuration
set PR_JVMMS=128
set PR_JVMMX=512
set PR_JVMSS=4000
set PR_JVMOPTIONS=-Duser.dir="%JETTY_BASE%" -Djava.io.tmpdir="C:\opt\temp" -Djetty.home="%JETTY_HOME%" -Djetty.base="%JETTY_BASE%"
@REM Startup Configuration
set JETTY_START_CLASS=org.eclipse.jetty.start.Main

set PR_STARTUP=auto
set PR_STARTMODE=java
set PR_STARTCLASS=%JETTY_START_CLASS%
set PR_STARTPARAMS=STOP.KEY="%STOPKEY%" ;STOP.PORT=%STOPPORT%

@REM Shutdown Configuration
set PR_STOPMODE=java
set PR_STOPCLASS=%JETTY_START_CLASS%
set PR_STOPPARAMS---stop;STOP.KEY="%STOPKEY%" ;STOP.PORT=%STOPPORT%;STOP.WAIT=10

%PR_INSTALL% //IS/%SERVICE_NAME% ^
--DisplayName="%SERVICE_NAME%" ^
--Install="%PR_INSTALL%" ^
--Startup="%PR_STARTUP%" ^
--LogPath="%PR_LOGPATH%" ^
--LogPrefix="%PR_LOGPREFIX%" ^
--LogLevel="%PR_LOGLEVEL%" ^
--StdOutput="%PR_STDOUTPUT%" ^
--StdError="%PR_STDError%" ^
--JavaHome="%JAVA_HOME%" ^
--Jvm="%PR_JVM%" ^
--JvmMs="%PR_JVMMS%" ^
--JvmMx="%PR_JVMMX%" ^
--JvmSs="%PR_JVMSS%" ^
--JvmOptions="%PR_JVMOPTIONS%" ^
--Classpath="%PR_CLASSPATH%" ^
--StartMode="%PR_STARTMODE%" ^
--StartClass="%JETTY_START_CLASS%" ^
--StartParams="%PR_STARTPARAMS%" ^
--StopMode="%PR_STOPMODE%" ^
--StopClass="%PR_STOPCLASS%" ^
--StopParams="%PR_STOPPARAMS%"

if not errorlevel 1 goto installed
echo Failed to install "%SERVICE_NAME%" service. Refer to log in %PR_LOGPATH%
goto end

:installed
echo The Service "%SERVICE_NAME%" has been installed

:end
```

Configuration's of note in this batch file:

SERVICE_NAME

This is the name of the service that Windows sees. The name in the Services window will show this name.

STOPKEY

This is the secret key (password) for the ShutdownMonitor, used to issue a formal command to stop the server.

STOPPORT

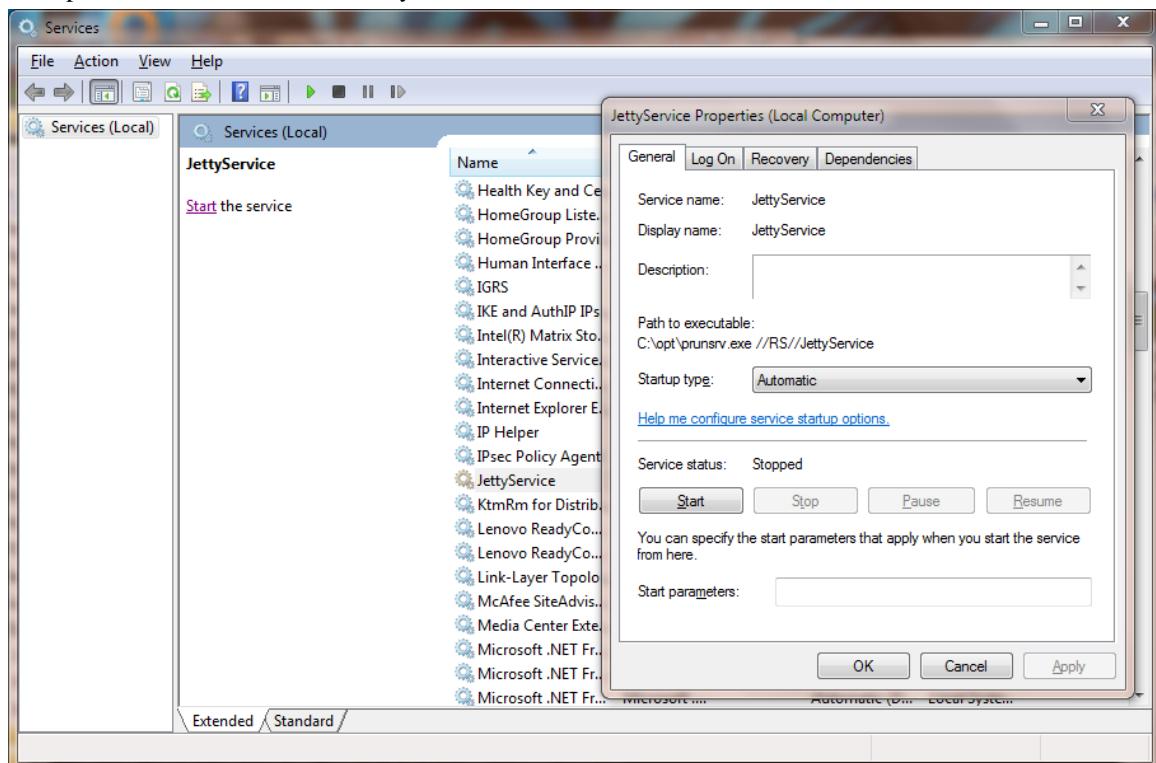
The port that the Shutdown Monitor listens on for the stop command.

If you have multiple Jetty servers on the same machine, this port will need to be different for each Service.

Once you have run `prunsrv.exe //IS/<service-name>` (done for you in the above batch file) to install the service, you can use the standard Windows utilities to manage (start/stop/restart) the Jetty service.

Start Your Service

Open the Service View and start your service.



Chapter 10. Session Management

Table of Contents

Setting Session Characteristics	141
Using Persistent Sessions	144
Session Clustering with a Database	146
Session Clustering with MongoDB	149

Setting Session Characteristics

To modify the session characteristics of a web application, you can use the following parameters, applying them as in one of the example configurations:

Using Init Parameters

Use these parameters to set session characteristics.

Table 10.1. Init Parameters

Context Parameter	Default Value	Description
org.eclipse.jetty.servlet.SessionCookieName	JSESSIONID	Session cookie name defaults to JSESSIONID, but can be set for a particular webapp with this context param.
org.eclipse.jetty.servlet.SessionIdPathParameterName	jsessionid	Session URL parameter name. Defaults to jsessionid, but can be set for a particular webapp with this context param. Set to "none" to disable URL rewriting.
org.eclipse.jetty.servlet.SessionDomain		Session Domain. If this property is set as a ServletContext param, then it is used as the domain for session cookies. If it is not set, then no domain is specified for the session cookie.
org.eclipse.jetty.servlet.SessionPath		Session Path. If this property is set as a ServletContext param, then it is used as the path for the session cookie. If it is not set, then the context path is used as the path for the cookie.
org.eclipse.jetty.servlet.MaxAge	-1	Session Max Age. If this property is set as a ServletContext param, then it is used as the max age for the session cookie. If it is not set, then a max age of -1 is used.
org.eclipse.jetty.servlet.CheckingRemoteSessionIdEncoding	false	If true, Jetty will add JSESSION-ID parameter even when encod-

Context Parameter	Default Value	Description
		ing external urls with calls to encodeURL(). False by default.

Applying Init Parameters

The following sections provide examples of how to apply the init parameters.

Context Parameter Example

You can set these parameters as context parameters in a web application's `WEB-INF/web.xml` file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-app_2_5.xsd"
  version="2.5">
  ...
  <context-param>
    <param-name>org.eclipse.jetty.servlet.SessionCookie</param-name>
    <param-value>XSESSIONID</param-value>
  </context-param>
  <context-param>
    <param-name>org.eclipse.jetty.servlet.SessionIdPathParameterName</param-name>
    <param-value>xsessionid</param-value>
  </context-param>
  ...
</web-app>
```

Web Application Examples

You can configure init parameters on a web application, either in code, or in a Jetty context xml file equivalent:

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath">/test</Set>
  <Set name="war"><SystemProperty name="jetty.home" default=". "/>/webapps/test</Set>
  ...

  <Call name="setInitParameter">
    <Arg>org.eclipse.jetty.servlet.SessionCookie</Arg>
    <Arg>XSESSIONID</Arg>
  </Call>
  <Call name="setInitParameter">
    <Arg>org.eclipse.jetty.servlet.SessionIdPathParameterName</Arg>
    <Arg>xsessionid</Arg>
  </Call>
</Configure>
```

SessionManager Examples

You can configure init parameters directly on a `SessionManager` instance, either in code or the equivalent in xml:

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath">/test</Set>
```

```

<Set name="war"><SystemProperty name="jetty.home" default=". "/>/webapps/test</Set>

...

<Get name="sessionHandler">
  <Set name="sessionManager">
    <New class="org.eclipse.jetty.server.session.HashSessionManager">
      <Set name="sessionCookie">xSESSIONID</Set>
      <Set name="sessionIdPathParameterName">xsessionid</Set>
    </New>
  </Set>
</Get>
</Configure>

```

Using Servlet 3.0 Session Configuration

With the advent of [Servlet Specification 3.0](#) there are new APIs for configuring session handling characteristics. What was achievable before only via jetty-specific [init-parameters](#) can now be achieved in a container-agnostic manner either in code, or via web.xml.

SessionCookieConfiguration

The [javax.servlet.SessionCookieConfig](#) class can be used to set up session handling characteristics. For full details, consult the [javadoc](#).

Here's an example of how you use it: this is a ServletContextListener that retrieves the SessionCookieConfig and sets up some new values for it when the context is being initialized:

```

import javax.servlet.SessionCookieConfig;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class TestListener implements ServletContextListener
{

    public void contextInitialized(ServletContextEvent sce)
    {
        String comment = "This is my special cookie configuration";
        String domain = "foo.com";
        String path = "/my/special/path";
        boolean isSecure = true;
        boolean httpOnly = false;
        int maxAge = 30000;
        String cookieName = "FOO_SESSION";

        SessionCookieConfig scf = sce.getServletContext().getSessionCookieConfig();

        scf.setComment(comment);
        scf.setDomain(domain);
        scf.setHttpOnly(httpOnly);
        scf.setMaxAge(maxAge);
        scf.setPath(path);
        scf.setSecure(isSecure);
        scf.setName(cookieName);
    }

    public void contextDestroyed(ServletContextEvent sce)
    {
    }
}

```

You can also use web.xml to configure the session handling characteristics instead: here's an example, doing exactly the same as we did above in code:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>

```

```

< xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-app_3_0.xsd"
  metadata-complete="true"
  version="3.0">

  <session-config>
    <comment>This is my special cookie configuration</comment>
    <domain>foo.com</domain>
    <http-only>false</http-only>
    <max-age>30000</max-age>
    <path>/my/special/path</path>
    <secure>true</secure>
    <name>FOO_SESSION</name>
  </session-config>
</web-app>

```

Session Tracking Modes

In addition to the configuration of [session cookies](#), since Servlet 3.0 you can also use the [javax.servlet.SessionTrackingMode](#) to configure session tracking.

To determine what are the *default* session tracking characteristics used by the container, call:

```
javax.servlet.SessionContext.getDefaultSessionTrackingModes();
```

This returns a java.util.Set of javax.servlet.SessionTrackingMode. The *default* session tracking modes for Jetty are:

- [SessionTrackingMode.COOKIE](#)
- [SessionTrackingMode.URL](#)

To see which session tracking modes are actually in effect for this Context, the following call returns a java.util.Set of javax.servlet.SessionTrackingMode:

```
javax.servlet.SessionContext.getEffectiveSessionTrackingModes();
```

To change the session tracking modes, call:

```
javax.servlet.SessionContext.setSessionTrackingModes(Set<SessionTrackingMode>);
```

You may also set the tracking mode in web.xml, eg:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-app_3_0.xsd"
  metadata-complete="true"
  version="3.0">

  <session-config>
    <tracking-mode>URL</tracking-mode>
    <tracking-mode>COOKIE</tracking-mode>
  </session-config>
</web-app>

```

Using Persistent Sessions

It is sometimes useful to preserve existing Sessions across restarts of Jetty. The [HashSessionManager](#) supports this feature. If you enable persistence, the HashSessionManager saves all existing, valid Sessions to disk before shutdown completes. On restart, Jetty restores the saved Sessions.

Enabling Persistence

A SessionManager does just what its name suggests—it manages the lifecycle and state of sessions on behalf of a webapp. Each webapp must have its own unique SessionManager instance. Enabling persistence is as simple as configuring the HashSessionManager as the SessionManager for a webapp and telling it where on disk to store the sessions:

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
    .
    .
    .
    <Set name="sessionHandler">
        <New class="org.eclipse.jetty.servlet.SessionHandler">
            <Arg>
                <New class="org.eclipse.jetty.servlet.HashSessionManager">
                    <Set name="storeDirectory">your/chosen/directory/goes/here</Set>
                </New>
            </Arg>
        </New>
    </Set>
    .
    .
    .
</Configure>
```



Tip

If you want to persist the sessions from multiple webapps:

1. Configure a separate HashSessionManager for each.
2. Assign to each a different value for 'storeDirectory'.

The above example uses a configuration file suitable for the [ContextProvider](#), thus you might want to check out ???.

Delaying Session Load

You might need to ensure that the sessions are loaded AFTER the servlet environment starts up (by default, Jetty eagerly loads sessions as part of the container startup, but before it initializes the servlet environment). For example, the Wicket web framework requires the servlet environment to be available when sessions are activated.

Using `SessionManager.setLazyLoad(true)`, Jetty loads sessions lazily either when it receives the first request for a session, or the session scavenger runs for the first time, whichever happens first. Here's how the configuration looks in XML:

```
<Set name="sessionHandler">
    <New class="org.eclipse.jetty.servlet.SessionHandler">
        <Arg>
            <New class="org.eclipse.jetty.servlet.HashSessionManager">
                <Set name="lazyLoad">true</Set>
            </New>
        </Arg>
    </New>
</Set>
```

Enabling Persistence for the Jetty Maven Plugin

To enable session persistence for the Jetty Maven plugin, set up the HashSessionManager in the configuration section like so:

```

<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>9.0.0.RC2 (or current version)</version>
  <configuration>
    <!-- ... -->
    <webAppConfig implementation="org.eclipse.jetty.maven.plugin.JettyWebAppContext">
      <defaultsDescriptor>${project.build.outputDirectory}/META-INF/webdefault.xml</
defaultsDescriptor>
      <contextPath>${jetty.contextRoot}</contextPath>
      <sessionHandler implementation="org.eclipse.jetty.server.session.SessionHandler">

      <sessionManager implementation="org.eclipse.jetty.server.session.HashSessionManager">
        <storeDirectory>${project.basedir}/target/jetty-sessions</storeDirectory>
        <idleSavePeriod>1</idleSavePeriod>
      </sessionManager>
    </sessionHandler>
  </webAppConfig>
  <!-- ... -->
</configuration>
</plugin>

```

Session Clustering with a Database

Jetty can support session clustering by persisting sessions to a shared database. Each Jetty instance locally caches sessions for which it has received requests, writing any changes to the session through to the database as the request exits the server. Sessions must obey the Serialization contract, and servlets must call the `Session.setAttribute()` method to ensure that changes are persisted.

The persistent session mechanism works in conjunction with a load balancer that supports stickiness. Stickiness can be based on various data items, such as source IP address or characteristics of the session ID or a load-balancer specific mechanism. For those load balancers that examine the session ID, the Jetty persistent session mechanism appends a node ID to the session ID, which can be used for routing.

In this type of solution, the database can become both a bottleneck and a single point of failure. Jetty takes steps to reduce the load on the database (discussed below), but in a heavily loaded environment you might need to investigate other optimization strategies such as local caching and database replication. You should also consult your database vendor's documentation for information on how to ensure high availability and failover of your database.

Configuration

There are two components to session management in Jetty: a session ID manager and a session manager.

- The session ID manager ensures that session IDs are unique across all webapps hosted on a Jetty instance, and thus there can only be one session ID manager per Jetty instance.
- The session manager handles the session lifecycle (create/update/invalidate/expire) on behalf of a web application, so there is one session manager per web application instance.

These managers also cooperate and collaborate with the `org.eclipse.jetty.server.session.SessionHandler` to enable cross-context dispatch.

Configuring the JDBCSessionIdManager

You need to configure an `org.eclipse.jetty.server.session.JDBCSessionIdManager` instance, either in embedded code or in a `jetty.xml` file. Here is an example of a `jetty.xml` setup:

```

<Set name="sessionIdManager">

    <New id="jdbcIdmgr" class="org.eclipse.jetty.server.session.JDBCSessionIdManager">
        <Arg>
            <Ref id="Server"/>
        </Arg>
        <Set name="workerName">fred</Set>
        <Set name="DatasourceName">javax.sql.DataSource/default</Set>
        <Set name="scavengeInterval">60</Set>
    </New>
</Set>
<Call name="setAttribute">
    <Arg>jdbcIdMgr</Arg>
    <Arg>
        <Ref id="jdbcIdmgr"/>
    </Arg>
</Call>

```

Notice that the JDBCSessionIdManager needs access to a database. The `jetty.xml` above configures it with the name of a `javax.sql.DataSource` that is defined elsewhere. Consult Jetty Naming Resources for more information on how to configure database access with Jetty. If you don't want to use a DataSource, you can configure JDBC Driver information instead. Here's an example:

```

<Set name="sessionIdManager">
    <New id="jdbcIdmgr" class="org.eclipse.jetty.server.session.JDBCSessionIdManager">
        <Arg>
            <Ref id="Server"/>
        </Arg>
        <Set name="workerName">fred</Set>
        <Call name="setDriverInfo">j
            <Arg>com.mysql.jdbc.Driver</Arg>
            <Arg>jdbc:mysql://127.0.0.1:3306/sessions?user=janb</Arg>
        </Call>
        <Set name="scavengeInterval">60</Set>
    </New>
</Set>
<Call name="setAttribute">
    <Arg>jdbcIdMgr</Arg>
    <Arg>
        <Ref id="jdbcIdmgr"/>
    </Arg>
</Call>

```

As Jetty configuration files are direct mappings of XML to Java, it is straightforward to see how to do this in code, but here's an example anyway:

```

Server server = new Server();
...
JDBCSessionIdManager idMgr = new JDBCSessionIdManager(server);
idMgr.setWorkerName("fred");
idMgr.setDriverInfo("com.mysql.jdbc.Driver", "jdbc:mysql://127.0.0.1:3306/sessions?
user=janb");
idMgr.setScavengeInterval(60);
server.setSessionIdManager(idMgr);

```

You must configure the JDBCSessionIdManager with a workerName that is unique across the cluster. Typically the name relates to the physical node on which the instance is executing. If this name is not unique, your load balancer might fail to distribute your sessions correctly.

You can also configure how often the persistent session mechanism sweeps the database looking for old, expired sessions with the scavengeInterval setting. The default value is 10mins. We recommend

that you not increase the frequency because doing so increases the load on the database with very little gain; old, expired sessions can harmlessly sit in the database.

Configuring the Database Schema

You may find it necessary to change the names of the tables and columns that the JDBC Session management uses to store the session information. The defaults used are:

Table 10.2. Default Values for Session Id Table

table name	JettySessionIds
columns	id

Table 10.3. Default Values for Session Table

table name	JettySessions
columns	rowId, sessionId, contextPath, virtualHost, lastNode, accessTime, lastAccessTime, createTime, cookieTime, lastSavedTime, expiryTime, maxInterval, map

To change these values, use the [org.eclipse.jetty.server.session.SessionIdTableSchema](#) and [org.eclipse.jetty.server.session.SessionTableSchema](#) classes. These classes have getter/setter methods for the table name and all columns.

Here's an example of changing the name of JettySessionsId table and its single column. This example will use java code, but as explained above, you may also do this via a jetty xml configuration file:

```
JDBCSessionIdManager idManager = new JDBCSessionIdManager(server);

SessionIdTableSchema idTableSchema = new SessionIdTableSchema();
idTableSchema.setTableName("mysessionids");
idTableSchema.setIdColumn("theid");
idManager.setSessionIdTableSchema(idTableSchema);
```

In a similar fashion, you can change the names of the table and columns for the JettySessions table. Note that both the SessionIdTableSchema and the SessionTableSchema instances are set on the JDBCSessionIdManager class.

```
JDBCSessionIdManager idManager = new JDBCSessionIdManager(server);

SessionTableSchema sessionTableSchema = new SessionTableSchema();
sessionTableSchema.setTableName("mysessions");
sessionTableSchema.setIdColumn("mysessionid");
sessionTableSchema.setAccessTimeColumn("atime");
sessionTableSchema.setContextPathColumn("cpath");
sessionTableSchema.setCookieTimeColumn("cooktime");
sessionTableSchema.setCreateTimeColumn("ctime");
sessionTableSchema.setExpiryTimeColumn("extime");
sessionTableSchema.setLastAccessTimeColumn("latime");
sessionTableSchema.setLastNodeColumn("lnode");
sessionTableSchema.setLastSavedTimeColumn("lstime");
sessionTableSchema.setMapColumn("mo");
sessionTableSchema.setMaxIntervalColumn("mi");
idManager.setSessionTableSchema(sessionTableSchema);
```

Configuring the JDBCSessionManager

The way you configure a JDBCSessionManager depends on whether you're configuring from a context xml file or a `jetty-web.xml` file or code. The basic difference is how you get a reference to the Jetty `org.eclipse.jetty.server.Server` instance.

From a context xml file, you reference the Server instance as a Ref:

```
<Ref name="Server" id="Server">
    <Call id="jdbcIdMgr" name="getAttribute">
        <Arg>jdbcIdMgr</Arg>
    </Call>
</Ref>

<Set name="sessionHandler">
    <New class="org.eclipse.jetty.server.session.SessionHandler">
        <Arg>
            <New id="jdbcmgr" class="org.eclipse.jetty.server.session.JDBCSessionManager">
                <Set name="idManager">
                    <Ref id="jdbcIdMgr"/>
                </Set>
            </New>
        </Arg>
    </New>
</Set>
```

From a WEB-INF/jetty-web.xml file, you can reference the Server instance directly:

```
<Get name="server">
    <Get id="jdbcIdMgr" name="sessionIdManager"/>
</Get>
<Set name="sessionHandler">
    <New class="org.eclipse.jetty.server.session.SessionHandler">
        <Arg>
            <New class="org.eclipse.jetty.server.session.JDBCSessionManager">
                <Set name="idManager">
                    <Ref id="jdbcIdMgr"/>
                </Set>
            </New>
        </Arg>
    </New>
</Set>
```

If you're embedding this in code:

```
//assuming you have already set up the JDBCSessionIdManager as shown earlier
//and have a reference to the Server instance:

WebApplicationContext wac = new WebApplicationContext();
... //configure your webapp context
JDBCSessionManager jdbcMgr = new JDBCSessionManager();
jdbcMgr.setIdManager(server.getSessionIdManager());
wac.setSessionHandler(jdbcMgr);
```

Session Clustering with MongoDB

Jetty can support session clustering by persisting sessions into [MongoDB](#). Each Jetty instance locally caches sessions for which it has received requests, writing any changes to the session through to the cluster as the request exits the server. Sessions must obey the Serialization contract, and servlets must call the Session.setAttribute() method to ensure that changes are persisted.

The persistent session mechanism works in conjunction with a load balancer that supports stickiness. Stickiness can be based on various data items, such as source IP address or characteristics of the session ID or a load-balancer specific mechanism. For those load balancers that examine the session ID, the Jetty persistent session mechanism appends a node ID to the session ID, which can be used for routing.

In this type of solution, the traffic on the network needs to be carefully watched and tends to be the bottleneck. You are probably investigating this solution in order to scale to large amount of users and sessions, so careful attention should be paid to your usage scenario. Applications with a heavy write profile to their sessions will consume more network bandwidth than profiles that are predominately read oriented. We recommend using this session manager with largely read based session scenarios.

Configuration

There are two components to session management in Jetty: a session ID manager and a session manager.

- The session ID manager ensures that session IDs are unique across all webapps hosted on a Jetty instance, and thus there should only be one session ID manager per Jetty instance.
- The session manager handles the session lifecycle (create/update/invalidate/expire) on behalf of a web application, so there is one session manager per web application instance.

These managers also cooperate and collaborate with the `org.eclipse.jetty.server.session.SessionHandler` to enable cross-context dispatch.

Configuring the MongoDBSessionIdManager

You need to configure an [org.eclipse.jetty.nosql.mongodb.MongoSessionIdManager](http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/nosql/mongodb/MongoSessionIdManager.html) instance, either in embedded code or in a jetty.xml file. Here is an example of a jetty.xml setup:

```
<Set name="sessionIdManager">

<New id="mongoIdMgr" class="org.eclipse.jetty.nosql.mongodb.MongoSessionIdManager">
    <Arg>
        <Ref id="Server"/>
    </Arg>
    <Set name="workerName">fred</Set>
    <Set name="scavengeInterval">60</Set>
</New>
</Set>
<Call name="setAttribute">
    <Arg>mongoIdMgr</Arg>
    <Arg>
        <Ref id="mongoIdMgr"/>
    </Arg>
</Call>
```

Notice that the MongoSessionIdManager needs access to a mongodb cluster. The default settings are for localhost and default port for mongo db.

The more complex setup is to configure with remote mongodb instances, this is an example of how that would look.

```
<New id="mongodb" class="com.mongodb.Mongo">
    <Arg>
        <New class="java.util.ArrayList">
            <Call name="add">
                <Arg>
                    <New class="com.mongodb.ServerAddress">
                        <Arg type="java.lang.String">foo.example.com</Arg>
                        <Arg type="int">27017</Arg>
                </New>
            </Call>
        </New>
    </Arg>
```

```

        </New>
        </Arg>
    </Call>
    <!-- Add more Call statements here as desired --> </New>
</Arg>

<Call name="getDB">
    <Arg>HttpSessions</Arg>
    <Call id="sessionDocument" name="getCollection">
        <Arg>sessions</Arg>
    </Call>
</Call>
<!-- If you want to configure Jetty to be able to read through the slaves, call the
following: -->
<Call name="slaveOk"/>
</New>

<Set name="sessionIdManager">
    <New id="mongoIdMgr" class="org.eclipse.jetty.nosql.mongodb.MongoSessionIdManager">
        <Arg>
            <Ref id="Server"/>
        </Arg>
        <Arg>
            <Ref id="sessionDocument"/>
        </Arg>
        <Set name="workerName">fred</Set>
        <Set name="scavengePeriod">60</Set>
    </New>
</Set>

```

As Jetty configuration files are direct mappings of XML to Java, it is straightforward to see how to do this in code, but here's an example anyway:

```

Server server = new Server();
...
MongoSessionIdManager idMgr = newMongoSessionIdManager(server);
idMgr.setWorkerName("fred");
idMgr.setScavengePeriod(60);
server.setSessionIdManager(idMgr);

```

The MongoDB Session Id Manager has slightly different options than some of our more traditional session options. The mongodb session id manager has the same scavenge timers which govern the setting of a valid session to invalid after a certain period of inactivity. New to this session id manager is the extra purge setting which governs removal from the mongodb cluster. This can be configured through the 'purge' option. Purge is by default set to true and by default runs daily for each node on the cluster. Also able to be configured is the age in which an invalid session will be retained which is set to 1 day by default. This means that invalid sessions will be removed after lingering in the mongodb instance for a day. There is also an option for purging valid sessions that have not been used recently. The default time for this is 1 week. You can disable these behaviors by setting purge to false.

You must configure the MongoSessionIdManager with a workerName that is unique across the cluster. Typically the name relates to the physical node on which the instance is executing. If this name is not unique, your load balancer might fail to distribute your sessions correctly.

You can also configure how often the persistent session mechanism sweeps the mongo cluster looking for old, expired sessions with the scavengePeriod setting. The default value is 60 seconds. We recommend that you not increase the frequency because doing so increases the load on the database with very little gain; old, expired sessions can harmlessly sit in the database.

scavengedelay

How long to delay before periodically looking for sessions to scavenge?

scavengeperiod

How much time after a scavenge has completed should you wait before doing it again?

purge (Boolean)

Do you want to purge (delete) sessions that are invalid from the session store completely?

purgeDelay

How often do you want to perform this purge operation?

purgeInvalidAge

How old should an invalid session be before it is eligible to be purged?

purgeValidAge

How old should a valid session be before it is eligible to be marked invalid and purged? Should this occur at all?

preserveOnStop

Whether or not to retain all sessions when the session manager stops. Default is true.

Configuring a MongoSessionManager

The way you configure a [org.eclipse.jetty.nosql.mongodb.MongoSessionManager](#) depends on whether you're configuring from a [context xml](#) file or a [jetty-web.xml](#) file or code. The basic difference is how you get a reference to the Jetty [org.eclipse.jetty.server.Server](#) instance.

From a context xml file, you reference the Server instance as a Ref:

```
<Ref name="Server" id="Server">
    <Call id="mongoIdMgr" name="getSessionIdManager"/>
</Ref>

<Set name="sessionHandler">
    <New class="org.eclipse.jetty.server.session.SessionHandler">
        <Arg>
            <New id="mongoMgr" class="org.eclipse.jetty.nosql.mongodb.MongoSessionManager">
                <Set name="idManager">
                    <Ref id="mongoIdMgr"/>
                </Set>
            </New>
        </Arg>
    </New>
</Set>
```

From a WEB-INF/jetty-web.xml file, you can reference the Server instance directly:

```
<Get name="server">
    <Get id="mongoIdMgr" name="sessionIdManager"/>
</Get>
<Set name="sessionHandler">
    <New class="org.eclipse.jetty.server.session.SessionHandler">
        <Arg>
            <New class="org.eclipse.jetty.nosql.mongodb.MongoSessionManager">
                <Set name="idManager">
                    <Ref id="mongoIdMgr"/>
                </Set>
            </New>
        </Arg>
    </New>
</Set>
```

If you're embedding this in code:

```
//assuming you have already set up the MongoSessionIdManager as shown earlier
//and have a reference to the Server instance:

WebApplicationContext wac = new WebApplicationContext();
... //configure your webapp context
MongoSessionManager mongoMgr = new MongoSessionManager();
mongoMgr.setIdManager(server.getSessionIdManager());
wac.setSessionHandler(mongoMgr);
```

Chapter 11. Configuring JNDI

Table of Contents

Quick Setup	154
Detailed Setup	154
Working with Jetty JNDI	156
Configuring JNDI	159
Using JNDI with Jetty Embedded	163
Datasource Examples	165

Jetty supports `java:comp/env` lookups in webapps. This is an optional feature for which you need to do some setup.

Quick Setup

If you are using the standard distribution of Jetty, and want to enable JNDI for *all* your webapps, edit the `$JETTY_HOME/start.ini` file and uncomment the following lines:

```
# =====
# Enable JNDI
#
OPTIONS=jndi

# =====
# Enable additional webapp environment configurators
#
OPTIONS=plus
etc/jetty-plus.xml
```

You can now start Jetty and use JNDI within your webapps. See [Using JNDI](#) for information on how to add entries to the JNDI environment that Jetty can look up within webapps.

Detailed Setup

The following sections describe the finer points of configuring JNDI in Jetty.

Setting Up the List of Configurations

When deploying a webapp, Jetty has an extensible list of [configurations](#) that it applies to the webapp in a specific order. These configurations do things like parse the `web.xml`, set up the classpath for the webapp, and parse jetty specific `WEB-INF/jetty-web.xml` files.

To use JNDI with Jetty, you need additional attributes that do things like read `WEB-INF/jetty-env.xml`, set up a `java:comp/env` context, and hook up JNDI entries from the environment into your webapp. The list below shows the two extra elements and the order you must use when you add them:

```
<Array id="plusConfig" type="java.lang.String">
<Item>org.eclipse.jetty.webapp.WebInfConfiguration</Item>
<Item>org.eclipse.jetty.webapp.WebXmlConfiguration</Item>
```

```

<Item>org.eclipse.jetty.webapp.MetaInfConfiguration</Item>
<Item>org.eclipse.jetty.webapp.FragmentConfiguration</Item>
<Item>org.eclipse.jetty.plus.webapp.EnvConfiguration</Item>      <!-- add for jndi -->
<Item>org.eclipse.jetty.plus.webapp.PlusConfiguration</Item>      <!-- add for jndi -->
<Item>org.eclipse.jetty.webapp.JettyWebXmlConfiguration</Item>
</Array>

```

This augmented list of configurations for JNDI is predefined for you in the `etc/jetty-plus.xml` file:

The [Quick Setup](#) section shows you how to enable it. Be aware that the `etc/jetty-plus.xml` file enables JNDI for **all** webapps for a given Server instance.

Now skip down to [Adding JNDI Implementation Jars to the Jetty Classpath](#) if you want to use JNDI with all webapps. If you only want to use JNDI with specific webapps, read on.

Enabling JNDI for a Single WebApp

If you have only a few webapps that you want to use with JNDI, you can apply the augmented list of configurations specifically to those webapps. To do that, create a context XML file for each webapp, and set up the configuration classes. Here's an example of how that should look:

```

<Configure id='wac' class="org.eclipse.jetty.webapp.WebAppContext">
  <Array id="plusConfig" type="java.lang.String">
    <Item>org.eclipse.jetty.webapp.WebInfConfiguration</Item>
    <Item>org.eclipse.jetty.webapp.WebXmlConfiguration</Item>
    <Item>org.eclipse.jetty.webapp.MetaInfConfiguration</Item>
    <Item>org.eclipse.jetty.webapp.FragmentConfiguration</Item>
    <Item>org.eclipse.jetty.plus.webapp.EnvConfiguration</Item>      <!-- add for JNDI -->
    <Item>org.eclipse.jetty.plus.webapp.PlusConfiguration</Item>      <!-- add for JNDI -->
    <Item>org.eclipse.jetty.webapp.JettyWebXmlConfiguration</Item>
  </Array>

  <Set name="war"><SystemProperty name="jetty.home" default=". "/>/webapps/my-cool-
  webapp</Set>
  <Set name="configurationClasses"><Ref refid="plusConfig"/></Set>
</Configure>

```

Enabling JNDI for a Deployer

Another alternative to enabling JNDI for all webapps (see [Quick Setup](#), or just individual webapps (see [Enabling JNDI for a Single WebApp](#)) is to create a specific Deployer instance that enables JNDI for all webapps it deploys. That way, you can [set up](#) the configurations once to apply them to every webapp you drop into the nominated directory; in this way you keep your JNDI-enabled webapps separate from the non-JNDI enabled webapps. The `etc/jetty-plus.xml` file has an example of this (commented out):

```

<!-- ===== -->
<!-- Apply plusConfig to all webapps in webapps-plus -->
<!-- ===== -->
<!-- Uncomment the following to set up a deployer that will -->
<!-- deploy webapps from a directory called webapps-plus. Note -->
<!-- that you will need to create this directory first! -->
<Ref refid="DeploymentManager">
  <Call name="addAppProvider">
    <Arg>
      <New class="org.eclipse.jetty.deploy.providers.WebAppProvider">
        <Set name="monitoredDirName"><Property name="jetty.home" default=". "/>/
        webapps-plus</Set>
    </Arg>
  </Call>
</Ref>

```

```

<Set name="defaultsDescriptor"><Property name="jetty.home" default=". "/>/etc/
webdefault.xml</Set>
<Set name="scanInterval">5</Set>
<Set name="parentLoaderPriority">false</Set>
<Set name="extractWars">true</Set>
<Set name="configurationClasses"><Ref refid="plusConfig"/></Set>
</New>
</Arg>
</Call>
</Ref>
```

This creates a new deployer that looks for things to deploy in the webapps-plus directory and sets up the JNDI configurations for them.



Note

If you do this, you need to either supply the etc/jetty-plus.xml file last on the runline, or move it to after the etc/jetty-deploy.xml file reference in start.ini. This is because this section of the etc/jetty-plus.xml file refers to the DeploymentManager, which is created in the etc/jetty-deploy.xml file.

Adding JNDI Implementation Jars to the Jetty Classpath

Now that you have the JNDI configuration for the webapp/s set up, you need to ensure that the JNDI implementation Jars are on the Jetty classpath. These jars are optional, so are not there by default. You add these into the classpath by using startup time OPTIONS.

In the [Quick Setup](#) section, we showed you how to use start.ini to accomplish this in a permanent fashion. Instead, you can supply OPTION on the command line so it is valid for that run only. Here's how to do it:

```
$ java -jar start.jar OPTIONS=jndi,plus
```

Be aware that if you haven't set up the configurations that enable JNDI for individual webapps, you need to use the etc/jetty-plus.xml file, which you can also do on the runline:

```
$ java -jar start.jar OPTIONS=jndi,plus etc/jetty-plus.xml
```

Working with Jetty JNDI

Defining the web.xml

You can configure naming resources to reference in a web.xml file and access from within the java:comp/env naming environment of the webapp during execution. Specifically, you can configure support for the following web.xml elements:

```
<env-entry/>
<resource-ref/>
<resource-env-ref/>
```

[Configuring env-entries](#) shows you how to set up overrides for `env-entry` elements in `web.xml`, while [Configuring resource-refs](#) and [resource-env-refs](#) discusses how to configure support resources such as `javax.sql.DataSource`.

You can also plug a JTA `javax.transaction.UserTransaction` implementation into Jetty so that webapps can look up `java:comp/UserTransaction` to obtain a distributed transaction manager: see [Configuring XA Transactions](#).

Declaring Resources

You must declare the objects you want bound into the Jetty environment so that you can then hook into your webapp via `env-entry`, `resource-ref` and `resource-env-refs` in `web.xml`. You create these bindings by using declarations of the following types:

- `org.eclipse.jetty.plus.jndi.EnvEntry`**
for `env-entry` type of entries
- `org.eclipse.jetty.plus.jndi.Resource`**
for all other type of resources
- `org.eclipse.jetty.plus.jndi.Transaction`**
for a JTA manager
- `org.eclipse.jetty.plus.jndi.Link`**
for link between a `web.xml` resource name and a naming entry

Declarations of each of these types follow the same general pattern:

```
<New class="org.eclipse.jetty.plus.jndi.xxxx">
  <Arg><!-- scope --></Arg>
  <Arg><!-- name --></Arg>
  <Arg><!-- value --></Arg>
</New>
```

You can place these declarations into three different files, depending on your needs and the `scope` of the resources being declared.

Deciding Where to Declare Resources

You can define naming resources in three places:

jetty.xml

Naming resources defined in a `jetty.xml` file are [scoped](#) at either the JVM level or the Server level. The classes for the resource must be visible at the Jetty container level. If the classes for the resource only exist inside your webapp, you must declare it in a `WEB-INF/jetty-env.xml` file.

WEB-INF/jetty-env.xml

Naming resources in a `WEB-INF/jetty-env.xml` file are [scoped](#) to the web app in which the file resides. While you can enter JVM or Server scopes if you choose, we do not recommend doing

so. The resources defined here may use classes from inside your webapp. This is a Jetty-specific mechanism.

context xml file

Entries in a context xml file should be [scoped](#) at the level of the webapp to which they apply, although you can supply a less strict scoping level of Server or JVM if you choose. As with resources declared in a `jetty.xml` file, classes associated with the resource must be visible on the container's classpath.

Scope of Resource Names

Naming resources within Jetty belong to one of three different scopes, in increasing order of restrictiveness:

JVM scope

The name is unique across the JVM instance, and is visible to all application code. You represent this scope by a null first parameter to the resource declaration. For example:

```
<New id="cf" class="org.eclipse.jetty.plus.jndi.Resource">
  <Arg></Arg>  <!-- empty arg -->
  <Arg>jms/ConnectionFactory</Arg>
  <Arg>
    <New class="org.apache.activemq.ActiveMQConnectionFactory">
      <Arg>vm://localhost?broker.persistent=false</Arg>
    </New>
  </Arg>
</New>
```

Server scope

The name is unique to a Server instance, and is only visible to code associated with that instance. You represent this scope by referencing the Server instance as the first parameter to the resource declaration. For example:

```
<Configure id="Server" class="org.eclipse.jetty.Server">
  <New id="cf" class="org.eclipse.jetty.plus.jndi.Resource">
    <Arg><Ref refid="Server"/></Arg>  <!-- reference to Server instance -->
    <Arg>jms/ConnectionFactory</Arg>
    <Arg>
      <New class="org.apache.activemq.ActiveMQConnectionFactory">
        <Arg>vm://localhost?broker.persistent=false</Arg>
      </New>
    </Arg>
  </New>
</Configure>
```

Webapp scope

The name is unique to the WebAppContext instance, and is only visible to code associated with that instance. You represent this scope by referencing the WebAppContext instance as the first parameter to the resource declaration. For example:

```
<Configure id='wac' class="org.eclipse.jetty.webapp.WebAppContext">
  <New id="cf" class="org.eclipse.jetty.plus.jndi.Resource">
    <Arg><Ref refid='wac'/></Arg>  <!-- reference to WebAppContext -->
    <Arg>jms/ConnectionFactory</Arg>
    <Arg>
      <New class="org.apache.activemq.ActiveMQConnectionFactory">
```

```
<Arg>vm://localhost?broker.persistent=false</Arg>
</New>
</Arg>
</New>
</Configure>
```

What Can Be Bound as a Resource?

You can bind four types of objects into a Jetty JNDI reference:

- An ordinary POJO instance.
- A [javax.naming.Reference](#) instance.
- An object instance that implements the [javax.naming.Referenceable](#) interface.
- A link between a name as referenced in web.xml and as referenced in the Jetty environment.

Configuring JNDI

Configuring JNDI env-entries

Sometimes it is useful to pass configuration information to a webapp at runtime that you either cannot or cannot conveniently code into a web.xml env-entry. In such cases, you can use `org.eclipse.jetty.plus.jndi.EnvEntry`, and even override an entry of the same name in web.xml.

```
<New class="org.eclipse.jetty.plus.jndi.EnvEntry">
<Arg></Arg>
<Arg>mySpecialValue</Arg>
<Arg type="java.lang.Integer">4000</Arg>
<Arg type="boolean">true</Arg>
</New>
```

This example defines a virtual env-entry called `mySpecialValue` with value `4000` that is [scoped](#) to the JVM. It is put into JNDI at `java:comp/env/mySpecialValue` for *every* web app deployed. Moreover, the boolean argument indicates that this value overrides an env-entry of the same name in web.xml. If you don't want to override, omit this argument, or set it to `false`.

The Servlet Specification allows binding only the following object types to an env-entry:

- `java.lang.String`
- `java.lang.Integer`
- `java.lang.Float`
- `java.lang.Double`
- `java.lang.Long`
- `java.lang.Short`
- `java.lang.Character`
- `java.lang.Byte`
- `java.lang.Boolean`

That being said, Jetty is a little more flexible and allows you to also bind custom POJOs, [javax.naming.References](#) and [javax.naming.Referenceables](#). Be aware that if you take advantage of this feature, your web application is *not portable*.

To use the env-entry configured above, use code in your servlet/filter/etc., such as:

```
import javax.naming.InitialContext;

public class MyClass {

    public void myMethod() {

        InitialContext ic = new InitialContext();
        Integer mySpecialValue = (Integer)ic.lookup("java:comp/env/mySpecialValue");
        ...
    }
}
```

Configuring resource-refs and resource-env-refs

You can configure any type of resource that you want to refer to in a web.xml file as a resource-ref or resource-env-ref, using the org.eclipse.jetty.plus.jndi.Resource type of naming entry. You provide the scope, the name of the object (relative to java:comp/env) and a POJO instance or a javax.naming.Reference instance or javax.naming.Referenceable instance.

The [J2EE Specification](#) recommends storing DataSources in java:comp/env/jdbc, JMS connection factories under java:comp/env/jms, JavaMail connection factories under java:comp/env/mail and URL connection factories under java:comp/env/url. For example:

Table 11.1. DataSource Declaration Conventions

Resource Type	Name in jetty.xml	Environment Lookup
javax.sql.DataSource	jdbc/myDB	java:comp/env/jdbc/myDB
javax.jms.QueueConnectionFactory	jms/myQueue	java:comp/env/jms/myQueue
javax.mail.Session	mail/myMailService	java:comp/env/mail/myMailService

Configuring DataSources

Here is an example of configuring a javax.sql.DataSource. Jetty can use any DataSource implementation available on its classpath. In this example, the DataSource is from the [Derby](#) relational database, but you can use any implementation of a javax.sql.DataSource. This example configures it as scoped to a web app with the id of *wac*:

```
<Configure id='wac' class="org.eclipse.jetty.webapp.WebAppContext">
<New id="myds" class="org.eclipse.jetty.plus.jndi.Resource">
<Arg><Ref refid="wac"/></Arg>
<Arg>jdbc/myds</Arg>
<Arg>
<New class="org.apache.derby.jdbc.EmbeddedDataSource">
<Set name="DatabaseName">test</Set>
<Set name="createDatabase">create</Set>
</New>
</Arg>
</New>
</Configure>
```

The code above creates an instance of org.apache.derby.jdbc.EmbeddedDataSource, calls the two setter methods `setDatabaseName("test")`, and `setCreateDatabase("create")`, and binds it into the JNDI scope for the web app. If you do

not have the appropriate `resource-ref` set up in your `web.xml`, it is available from application lookups as `java:comp/env/jdbc/myds`.

Here's an example `web.xml` declaration for the datasource above:

```
<resource-ref>
  <res-ref-name>jdbc/myds</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

To look up your `DataSource` in your `servlet/filter/etc..:`

```
import javax.naming.InitialContext;
import javax.sql.DataSource;

public class MyClass {

    public void myMethod() {

        InitialContext ic = new InitialContext();
        DataSource myDS = (DataSource)ic.lookup("java:comp/env/jdbc/myds");

        ...
    }
}
```

Note



Careful! When configuring Resources, ensure that the type of object you configure matches the type of object you expect to look up in `java:comp/env`. For database connection factories, this means that the object you register as a Resource *must* implement the `javax.sql.DataSource` interface.

For more examples of datasource configurations, see the section called “Datasource Examples”.

Configuring JMS Queues, Topics and ConnectionFactories

Jetty can bind any implementation of the JMS destinations and connection factories. You just need to ensure the implementation Jars are available on Jetty's classpath. Here is an example of binding an [ActiveMQ](#) in-JVM connection factory:

```
<Configure id='wac' class="org.eclipse.jetty.webapp.WebAppContext">
  <New id="cf" class="org.eclipse.jetty.plus.jndi.Resource">
    <Arg><Ref refid='wac'/></Arg>
    <Arg>jms/connectionFactory</Arg>
    <Arg>
      <New class="org.apache.activemq.ActiveMQConnectionFactory">
        <Arg>vm://localhost?broker.persistent=false</Arg>
      </New>
    </Arg>
  </New>
</Configure>
```

The entry in `web.xml` would be:

```
<resource-ref>
  <res-ref-name>jms/connectionFactory</res-ref-name>
  <res-type>javax.jms.ConnectionFactory</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

TODO: put in an example of a QUEUE from progress demo

Configuring Mail

Jetty also provides infrastructure for access to `javax.mail.Session`s from within an application:

```
<Configure id='wac' class="org.eclipse.jetty.webapp.WebAppContext">
<New id="mail" class="org.eclipse.jetty.plus.jndi.Resource">
<Arg><Ref refid="wac"/></Arg>
<Arg>mail/Session</Arg>
<Arg>
<New class="org.eclipse.jetty.jndi.factories.MailSessionReference">
<Set name="user">fred</Set>
<Set name="password">OBF:1xmk1w261z0f1wlclxmq</Set>
<Set name="properties">
<New class="java.util.Properties">
<Put name="mail.smtp.host">XXX</Put>
<Put name="mail.from">me@me</Put>
<Put name="mail.debug">true</Put>
</New>
</Set>
</New>
</Arg>
</New>
</Configure>
```

This setup creates an instance of the `org.eclipse.jetty.jndi.factories.MailSessionReference` class, calls its setter methods to set up the authentication for the mail system, and populates a set of Properties, setting them on the `MailSessionReference` instance. The result is that an application can look up `java:comp/env/mail/Session` at runtime and obtain access to a `javax.mail.Session` that has the necessary configuration to permit it to send email via SMTP.



Tip

You can set the password to be plain text, or use Jetty's [Secure Password Obfuscation](#) (OBF:) mechanism to make the config file a little more secure from prying eyes. Remember that you cannot use the other Jetty encryption mechanisms of MD5 and Crypt because they do not allow you to recover the original password, which the mail system requires.

Configuring XA Transactions

If you want to perform distributed transactions with your resources, you need a *transaction manager* that supports the JTA interfaces, and that you can look up as `java:comp/UserTransaction` in your webapp. Jetty does not ship with one as standard, but you can plug in the one you prefer. You can configure a transaction manager using the [JNDI Transaction](#) object in a Jetty config file. The following example configures the [Atomikos](#) transaction manager:

```
<New id="tx" class="org.eclipse.jetty.plus.jndi.Transaction">
<Arg>
<New class="com.atomikos.icatch.jta.J2eeUserTransaction"/>
</Arg>
</New>
```

Configuring Links

Generally, the name you set for your `Resource` should be the same name you use for it in `web.xml`. For example:

In a context xml file:

```
<Configure id='wac' class="org.eclipse.jetty.webapp.WebAppContext">
<New id="myds" class="org.eclipse.jetty.plus.jndi.Resource">
<Arg><Ref refid="wac"/></Arg>
```

```
<Arg>jdbc/mydatasource</Arg>
<Arg>
<New class="org.apache.derby.jdbc.EmbeddedDataSource">
    <Set name="DatabaseName">test</Set>
    <Set name="createDatabase">create</Set>
</New>
</Arg>
</New>
</Configure>
```

In web.xml:

```
<resource-ref>
    <res-ref-name>jdbc/mydatasource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <injection-target>
        <injection-target-class>com.acme.JNDITest</injection-target-class>
        <injection-target-name>myDatasource</injection-target-name>
    </injection-target>
</resource-ref>
```

However, you can refer to it in web.xml by a different name, and link it to the name in your org.eclipse.jetty.plus.jndi.Resource by using an org.eclipse.jetty.plus.jndi.Link. For the example above, you can refer to the jdbc/mydatasource resource as jdbc/mydatasource1 as follows:

In a context xml file declare jdbc/mydatasource:

```
<Configure id='wac' class="org.eclipse.jetty.webapp.WebAppContext">
    <New id="myds" class="org.eclipse.jetty.plus.jndi.Resource">
        <Arg><Ref refid="wac"/></Arg>
        <Arg>jdbc/mydatasource</Arg>
        <Arg>
            <New class="org.apache.derby.jdbc.EmbeddedDataSource">
                <Set name="DatabaseName">test</Set>
                <Set name="createDatabase">create</Set>
            </New>
        </Arg>
    </New>
</Configure>
```

Then in a WEB-INF/jetty-env.xml file, link the name jdbc/mydatasource to the name you want to reference it as in web.xml, which in this case is jdbc/mydatasource1:

```
<New id="map1" class="org.eclipse.jetty.plus.jndi.Link">
    <Arg><Ref refid='wac'/></Arg>
    <Arg>jdbc/mydatasource1</Arg> <!-- name in web.xml -->
    <Arg>jdbc/mydatasource</Arg> <!-- name in container environment -->
</New>
```

Now you can refer to jdbc/mydatasource1 in the web.xml like this:

```
<resource-ref>
    <res-ref-name>jdbc/mydatasource1</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <injection-target>
        <injection-target-class>com.acme.JNDITest</injection-target-class>
        <injection-target-name>myDatasource</injection-target-name>
    </injection-target>
</resource-ref>
```

This can be useful when you cannot change a JNDI resource directly in the web.xml but need to link it to a specific resource in your deployment environment.

Using JNDI with Jetty Embedded

Setting up the Classpath

In addition to the jars that you require for your application, and the jars needed for core Jetty, you will need to place the following jars onto your classpath:

```
jetty-jndi.jar
jetty-plus.jar
```

If you are using transactions, you will also need the javax.transaction api. You can [obtain this jar from the Jetty dependencies site](#).

If you wish to use mail, you will also need the javax.mail api and implementation. You can [obtain this jar from the Jetty dependencies site](#). Note that this jar also requires the javax.activation classes, which you can also [obtain](#) from the [Jetty dependencies site](#).

Example Code

Here is an example class that sets up some JNDI entries and deploys a webapp that references these JNDI entries in code. We'll use some mocked up classes for the transaction manager and the Data-Source in this example for simplicity:

```
import java.util.Properties;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.webapp.WebAppContext;

/**
 * ServerWithJNDI
 *
 */
public class ServerWithJNDI
{
    public static void main(String[] args) throws Exception
    {

        //Create the server
        Server server = new Server(8080);

        //Enable parsing of jndi-related parts of web.xml and jetty-env.xml
        org.eclipse.jetty.webapp.Configuration.ClassList classlist =
        org.eclipse.jetty.webapp.Configuration.ClassList.setServerDefault(server);

        classlist.addAfter("org.eclipse.jetty.webapp.FragmentConfiguration", "org.eclipse.jetty.plus.webapp.EnvConfig");

        //Create a WebApp
        WebAppContext webapp = new WebAppContext();
        webapp.setContextPath("/");
        webapp.setWar("./my-foo-webapp.war");
        server.setHandler(webapp);

        //Register new transaction manager in JNDI
        //At runtime, the webapp accesses this as java:comp/UserTransaction
        org.eclipse.jetty.plus.jndi.Transaction transactionMgr = new
        org.eclipse.jetty.plus.jndi.Transaction(new com.acme.MockUserTransaction());

        //Define an env entry with Server scope.
        //At runtime, the webapp accesses this as java:comp/env/woggle
        //This is equivalent to putting an env-entry in web.xml:
        //<env-entry>
        //  <env-entry-name>woggle</env-entry-name>
        //  <env-entry-type>java.lang.Integer</env-entry-type>
        //  <env-entry-value>4000</env-entry-value>
        //</env-entry>
        org.eclipse.jetty.plus.jndi.EnvEntry woggle = new
        org.eclipse.jetty.plus.jndi.EnvEntry(server, "woggle", new Integer(4000), false);

        //Define an env entry with webapp scope.
        //At runtime, the webapp accesses this as java:comp/env/wiggle
        //This is equivalent to putting a web.xml entry in web.xml:
```

```

//<env-entry>
//  <env-entry-name>wiggle</env-entry-name>
//  <env-entry-value>100</env-entry-value>
//  <env-entry-type>java.lang.Double</env-entry-type>
//</env-entry>
//Note that the last arg of "true" means that this definition for "wiggle" would
override an entry of the
//same name in web.xml
org.eclipse.jetty.plus.jndi.EnvEntry wiggle = new
org.eclipse.jetty.plus.jndi.EnvEntry(webapp, "wiggle", new Double(100), true);

//Register a reference to a mail service scoped to the webapp.
//This must be linked to the webapp by an entry in web.xml:
// <resource-ref>
//   <res-ref-name>mail/Session</res-ref-name>
//   <res-type>javax.mail.Session</res-type>
//   <res-auth>Container</res-auth>
// </resource-ref>
//At runtime the webapp accesses this as java:comp/env/mail/Session
org.eclipse.jetty.jndi.factories.MailSessionReference mailref = new
org.eclipse.jetty.jndi.factories.MailSessionReference();
mailref.setUser("CHANGE-ME");
mailref.setPassword("CHANGE-ME");
Properties props = new Properties();
props.put("mail.smtp.auth", "false");
props.put("mail.smtp.host", "CHANGE-ME");
props.put("mail.from", "CHANGE-ME");
props.put("mail.debug", "false");
mailref.setProperties(props);
org.eclipse.jetty.plus.jndi.Resource xxxmail = new
org.eclipse.jetty.plus.jndi.Resource(webapp, "mail/Session", mailref);

// Register a mock DataSource scoped to the webapp
//This must be linked to the webapp via an entry in web.xml:
//<resource-ref>
//  <res-ref-name>jdbc/mydatasource</res-ref-name>
//  <res-type>javax.sql.DataSource</res-type>
//  <res-auth>Container</res-auth>
//</resource-ref>
//At runtime the webapp accesses this as java:comp/env/jdbc/mydatasource
org.eclipse.jetty.plus.jndi.Resource mydatasource = new
org.eclipse.jetty.plus.jndi.Resource(webapp, "jdbc/mydatasource",
new com.acme.MockDataSource());

server.start();
server.join();
}
}

```

Datasource Examples

Here are examples of configuring a JNDI datasource for various databases.



Note

Read the section called “Configuring DataSources” in the section called “Configuring JNDI” for more information about configuring datasources.

All of these examples correspond to a `resource-ref` in `web.xml`.

```

<resource-ref>
  <description>My DataSource Reference</description>
  <res-ref-name>jdbc/DSTest</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>

```

These examples assume that all of the datasources are declared at the JVM scope, but you can use other scopes if desired. You can configure all JNDI resources in a `jetty.xml` file or in a `WEB-INF/jetty-env.xml` file, or a context XML file. See the section the section called “Deciding Where to Declare Resources” for more information.



Important

You must provide Jetty with the libraries necessary to instantiate the datasource you have configured by putting the corresponding Jar in `JETTY_HOME/lib/ext`.

Pooling DataSources

Pooling datasources enables connection pooling, which lets you reuse an existing connection instead of creating a new connection to the database. This is highly efficient in terms of memory allocation and speed of the request to the database. We highly recommend this option for production environments.

The following is a list of the pooled datasource examples we have worked with in the past:

- the section called “HikariCP”
- the section called “BoneCP”
- the section called “c3p0”
- the section called “DBCP”
- the section called “Atomikos 3.3.2+”
- the section called “MySQL”
- the section called “PostgreSQL”
- the section called “DB2”

HikariCP

Connection pooling, available at [HikariCP Download](#). All configuration options for HikariCP are described here: [HikariCP documentation](#).

```
<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
    <Arg></Arg>
    <Arg>jdbc/DSTest</Arg>
    <Arg>
        <New class="com.zaxxer.hikari.HikariDataSource">
            <Arg>
                <New class="com.zaxxer.hikari.HikariConfig">
                    <Set name="minimumPoolSize">5</Set>
                    <Set name="maximumPoolSize">50</Set>

                    <Set name="dataSourceClassName">com.mysql.jdbc.jdbc2.optional.MysqlDataSource</Set>
                        <Call name="addDataSourceProperty">
                            <Arg>url</Arg>
                            <Arg>jdbc.url</Arg>
                        </Call>
                        <Call name="addDataSourceProperty">
                            <Arg>user</Arg>
                            <Arg>jdbc.user</Arg>
                        </Call>
                        <Call name="addDataSourceProperty">
                            <Arg>password</Arg>
                            <Arg>jdbc.pass</Arg>
                        </Call>
                </New>
            </Arg>
        </New>
    </Arg>
</New>
```

```

        </Call>
    </New>
</Arg>
</New>
</Arg>
</New>

```

BoneCP

Connection pooling, available at [BoneCP Download](#). All configuration options for BoneCP are described here: [BoneCP API](#).

```

<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
<Arg></Arg>
<Arg>jdbc/DSTest</Arg>
<Arg>
<New class="com.jolbox.bonecp.BoneCPDataSource">
<Set name="driverClass">com.mysql.jdbc.Driver</Set>
<Set name="jdbcUrl">jdbc.url</Set>
<Set name="username">jdbc.user</Set>
<Set name="password">jdbc.pass</Set>
<Set name="minConnectionsPerPartition">5</Set>
<Set name="maxConnectionsPerPartition">50</Set>
<Set name="acquireIncrement">5</Set>
<Set name="idleConnectionTestPeriod">30</Set>
</New>
</Arg>
</New>

```

c3p0

Connection pooling, available at [c3p0 Jar](#).

```

<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
<Arg></Arg>
<Arg>jdbc/DSTest</Arg>
<Arg>
<New class="com.mchange.v2.c3p0.ComboPooledDataSource">
<Set name="driverClass">org.some.Driver</Set>
<Set name="jdbcUrl">jdbc.url</Set>
<Set name="user">jdbc.user</Set>
<Set name="password">jdbc.pass</Set>
</New>
</Arg>
</New>

```

DBCP

Connection pooling, available at [dbcp Jar](#).

```

<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
<Arg></Arg>
<Arg>jdbc/DSTest</Arg>
<Arg>
<New class="org.apache.commons.dbcp.BasicDataSource">
<Set name="driverClassName">org.some.Driver</Set>
<Set name="url">jdbc.url</Set>
<Set name="username">jdbc.user</Set>

```

```

<Set name="password">jdbc.pass</Set>
</New>
</Arg>
</New>

```

Atomikos 3.3.2+

Connection pooling + XA transactions.

```

<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
    <Arg></Arg>
    <Arg>jdbc/DSTest</Arg>
    <Arg>
        <New class="com.atomikos.jdbc.AtomikosDataSourceBean">
            <Set name="minPoolSize">2</Set>
            <Set name="maxPoolSize">50</Set>

            <Set name="xaDataSourceClassName">com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</Set>
            <Set name="UniqueResourceName">DSTest</Set>
            <Get name="xaProperties">
                <Call name="setProperty">
                    <Arg>url</Arg>
                    <Arg>jdbc:mysql://localhost:3306/databasename</Arg>
                </Call>
                <Call name="setProperty">
                    <Arg>user</Arg>
                    <Arg>some_username</Arg>
                </Call>
                <Call name="setProperty">
                    <Arg>password</Arg>
                    <Arg>some_password</Arg>
                </Call>
            </Get>
        </New>
    </Arg>
</New>

```

MySQL

Implements javax.sql.DataSource, javax.sql.ConnectionPoolDataSource.

```

<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
    <Arg></Arg>
    <Arg>jdbc/DSTest</Arg>
    <Arg>
        <New class="com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource">
            <Set name="Url">jdbc:mysql://localhost:3306/databasename</Set>
            <Set name="User">user</Set>
            <Set name="Password">pass</Set>
        </New>
    </Arg>
</New>

```

PostgreSQL

Implements javax.sql.ConnectionPoolDataSource

```

<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
```

```

<Arg></Arg>
<Arg>jdbc/DSTest</Arg>
<Arg>
  <New class="org.postgresql.ds.PGConnectionPoolDataSource">
    <Set name="User">user</Set>
    <Set name="Password">pass</Set>
    <Set name="DatabaseName">dbname</Set>
    <Set name="ServerName">localhost</Set>
    <Set name="PortNumber">5432</Set>

  </New>
</Arg>
</New>

```

DB2

Implements javax.sql.ConnectionPoolDataSource

```

<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
<Arg></Arg>
<Arg>jdbc/DSTest</Arg>
<Arg>
  <New class="com.ibm.db2.jcc.DB2ConnectionPoolDataSource">
    <Set name="DatabaseName">dbname</Set>
    <Set name="User">user</Set>
    <Set name="Password">pass</Set>
    <Set name="ServerName">servername</Set>
    <Set name="PortNumber">50000</Set>
  </New>
</Arg>
</New>

```

Non-pooling DataSources

If you are deploying in a production environment, we highly recommend using a Pooling DataSource. Since that is not always an option we have a handful of examples for non-pooling datasources listed here as well.

The following is a list of the non-pooled datasource examples:

- the section called “SQL Server 2000”
- the section called “Oracle 9i/10g”
- the section called “PostgreSQL”
- the section called “Sybase”
- the section called “DB2”

SQL Server 2000

Implements javax.sql.DataSource, javax.sql.ConnectionPoolDataSource.

```

<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
<Arg></Arg>
<Arg>jdbc/DSTest</Arg>
<Arg>
  <New class="net.sourceforge.jtds.jdbcx.JtdsDataSource">
    <Set name="User">user</Set>
    <Set name="Password">pass</Set>

```

```

<Set name="DatabaseName">dbname</Set>
<Set name="ServerName">localhost</Set>
<Set name="PortNumber">1433</Set>
</New>
</Arg>
</New>

```

Oracle 9i/10g

Implements javax.sql.DataSource, javax.sql.ConnectionPoolDataSource.

```

<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
<Arg></Arg>
<Arg>jdbc/DSTest</Arg>
<Arg>
<New class="oracle.jdbc.pool.OracleDataSource">
<Set name="DriverType">thin</Set>
<Set name="URL">jdbc:oracle:thin:@fmsswdb1:10017:otcd</Set>
<Set name="User">xxxxx</Set>
<Set name="Password">xxxxx</Set>
<Set name="connectionCachingEnabled">true</Set>
<Set name="connectionCacheProperties">
<New class="java.util.Properties">
<Call name="setProperty">
<Arg>MinLimit</Arg>
<Arg>5</Arg>
</Call>
<!-- put the other properties in here too -->
</New>
</Set>
</New>
</Arg>
</New>

```

For more information, refer to: [Oracle Database JDBC documentation](#).

PostgreSQL

Implements javax.sql.DataSource.

```

<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
<Arg></Arg>
<Arg>jdbc/DSTest</Arg>
<Arg>
<New class="org.postgresql.ds.PGSimpleDataSource">
<Set name="User">user</Set>
<Set name="Password">pass</Set>
<Set name="DatabaseName">dbname</Set>
<Set name="ServerName">localhost</Set>
<Set name="PortNumber">5432</Set>
</New>
</Arg>
</New>

```

Sybase

Implements javax.sql.DataSource.

```

<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
<Arg></Arg>
<Arg>jdbc/DSTest</Arg>
<Arg>
<New class="com.sybase.jdbc2.jdbc.SybDataSource">
<Set name="DatabaseName">dbname</Set>
<Set name="User">user</Set>
<Set name="Password">pass</Set>
<Set name="ServerName">servername</Set>
<Set name="PortNumber">5000</Set>
</New>
</Arg>

```

</New>

DB2

Implements javax.sql.DataSource.

```
<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
  <Arg></Arg>
  <Arg>jdbc/DSTest</Arg>
  <Arg>
    <New class="com.ibm.db2.jcc.DB2SimpleDataSource">
      <Set name="DatabaseName">dbname</Set>
      <Set name="User">user</Set>
      <Set name="Password">pass</Set>
      <Set name="ServerName">servername</Set>
      <Set name="PortNumber">50000</Set>
    </New>
  </Arg>
</New>
```

Chapter 12. Annotations

Table of Contents

Quick Setup	172
Working with Annotations	173
Using Annotations with Jetty Embedded	175

Jetty supports the servlet specification annotations. It is not enabled by default, so the following sections show you how to enable it, and how to use them.

Quick Setup

If you are using the standard distribution of Jetty, and want to enable processing of annotation for *all* your webapps, edit the `$JETTY_HOME/start.ini` file and uncomment the following lines:

```
=====
# Enable servlet 3.1 annotations
# -----
OPTIONS=annotations
etc/jetty-annotations.xml

=====
# Enable additional webapp environment configurators
# -----
OPTIONS=plus
etc/jetty-plus.xml
```

If you want to use annotations that relate to [JNDI](#), such as `@Resource` and `@Resources` then you should also uncomment these other lines also:

```
=====
# JNDI
# -----
OPTIONS=jndi
```

You can now start Jetty and use annotations within your webapps. See [Using Annotations](#) for more information.

Since **jetty-9.1.0** annotation scanning will be conducted by default in a multi-threaded manner. This speeds up the process of opening jar files and looking for [discoverable annotations](#). Jetty will use threads from the [configured thread pool](#) to perform the scanning. You can disable threaded scanning and use just the single main execution thread if for some reason you need to. You have several options for doing this:

1. Set `org.eclipse.jetty.annotations.multiThreaded` to `false` on a [context](#) attribute for the specific webapp you want to affect
2. Set `org.eclipse.jetty.annotations.multiThreaded` to `false` on a [Server](#) attribute to affect all webapps deployed into that Server[server_attributes](#)
3. Set `-Dorg.eclipse.jetty.annotations.multiThreaded=false` as a System property in a `start.ini` file or command line to affect all webapps in the jvm

By default, Jetty will wait up to 60 seconds for the scanning to complete when using the multi-threaded mode. If you would like to make it longer or shorter, you can use any of the following:

1. Set **org.eclipse.jetty.annotations.maxWait** as a [context attribute](#) to the time in seconds you want to maximally wait for scanning to complete. This affects the specific webapp only.
2. Set **org.eclipse.jetty.annotations.maxWait** as a [Server attribute](#) to the time in seconds you want to maximally wait for scanning to complete. This affects all webapps deployed to that Server.
3. Set **-Dorg.eclipse.jetty.annotations.maxWait** as a System property to the time in seconds you want to maximally wait for scanning to complete. This affects all webapps deployed in the jvm.

Working with Annotations

Which Annotations Are Supported

Jetty supports interpretation and application of the following annotations:

- **@Resource**
- **@Resources**
- **@PostConstruct**
- **@PreDestroy**
- **@DeclaredRoles**
- **@RunAs**
- **@MultipartConfig**
- **@WebServlet**
- **@WebFilter**
- **@WebListener**
- **@WebInitParam**
- **@ServletSecurity**, **@HttpConstraint**, **@HttpMethodConstraint**
- **@HandlesTypes** (on **ServletContainerInitializers**)

Discovered vs Introspected Annotations

Some types of annotation can be placed on any classes, not necessarily just those with which the container interacts directly. We call these type of annotations "discovered" to indicate that the container must take proactive action to go out and find them. The other type of annotation we call "introspected", meaning that they occur on classes with which the container interacts during their lifecycle (eg `javax.servlet.Servlet`, `javax.servlet.Filter` etc), and hence can be found by simple inspection of the class at that point.

Some examples of discovered annotations are:

- **@WebServlet**
- **@WebFilter**

- `@WebListener`

Some examples of introspected annotations are:

- `@PostConstruct`
- `@PreDestroy`
- `@Resource`

Which Jars Are Scanned For Discovered Annotations

The web.xml file can contain the attribute `metadata-complete`. If this is set to `true`, then *no* scanning of discoverable annotations takes place. However, scanning of classes may *still* occur because of [javax.servlet.ServletContainerInitializers](#). Classes implementing this interface are found by Jetty using the [javax.util.ServiceLoader](#) mechanism, and if one is present *and* it includes the `@HandlesTypes` annotation, then Jetty must scan the class hierarchy of the web application. This may be very time-consuming if you have many jars in the container's path or in the webapp's WEB-INF/lib.

If scanning is to take place - because either `metadata-complete` is `false` or missing, or because there are one or more [javax.servlet.ServletContainerInitializers](#) with `@HandlesTypes` - then Jetty must consider both the container's classpath and the webapp's classpath.

By default, Jetty will *not* scan any classes that are on the container's classpath. If you need to cause jars and classes that are on the container's classpath to be scanned, then you can use the [org.eclipse.jetty.server.webapp.ContainerIncludeJarPattern context attribute](#) to specify a pattern for jars and directories from the container's classpath to scan.

By default, Jetty will scan *all* classes from WEB-INF/classes, and all jars from WEB-INF/lib according to the order, if any, established by absolute or relative ordering clauses in web.xml. If your webapp contains many jars, you can significantly speed up deployment by omitting them from scanning. To do this, use the [org.eclipse.jetty.server.webapp.WebInfIncludeJarPattern context attribute](#) to define the patterns of jars that you specifically want to be scanned.

Note that if you have configured an [extraClasspath](#) for the webapp, then it participates in the scanning process too. Any classes dirs are treated the same for scanning purposes as if they were in WEB-INF/classes and jars are treated as if they were in WEB-INF/lib.

See also the next section on [ServletContainerInitializers](#) if you need to [control the order in which they are applied](#).

Multi-threaded Annotation Scanning

Since jetty-9.1.0.RC1, [if annotation scanning is to be performed](#), by default Jetty will do it in a multi-threaded manner in order to complete it in the minimum amount of time.

If for some reason you don't want to do it multi-threaded, you can configure Jetty to revert to single-threaded scanning. You have several ways to configure this:

1. set the `context attribute` `org.eclipse.jetty.annotations.multiThreaded` to `false`
2. set the `Server attribute` `org.eclipse.jetty.annotations.multiThreaded` to `false`
3. set the `System property` `org.eclipse.jetty.annotations.multiThreaded` to `false`

Method 1 will only affect the current webapp. Method 2 will affect all webapps deployed to the same Server instance. Method 3 will affect all webapps deployed in the same jvm.

By default, Jetty will wait a maximum of 60 seconds for all of the scanning threads to complete. You can set this to a higher or lower number of seconds by doing one of the following:

1. set the [context attribute](#) `org.eclipse.jetty.annotations.maxWait`
2. set the [Server attribute](#) `org.eclipse.jetty.annotations.maxWait`
3. set the System property `org.eclipse.jetty.annotations.maxWait`

Method 1 will only affect the current webapp. Method 2 will affect all webapps deployed to the same Server instance. Method 3 will affect all webapps deployed in the same jvm.

ServletContainerInitializers

[javax.servlet.ServletContainerInitializers](#) can exist in: the container's classpath, the webapp's WEB-INF/classes directory, the webapp's WEB-INF/lib jars, or any external [extraClasspath](#) that you have configured on the webapp.

The [Servlet Specification](#) does not define any order in which these ServletContainerInitializers must be called when the webapp starts. Since jetty-9.1.0.RC1, by default Jetty will call them in the following order:

1. ServletContainerInitializers from the container's classpath
2. ServletContainerInitializers from WEB-INF/classes
3. ServletContainerInitializers from WEB-INF/lib jars in *the order established in web.xml*, or in the order that the SCI is returned by the [javax.util.ServiceLoader](#) if there is *no ordering*

As is the case with annotation scanning, the [extraClasspath](#) is fully considered for ServletContainerInitializer callbacks. ServletContainerInitializers derived from a classes dir on the extraClasspath and jars from an extraClasspath for the webapp are called in step 2 and 3 respectively.

If you need ServletContainerInitializers called in a specific order that is different from that outlined above, then you can use the [context attribute](#) `org.eclipse.jetty.containerInitializerOrder`. Set it to a list of comma separated class names of ServletContainerInitializers in the order that you want them applied. You may optionally use the wildcard character "*" once in the list. It will match all ServletContainerInitializers not explicitly named in the list. Here's an example, setting the context attribute in code (although you can also do the [same in xml](#)):

```
WebAppContext context = new WebAppContext();
context.setAttribute("org.eclipse.jetty.containerInitializerOrder",
                     "org.eclipse.jetty.websocket.jsr356.server.deploy.WebSocketServerContainerInitializer,
                     com.acme.Foo.MySCI, *");
```

In this example, we ensure that the WebSocketServerContainerInitializer is the very first ServletContainerInitializer that is called, followed by MySCI and then any other ServletContainerInitializers that were discovered but not yet called.

Using Annotations with Jetty Embedded

Setting up the Classpath

You will need to place the following jetty jars onto the classpath of your application. You can obtain them from the [jetty distribution](#), or the [maven repository](#):

```
jetty-plus.jar
jetty-annotations.jar
```

You will also need the [asm](#) jar, which you can obtain from the [Jetty dependencies site](#).

Example

Here's an example application that sets up a Jetty server, does some setup to ensure that annotations are scanned and deploys a webapp that uses annotations. This example also uses the @Resource annotation which involves JNDI, so we would also [add the necessary jndi jars to the classpath](#), and we also add in the configuration classes that are responsible for JNDI (see line 19).

Here is the embedding code:

```
import org.eclipse.jetty.security.HashLoginService;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.webapp.WebAppContext;

/**
 * ServerWithAnnotations
 *
 */
public class ServerWithAnnotations
{
    public static final void main(String args[]) throws Exception
    {
        //Create the server
        Server server = new Server(8080);

        //Enable parsing of jndi-related parts of web.xml and jetty-env.xml
        org.eclipse.jetty.webapp.Configuration.ClassList classlist =
        org.eclipse.jetty.webapp.Configuration.ClassList.setServerDefault(server);

        classlist.addAfter("org.eclipse.jetty.webapp.FragmentConfiguration", "org.eclipse.jetty.plus.webapp.EnvConfig");
        classlist.addBefore("org.eclipse.jetty.webapp.JettyWebXmlConfiguration", "org.eclipse.jetty.annotations.AnnotationConfiguration");

        //Create a WebApp
        WebAppContext webapp = new WebAppContext();
        webapp.setContextPath("/");
        webapp.setWar("../tests/test-webapps/test-servlet-spec/test-spec-webapp/target/test-spec-webapp-9.0.4-SNAPSHOT.war");
        server.setHandler(webapp);

        //Register new transaction manager in JNDI
        //At runtime, the webapp accesses this as java:comp/UserTransaction
        org.eclipse.jetty.plus.jndi.Transaction transactionMgr = new
        org.eclipse.jetty.plus.jndi.Transaction(new com.acme.MockUserTransaction());

        //Define an env entry with webapp scope.
        org.eclipse.jetty.plus.jndi.EnvEntry maxAmount = new
        org.eclipse.jetty.plus.jndi.EnvEntry(webapp, "maxAmount", new Double(100), true);

        // Register a mock DataSource scoped to the webapp
        org.eclipse.jetty.plus.jndi.Resource mydatasource = new
        org.eclipse.jetty.plus.jndi.Resource(webapp, "jdbc/mydatasource", new
        com.acme.MockDataSource());

        // Configure a LoginService
        HashLoginService loginService = new HashLoginService();
        loginService.setName("Test Realm");
        loginService.setConfig("src/test/resources/realm.properties");
        server.addBean(loginService);

        server.start();
        server.join();
    }
}
```

On line 19 we add in the configuration classes responsible for setting up JNDI and java:comp/env.

On line 20 we add in the configuration class that ensures annotations are inspected.

On lines 30, 33 and 37 we set up some JNDI resources that we will be able to reference with @Resource annotations.

With the setup above, we can create a servlet that uses annotations and Jetty will honour the annotations when the webapp is deployed:

```

import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RunAs;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
import javax.transaction.UserTransaction;

/**
 * AnnotationTest
 *
 * Use servlet 3.0 annotations from within Jetty.
 *
 * Also uses servlet 2.5 resource injection and lifecycle callbacks
 */

@RunAs("special")
@WebServlet(urlPatterns = {"/", "/test/*"}, name="AnnotationTest",
    initParams={@WebInitParam(name="fromAnnotation", value="xyz")})
@DeclareRoles({"user", "client"})
public class AnnotationTest extends HttpServlet
{
    private DataSource myDS;

    @Resource(mappedName="UserTransaction")
    private UserTransaction myUserTransaction;

    @Resource(mappedName="maxAmount")
    private Double maxAmount;

    @Resource(mappedName="jdbc/mydatasource")
    public void setMyDatasource(DataSource ds)
    {
        myDS=ds;
    }

    @PostConstruct
    private void myPostConstructMethod()
    {
        System.err.println("PostConstruct called");
    }

    @PreDestroy
    private void myPreDestroyMethod()
    {
        System.err.println("PreDestroy called");
    }

    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException
    {

```

```
        doGet(request, response);
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException
    {
        try
        {
            response.setContentType("text/html");
            ServletOutputStream out = response.getOutputStream();
            out.println("<html>");
            out.println("<body>");
            out.println("<h1>Results</h1>");
            out.println(myDS.toString());
            out.println("<br/>");
            out.println(maxAmount.toString());
            out.println("</body>");
            out.println("</html>");
            out.flush();
        }
        catch (Exception e)
        {
            throw new ServletException(e);
        }
    }
}
```

Chapter 13. JMX

Table of Contents

Using Java Management Extensions (JMX)	179
Jetty JConsole	182
Jetty JMX Annotations	184

Using Java Management Extensions (JMX)

The [Java Management Extensions \(JMX\) API](#) is a standard API for managing and monitoring resources such as applications, devices, services, and the Java virtual machine.

Typical uses of the JMX technology include:

- Consulting and changing application configuration.
- Accumulating and making available statistics about application behavior.
- Notifying of state changes and erroneous conditions.

The JMX API includes remote access, so a remote management program can interact with a running application for these purposes.

Jetty JMX integration uses the platform MBean server implementation that Java VM provides. The integration is based on the ObjectMBean implementation of DynamicMBean. This implementation allows you to wrap an arbitrary POJO in an MBean and annotate it appropriately to expose it via JMX. See the section called “Jetty JMX Annotations”.

The MBeanContainer implementation of the ContainerListener interface coordinates creation of MBeans. The Jetty Server and its components use a [Container](#) to maintain a containment tree of components and to support notification of changes to that tree. The MBeanContainer class listens for Container events and creates and destroys MBeans as required to wrap all Jetty components.

You can access the MBeans that Jetty publishes both through built-in Java VM connector via JConsole, or by registering a remote JMX connector and using a remote JMX agent to monitor Jetty.

Configuring JMX

This guide describes how to initialize and configure the Jetty JMX integration.

To monitor an application using JMX, perform the following steps:

- Configure the application to instantiate an MBean container.
- Instrument objects to be MBeans.
- Provide access for JMX agents to MBeans.

Using JConsole to Access Jetty MBeans

The simplest way to access the MBeans that Jetty publishes is to use the [JConsole utility](#) the Java Virtual Machine supplies. See the section called “Jetty JConsole” for instructions on how to configure JVM for use with JConsole.

To access Jetty MBeans via JConsole, you must:

- Enable the registration of Jetty MBeans into the platform MBeanServer.
- Enable a JMXConnectorServer so that JConsole can connect and visualize the MBeans.

Registering Jetty MBeans

Configuring Jetty JMX integration differs for standalone and embedded Jetty.

Standalone Jetty

JMX is enabled by default in the jetty-9 distribution. If you are having difficulties validate that the section in the jetty.home/start.ini file is uncommented.

Embedded Jetty

When running Jetty embedded into an application, create and configure an MBeanContainer instance as follows:

```
Server server = new Server();

// Setup JMX
MBeanContainer mbContainer=new
MBeanContainer(ManagementFactory.getPlatformMBeanServer());
server.getContainer().addEventListener(mbContainer);
server.addBean(mbContainer);

// Add loggers MBean to server (will be picked up by MBeanContainer above)
server.addBean(Log.getLog());
```

Notice that Jetty creates the MBeanContainer immediately after creating the Server, and immediately after registering it as an EventListener of the Server's Container object.

Because logging is initialized prior to the MBeanContainer (even before the Server itself), it is necessary to register the logger manually via `server.addBean()` so that the loggers may show up in the JMX tree

Using the Jetty Maven Plugin with JMX

If you are using the [jetty maven plugin](#) you should copy the `etc/jetty-jmx.xml` file into your webapp project somewhere, such as `src/etc`, then add a `<jettyconfig>` element to the plugin `<configuration>`:

```
<plugin>
<groupId>org.eclipse.jetty</groupId>
<artifactId>jetty-maven-plugin</artifactId>
<version>9.2.1.v20140609</version>
<configuration>
  <scanIntervalSeconds>10</scanIntervalSeconds>
    <jettyXml>src/etc/jetty-jmx.xml</jettyXml>
  </configuration>
</plugin>
```

Enabling JMXConnectorServer for Remote Access

There are two ways of enabling remote connectivity so that JConsole can connect to visualize MBeans.

- Use the `com.sun.management.jmxremote` system property on the command line. Unfortunately, this solution does not play well with firewalls and it is not flexible.
- Use Jetty's `ConnectorServer` class. To enable use of this class, uncomment the correspondent portion in `etc/jetty-jmx.xml`, like this:

```
<New id="ConnectorServer" class="org.eclipse.jetty.jmx.ConnectorServer">
<Arg>
<New class="javax.management.remote.JMXServiceURL">
<Arg type="java.lang.String">rmi</Arg>
<Arg type="java.lang.String" />

<Arg type="java.lang.Integer"><SystemProperty name="jetty.jmxrmiport" default="1099"/>
</Arg>
<Arg type="java.lang.String">/jndi/
rmi://<SystemProperty name="jetty.jmxrmihost" default="localhost"/><SystemProperty name="jetty.jmxrmiport" default="1099"/>/jmxrmi</Arg>
</New>
</Arg>
<Arg>org.eclipse.jetty.jmx:name=rmiconNECTORserver</Arg>
<Call name="start" />
</New>
```

This configuration snippet starts an RMIServer and a JMXConnectorServer both on port 1099 (by default), so that firewalls should open just that one port to allow connections from JConsole.

Securing Remote Access

JMXConnectorServer several options to restrict access. For a complete guide to controlling authentication and authorization in JMX, see [Authentication and Authorization in JMX RMI connectors](#) in Luis-Miguel Alventosa's blog.

To restrict access to the JMXConnectorServer, you can use this configuration, where the `jmx.password` and `jmx.access` files have the format specified in the blog entry above:

```
<New id="ConnectorServer" class="org.eclipse.jetty.jmx.ConnectorServer">
<Arg>
<New class="javax.management.remote.JMXServiceURL">
<Arg type="java.lang.String">rmi</Arg>
<Arg type="java.lang.String" />

<Arg type="java.lang.Integer"><SystemProperty name="jetty.jmxrmiport" default="1099"/>
</Arg>
<Arg type="java.lang.String">/jndi/
rmi://<SystemProperty name="jetty.jmxrmihost" default="localhost"/><SystemProperty name="jetty.jmxrmiport" default="1099"/>/jmxrmi</Arg>
</New>
</Arg>
<Arg>
<Map>
<Entry>
<Item>jmx.remote.x.password.file</Item>
<Item>
<New class="java.lang.String"><Arg><Property name="jetty.home" default="." />/resources/jmx.password</Arg></New>
</Item>
</Entry>
<Entry>
<Item>jmx.remote.x.access.file</Item>
<Item>
<New class="java.lang.String"><Arg><Property name="jetty.home" default="." />/resources/jmx.access</Arg></New>
</Item>
</Entry>
</Map>
```

```
</Arg>
<Arg>org.eclipse.jetty.jmx:name=rmiconnectorserver</Arg>
<Call name="start" />
</New>
```

Custom Monitor Application

Using the JMX API, you can also write a custom application to monitor your Jetty server. To allow this application to connect to your Jetty server, you need to uncomment the last section of your `etc/jetty-jmx.xml` configuration file and optionally modify the endpoint name. Doing so creates a JMX HTTP connector and registers a JMX URL that outputs to the `Stderr` log.

You should provide the URL that appears in the log to your monitor application in order to create an `MBeanServerConnection`. You can use the same URL to connect to your Jetty instance from a remote machine using JConsole. See the [configuration file](#) for more details.

Jetty JConsole

JConsole is a graphical tool; it allows you to remotely manage and monitor your server and web application status using JMX. When following the instructions given below, please also ensure that you make any necessary changes to any anti-virus software you may be using which may prevent jconsole from running.

Monitoring Jetty with JConsole

To monitor Jetty's server status with JConsole, make sure JConsole is running, and start Jetty with a special system property.

Starting Jetty Standalone

The simplest way to enable support is to add the jmx support module to your `${jetty.base}`.

```
[mybase]$ java /opt/jetty-dist/start.jar --add-to-start=jmx
INFO: jmx-remote      initialised in ${jetty.base}/start.ini (appended)
INFO: jmx            initialised transitively
```

Then open the `${jetty.base}/start.ini` file and edit the properties to suit your needs:

```
# 
# Initialize module jmx-remote
#
--module=jmx-remote
## JMX Configuration
## Enable for an open port accessible by remote machines
jetty.jmxrmihost=localhost
jetty.jmxrmiport=1099
## Strictly speaking you shouldn't need --exec to use this in most environments.
## If this isn't working, make sure you enable --exec as well
-Dcom.sun.management.jmxremote
```

Starting the Jetty Maven Plugin

If you are running the Jetty Maven Plugin, you must set the system property `com.sun.management.jmxremote` on Maven before running the plugin. The way to do this is to set

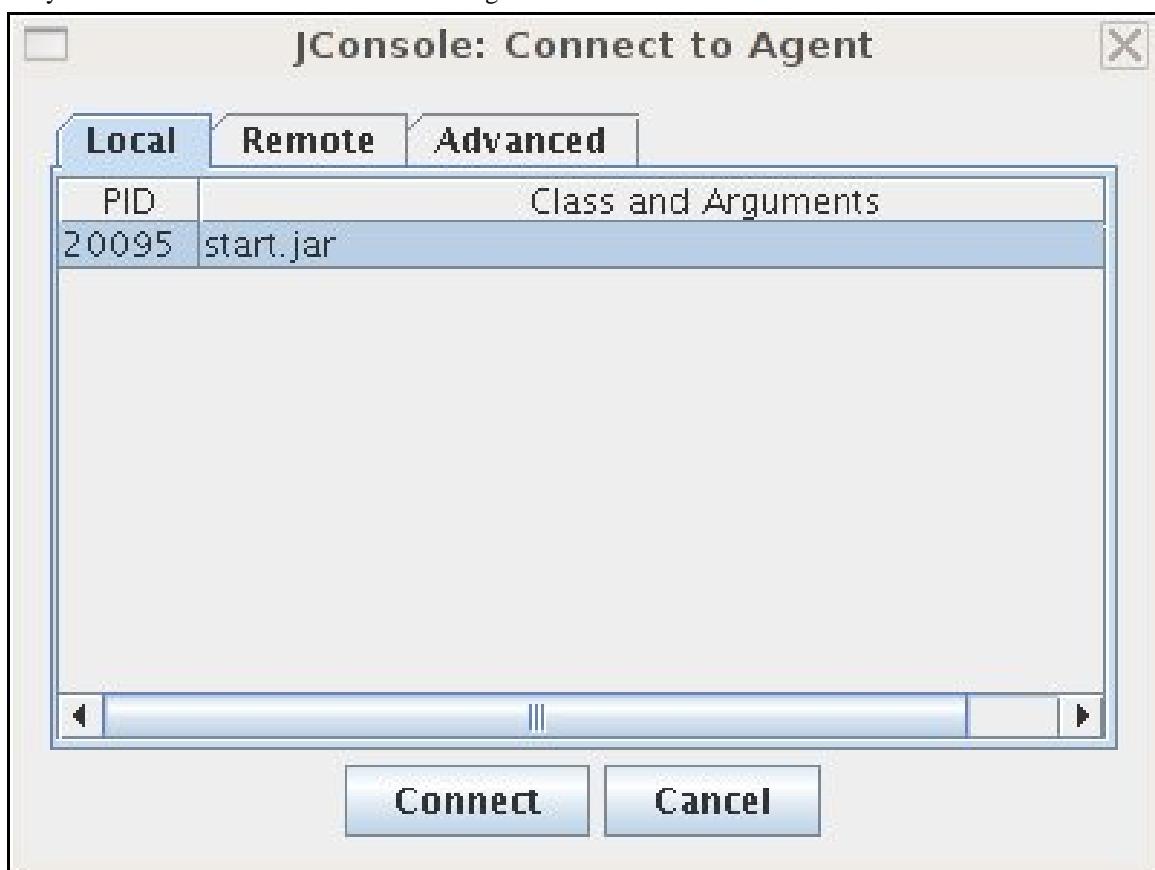
your MAVEN_OPTS environment variable (if you're not sure how to do this, consult the Maven documentation).

Here is an example that sets the system property on the fly in a BASH shell, before starting Jetty via the plugin:

```
$ export MAVEN_OPTS=-Dcom.sun.management.jmxremote
$ mvn jetty:run
$ jconsole &           # runs jconsole in the background
```

Connecting to your server process

When you start Jetty, you see a dialog box from JConsole with a list of running processes to which you can connect. It should look something like so:

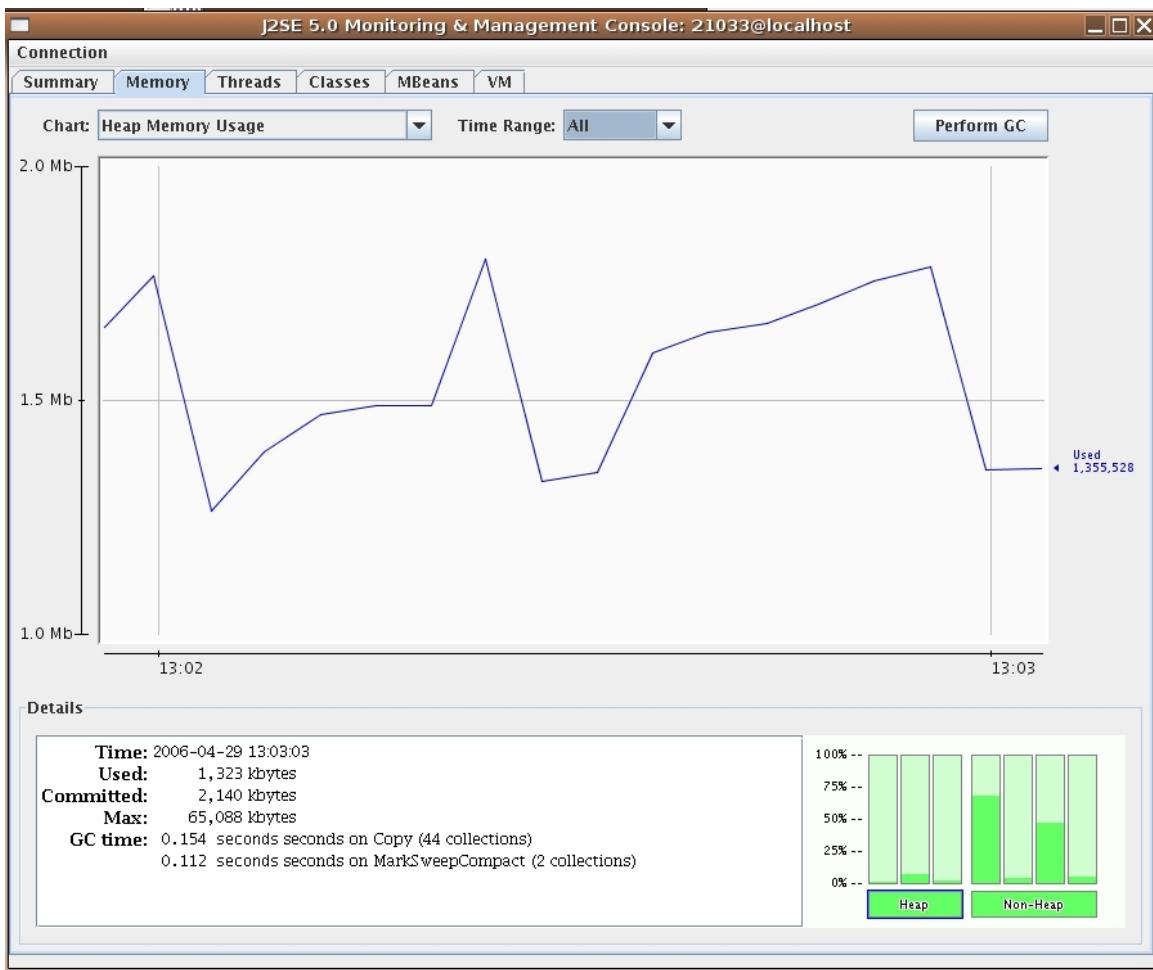


Finding your process



If you don't see your Jetty process in the list of processes you can connect to, quickly switch tabs, or close and reopen a new "New Connection" dialog window. This forces JConsole to refresh the list, and recognize your newly-started Jetty process.

Select the start.jar entry and click the "Connect" button. A new JConsole window opens:



From this window you can monitor memory usage, thread usage, classloading and VM statistics. You can also perform operations such as a manual garbage collect. JConsole is an extremely powerful and useful tool.

Managing Jetty Objects with JConsole

The MBean tab of JConsole allows access to managed objects within the Java application, including MBeans the JVM provides. If you also want to interact with the Jetty JMX implementation via JConsole, you need to start Jetty JMX in a form that JConsole can access. See the section called “Using Java Management Extensions (JMX)” for more information.

Jetty JMX Annotations

When the jetty-jmx libraries are present on startup and the wiring is enabled for exposing jetty mbeans to jmx, there are three annotations that govern when and how mbeans are created and exposed.

Annotation Introspection

When jmx is configured and enabled in jetty any time an object is registered with the Server it is introspected as a potential mbean to be exposed. This introspection proceeds as follows assuming the class is named `com.acme.Foo`:

1. All influences for `com.acme.Foo` determined. These include each class in the chain of super classes, and by convention each of these classes following a form of

com.acme.jmx.FooMBean. All super classes and their corresponding MBean representations are then used in the next step.

2. Each potential influencing class is checked for the @ManagedObject annotation, should this annotation exist at any point in the chain of influencers then a mbean is created with the description of the version @ManageObject discovered.
3. Once an mbean has been created for an object then each potential influencing object is introspected for @ManagedAttribute and @ManagedOperation annotations and the corresponding type is exposed to the mbean.

The convention of looking for @ManageObject annotations on .jmx.ClassMBean allows for a normal POJO to be wrapped in an mbean without itself without requiring it being marked up with annotations. Since the pojo is passed to these wrapped derived Mbean instances and is an internal variable then the MBean can be used to better expose a set of attributes and operations that may not have been anticipated when the original object was created.

@ManagedObject

The @ManagedObject annotation is used on a class at the top level to indicate that it should be exposed as an mbean. It has only one attribute to it which is used as the description of the MBean. Should multiple @ManagedObject annotations be found in the chain of influence then the first description is used.

The list of attributes available are:

value

The description of the Managed Object.

@ManagedAttribute

The @ManagedAttribute annotation is used to indicate that a given method exposes a JMX attribute. This annotation is placed always on the reader method of a given attribute. Unless it is marked as read-only in the configuration of the annotation a corresponding setter is looked for following normal naming conventions. For example if this annotation is on a method called getFoo() then a method called setFoo() would be looked for and if found wired automatically into the jmx attribute.

The list of attributes available are:

value

The description of the Managed Attribute.

name

The name of the Managed Attribute.

proxied

Value is true if the corresponding mbean for this object contains the method of this JMX attribute in question.

readonly

By default this value is false which means that a corresponding setter will be looked for and wired into the attribute should one be found. Setting this to true make the jmx attribute read only.

setter

This attribute can be used when the corresponding setter for a JMX attribute follows a non-standard naming convention and it should still be exposed as the setter for the attribute.

@ManagedOperation

The @ManagedOperation annotation is used to indicate that a given method should be considered a JMX operation.

The list of attributes available are:

value

The description of the Managed Operation.

impact

The impact of an operation. By default this value is "UNKNOWN" and acceptable values are "ACTION", "INFO", "ACTION_INFO" and should be used according to their definitions with JMX.

proxied

Value is true if the corresponding mbean for this object contains the method of this JMX operation in question.

@Name

A fourth annotation is often used in conjunction with the JMX annotations mentioned above. This annotation is used to describe variables in method signatures so that when rendered into tools like JConsole it is clear what the parameters are. For example:

The list of attributes available are:

value

The name of the parameter.

description

The description of the parameter.

Example

The following is an example of each of the annotations mentioned above in practice.

```
package com.acme;

import org.eclipse.jetty.util.annotation.ManagedAttribute;
import org.eclipse.jetty.util.annotation.ManagedObject;
import org.eclipse.jetty.util.annotation.ManagedOperation;
import org.eclipse.jetty.util.annotation.Name;

@ManagedObject( "Test MBean Annotations" )
public class Derived extends Base implements Signature
{
    String fname="Full Name";

    @ManagedAttribute(value="The full name of something", name="fname")
    public String getFullName()
    {
        return fname;
    }

    public void setFullName(String name)
    {
        fname=name;
    }

    @ManagedOperation("Doodle something")
    public void doodle(@Name(value="doodle", description="A description of the argument")
String doodle)
    {
        System.out.println("doodle "+doodle);
    }
}
```

Chapter 14. SPDY

Table of Contents

Introducing SPDY	187
Configuring SPDY	187
Configuring SPDY Proxy	188
Configuring SPDY push	194
Implement a custom SPDY PushStrategy	197

Introducing SPDY

Jetty supports both a client and a server implementation for the [SPDY](#) protocol, beginning with versions 7.6.2 and 8.1.2. As Jetty 7 and 8 are now in maintenance-only mode, we recommend using Jetty 9 if you intend to use SPDY. Not all features described here exist in 7/8. To provide the best support possible for SPDY, the Jetty project also provides an implementation for [NPN](#). Both the SPDY and the NPN implementations require OpenJDK 1.7 or greater.

A server deployed over TLS normally advertises the SPDY protocol via the TLS Extension Next Protocol Negotiation ([NPN](#)).



Important

To use SPDY in Jetty you need to add the [NPN boot Jar in the boot classpath](#).

See a [SPDY Push Demonstration video from JavaOne 2012](#).

Jetty's SPDY modules

Jetty's SPDY implementation consists of four modules:

1. `spdy-core`—contains the SPDY API and a partial implementation. This module is independent of Jetty (the servlet container), and you can reuse it with other Java SPDY implementations. One of the goals of this module is to standardize the SPDY Java API.
2. `spdy-jetty` module—binds the `spdy-core` module to Jetty's NIO framework to provide asynchronous socket I/O. This module uses Jetty internals, but Java SPDY client applications can also use it to communicate with a SPDY server.
3. `spdy-jetty-http` module—provides a server-side layering of HTTP over SPDY. This module allows any SPDY compliant browser, such as Chromium/Chrome, to talk SPDY to a Jetty server that deploys a standard web application made of servlets, filters and JSPs. This module performs the conversion of SPDY to HTTP and vice versa so that for the web application it is as if a normal HTTP request has arrived, and a normal HTTP response is returned.
4. `spdy-jetty-http-webapp` module—provides a demo application for the a `spdy-jetty-http` module.

Configuring SPDY

The `spdy-jetty-http` module provides an out-of-the-box server connector that performs the SPDY to HTTP conversion and vice versa (HTTP over SPDY). You can use this connector instead of Jetty's `SslSelectChannelConnector` (which only speaks HTTP), and it falls back to HTTPS if SPDY is not negotiated.

An example `jetty-spdy.xml` file that you can use instead of `jetty-ssl.xml` follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure.dtd">

<Configure id="Server" class="org.eclipse.jetty.server.Server">

    <New id="sslContextFactory" class="org.eclipse.jetty.util.ssl.SslContextFactory">
        <Set name="keyStorePath">your_keystore.jks</Set>
        <Set name="keyStorePassword">storepwd</Set>
        <Set name="includeProtocols">TLSv1</Set>
    </New>

    <Call name="addConnector">
        <Arg>
            <New class="org.eclipse.jetty.spdy.http.HTTPSPDYServerConnector">
                <Arg>
                    <Ref id="sslContextFactory"/>
                </Arg>
                <Set name="Port">8443</Set>
            </New>
        </Arg>
    </Call>
</Configure>

```

This is sufficient to enable your Jetty server to speak SPDY to browsers that support it. Old browsers or browsers that don't support SPDY yet speak plain HTTP on the same connector.

Remember, however, that SPDY over SSL (as set up like the configuration above) requires that you set up NPN correctly; in particular, you need to start the JVM with the [NPN boot Jar in the boot classpath](#).

Be aware that NPN is supported only for the TLS protocol, version 1 or greater; this means you cannot use it with SSLv2, which implies that you have to configure the SslContextFactory to use TLSv1 or above, since the JDK usually sends a SSLv2 ClientHello message to secure servers. To do so, specify the *includeProtocols* property to contain at least the value *TLSv1* (see code example above).

Configuring SPDY Proxy

Configuring SPDY by Example

`spdy-jetty-http` provides a fully functional SPDY proxy server out of the box. Jetty's SPDY proxy can receive SPDY (currently v2/v3) and HTTP requests, and proxy those requests to other SPDY servers. If necessary, an implementation of the Proxy Engine class translates the incoming protocol to a protocol the target host understands. Currently we provide a `SPDYProxyEngine` that can talk SPDY v2 and SPDY v3. We plan to support other protocols soon. As always, contributions are welcome.

Configuring SPDY is straightforward, as the following example shows. The example server has a plain SPDY connector listening on port 9090 and a SPDY proxy connector listening on port 8080. Incoming requests to port 8080 are proxied and translated to the connector listening on port 9090. In real world scenarios you more likely proxy to a different host.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure.dtd">

<Configure id="Server" class="org.eclipse.jetty.server.Server">

    <New id="sslContextFactory" class="org.eclipse.jetty.util.ssl.SslContextFactory">
        <Set name="keyStorePath">src/main/resources/keystore.jks</Set>
        <Set name="keyStorePassword">storepwd</Set>
        <Set name="trustStore">src/main/resources/truststore.jks</Set>
    </New>

```

```

<Set name="trustStorePassword">storepwd</Set>
<Set name="protocol">TLSv1</Set>
</New>

<!--
<Set class="org.eclipse.jetty.npn.NextProtoNego" name="debug" type="boolean">
true</Set>
-->

<!--
This is the upstream server connector. It speaks non-SSL SPDY/2(HTTP) on port
9090.
-->
<Call name="addConnector">
<Arg>
<New class="org.eclipse.jetty.spdy.http.HTTPSPDYSERVERCONNECTOR">
<Set name="Port">9090</Set>
<Set name="defaultAsyncConnectionFactory">
<Call name="getAsyncConnectionFactory">
<Arg>spdy/2</Arg>
</Call>
</Set>
</New>
</Arg>
</Call>
</New>
<!--
This ProxyEngine translates the incoming SPDY/x(HTTP) request to SPDY/2(HTTP)
-->
<New id="spdyProxyEngine" class="org.eclipse.spdy.proxy.SPDYProxyEngine">
<Arg>spdy/2</Arg>
<Arg>
<New class="org.eclipse.spdy.SPDYClient$Factory">
<Call name="start"/>
</New>
</Arg>
</New>
<!--
The ProxyEngineSelector receives SPDY/x(HTTP) requests from proxy connectors
below
and is configured to process requests for host "localhost".
Such requests are converted from SPDY/x(HTTP) to SPDY/2(HTTP) by the configured
ProxyEngine
and forwarded to 127.0.0.1:9090, where they are served by the upstream server
above.
-->

<New id="proxyEngineSelector" class="org.eclipse.spdy.proxy.ProxyEngineSelector">
<Call name="putProxyEngine">
<Arg>
<Ref id="spdyProxyEngine"/>
</Arg>
</Call>
<Set name="proxyServerInfos">
<Map>
<Entry>
<Item>localhost</Item>
<Item>
<New class="org.eclipse.spdy.proxy.ProxyEngineSelector
$ProxyServerInfo">
<Arg type="String">spdy/2</Arg>
<Arg>127.0.0.1</Arg>
<Arg type="int">9090</Arg>
</New>
</Item>
</Entry>
</Map>
</Set>
</New>

<!--
These are the reverse proxy connectors accepting requests from clients.
They accept non-SSL (on port 8080) and SSL (on port 8443) HTTP,
SPDY/2(HTTP) and SPDY/3(HTTP).
Non-SPDY HTTP requests are converted to SPDY internally and passed to the
ProxyEngine above.
-->

```

```
-->
<Call name="addConnector">
    <Arg>
        <New class="org.eclipse.jetty.spdy.proxy.HTTPSPDYProxyConnector">
            <Arg>
                <Ref id="proxyEngineSelector"/>
            </Arg>
            <Set name="Port">8080</Set>
        </New>
    </Arg>
</Call>
<Call name="addConnector">
    <Arg>
        <New class="org.eclipse.jetty.spdy.proxy.HTTPSPDYProxyConnector">
            <Arg>
                <Ref id="proxyEngineSelector"/>
            </Arg>
            <Arg>
                <Ref id="sslContextFactory"/>
            </Arg>
            <Set name="Port">8443</Set>
        </New>
    </Arg>
</Call>

</Configure>
```

Let's take this apart:

```
<!--
This ProxyEngine translates the incoming SPDY/x(HTTP) request to SPDY/2(HTTP)
-->
<New id="spdyProxyEngine" class="org.eclipse.jetty.spdy.proxy.SPDYProxyEngine">
    <Arg>spdy/2</Arg>
    <Arg>
        <New class="org.eclipse.jetty.spdy.SPDYClient$Factory">
            <Call name="start"/>
        </New>
    </Arg>
</New>

This is the ProxyEngine configuration. It is a SPDYProxyEngine that can translate to
SPDY v2 as
configured above. If your target host is capable of speaking SPDY v3, you change the
first constructor argument
to <code>spdy/3.</code> If you have different target hosts speaking different
protocols, you configure multiple proxy
engines and feed them to the ProxyEngineSelector as follows.

<!--
The ProxyEngineSelector receives SPDY/x(HTTP) requests from proxy connectors below
and is configured to process requests for host "localhost".
Such requests are converted from SPDY/x(HTTP) to SPDY/2(HTTP) by the configured
ProxyEngine
and forwarded to 127.0.0.1:9090, where they are served by the upstream server above.
-->

<New id="proxyEngineSelector" class="org.eclipse.jetty.spdy.proxy.ProxyEngineSelector">
    <Call name="putProxyEngine">
        <Arg>
            <Ref id="spdyProxyEngine"/>
        </Arg>
    </Call>
    <Set name="proxyServerInfos">
        <Map>
            <Entry>
                <Item>localhost</Item>
                <Item>
                    <New class="org.eclipse.jetty.spdy.proxy.ProxyEngineSelector
$ProxyServerInfo">
                        <Arg type="String">spdy/2</Arg>
                        <Arg>127.0.0.1</Arg>
                </New>
            </Item>
        </Map>
    </Set>
</New>
```

```

        <Arg type="int">9090</Arg>
    </New>
</Item>
</Entry>
</Map>
</Set>
</New>

```

This is the ProxyEngineSelector. The ProxyEngineSelector keeps the configurations for the known target hosts and also chooses the right ProxyEngine for the protocol the target host speaks.

Let's take even smaller parts of the snippet above to explain them in detail:

```

<Call name="putProxyEngine">
    <Arg>
        <Ref id="spdyProxyEngine"/>
    </Arg>
</Call>

```

This adds the SPDYProxyEngine configured above to the ProxyEngineSelector. The SPDYProxyEngine is configured to translate to `spdy/2`. By adding it to the Selector, it now knows how to translate to `spdy/2`.

```

<Set name="proxyServerInfos">
    <Map>
        <Entry>
            <Item>localhost</Item>
            <Item>
                <New class="org.eclipse.jetty.spdy.proxy.ProxyEngineSelector
$ProxyServerInfo">
                    <Arg type="String">spdy/2</Arg>
                    <Arg>127.0.0.1</Arg>
                    <Arg type="int">9090</Arg>
                </New>
            </Item>
        </Entry>
    </Map>
</Set>

```

You configure target hosts and the protocol you want to communicate with them by adding so-called ProxyServerInfos. Key for the map is the hostname. You configure protocol, host, and port for the ProxyServerInfo.

Request flow with the proxy given above:

- Incoming SPDY Request to `https://localhost:8443/` reaches the HTTPSPDYProxyConnector listening on port 8443.
- HTTPSPDYProxyConnector forwards the request to the ProxyEngineSelector.
- ProxyEngineSelector reads the Host header's host portion: "localhost".
- ProxyEngineSelector looks up the host in its ProxyServerInfo mappings. It finds the matching entry, "localhost". (If there is no matching entry it rst the stream).
- ProxyEngineSelector looks up a ProxyEngine that matches the protocol configured in the ProxyServerInfo it found in the last step and forwards the request to that ProxyEngine.
- The ProxyEngine translates the request to the given target protocol and proxies it to the target host.
- All responses are forwarded to the client.

An Example Configuration for a SPDY to HTTP Proxy

spdy-jetty-[http](#) provides full proxy functionality as described above. Here's another example configuration for a SPDY to HTTP proxy. This proxy accepts SPDY requests and proxies them to an HTTP server.

This is a very common use case, for example to terminate SPDY on a frontend server when you need to talk plain HTTP to your backend, because either your network hardware needs to inspect HTTP content or your backend is unable to talk SPDY. You have the performance advantages of SPDY over the slow internet with high latencies, and you talk HTTP to your backend as needed.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<!-- ====== -->
<!-- Configure the Jetty Server instance with an ID "Server"      -->
<!-- by adding a SPDY connector.                                     -->
<!-- This configuration must be used in conjunction with jetty.xml -->
<!-- It should not be used with jetty-https.xml as this connector -->
<!-- can provide both HTTPS and SPDY connections                  -->
<!-- ====== -->

<Configure id="Server" class="org.eclipse.jetty.server.Server">

<!-- ====== -->
<!-- Set up the SSL Context factory used to establish all TLS       -->
<!-- Connections and session.                                       -->
<!--
<!-- Consult the javadoc of o.e.j.util.ssl.SslContextFactory        -->
<!-- o.e.j.server.HttpConnectionFactory for all configuration     -->
<!-- that may be set here.                                         -->
<!-- ====== -->
<New id="sslContextFactory" class="org.eclipse.jetty.util.ssl.SslContextFactory">
    <Set name="KeyStorePath"><Property name="jetty.home" default="." />/etc/keystore</
Set>
    <Set name="KeyStorePassword">OBF:lvny1zl0lx8elvnwlvn6lx8glzlulvn4</Set>
    <Set name="KeyManagerPassword">OBF:lu2ulwmlz7s1z7alwnl1u2g</Set>
    <Set name="TrustStorePath"><Property name="jetty.home" default="." />/etc/
keystore</Set>
    <Set name="TrustStorePassword">OBF:lvny1zl0lx8elvnwlvn6lx8glzlulvn4</Set>
</New>

<!-- ====== -->
<!-- Enables NPN debugging on System.err                           -->
<!-- ====== -->
<Set class="org.eclipse.jetty.npn.NextProtoNego" name="debug" type="boolean">true</
Set>
-->

<!-- ====== -->
<!-- Create a TLS specific HttpConfiguration based on the         -->
<!-- common HttpConfiguration defined in jetty.xml                -->
<!-- Add a SecureRequestCustomizer to extract certificate and   -->
<!-- session information                                         -->
<!-- ====== -->
<New id="tlsHttpConfig" class="org.eclipse.jetty.server.HttpConfiguration">
    <Arg><Ref refid="httpConfig"/></Arg>
    <Call name="addCustomizer">
        <Arg><New class="org.eclipse.jetty.server.SecureRequestCustomizer"/></Arg>
    </Call>
</New>

<!-- ====== -->
<!-- This is the upstream server connector.                         -->
<!-- It speaks HTTP on port 9090.                                    -->
<!-- ====== -->
<Call name="addConnector">
    <Arg>
        <New class="org.eclipse.jetty.server.ServerConnector">
            <Arg name="server"><Ref id="Server" /></Arg>
            <Arg name="factories">
```

```

<Array type="org.eclipse.jetty.server.ConnectionFactory">
    <Item>
        <New class="org.eclipse.jetty.server.HttpConnectionFactory">
            <Arg name="config"><Ref id="httpConfig" /></Arg>
        </New>
    </Item>
</Array>
</Arg>
<Set name="host"><Property name="jetty.host" /></Set>
<Set name="port"><Property name="jetty.port" default="9090" /></Set>
<Set name="idleTimeout">30000</Set>
</New>
</Arg>
</Call>

<!-- ===== -->
<!-- This ProxyEngine translates the incoming SPDY/x(HTTP)      -->
<!-- requests to HTTP                                         -->
<!-- ===== -->
<New id="httpProxyEngine" class="org.eclipse.spdy.server.proxy.HTTPProxyEngine">
    <Arg>
        <New class="org.eclipse.jetty.client.HttpClient">
            <Call name="start"/>
        </New>
    </Arg>
</New>

<!-- ===== -->
<!-- The ProxyEngineSelector receives SPDY/x(HTTP) requests      -->
<!-- from proxy connectors below and is configured to process      -->
<!-- requests for host "localhost".                                -->
<!-- Such requests are converted from SPDY/x(HTTP) to             -->
<!-- HTTP by the configured ProxyEngine and forwarded              -->
<!-- to 127.0.0.1:9090, where they are served by the upstream     -->
<!-- server above.                                                 -->
<!-- ===== -->
<New id="proxyEngineSelector" class="org.eclipse.spdy.server.proxy.ProxyEngineSelector">
    <Call name="putProxyEngine">
        <Arg>http/1.1</Arg>
        <Arg>
            <Ref refid="httpProxyEngine"/>
        </Arg>
    </Call>
<Set name="proxyServerInfos">
    <Map>
        <Entry>
            <Item>localhost</Item>
            <Item>
                <New class="org.eclipse.spdy.server.proxy.ProxyEngineSelector
$ProxyServerInfo">
                    <Arg type="String">http/1.1</Arg>
                    <Arg>127.0.0.1</Arg>
                    <Arg type="int">9090</Arg>
                </New>
            </Item>
        </Entry>
    </Map>
</Set>
</New>

<!-- ===== -->
<!-- These are the reverse proxy connectors accepting requests   -->
<!-- from clients.                                               -->
<!-- They accept non-SSL (on port 8080) and SSL (on port 8443)   -->
<!-- HTTP, SPDY/2(HTTP) and SPDY/3(HTTP).                      -->
<!-- Non-SPDY HTTP requests are converted to SPDY internally     -->
<!-- and passed to the ProxyEngine above.                         -->
<!-- ===== -->
<Call name="addConnector">
    <Arg>
        <New class="org.eclipse.spdy.server.proxy.HTTPSPDYProxyServerConnector">
            <Arg>
                <Ref refid="Server"/>
            </Arg>
            <Arg>
                <Ref refid="proxyEngineSelector"/>
            </Arg>

```

```

        <Set name="Port">8080</Set>
    </New>
</Arg>
</Call>
<Call name="addConnector">
<Arg>
<New class="org.eclipse.jetty.spdy.server.proxy.HTTPSPDYProxyServerConnector">
<Arg>
<Ref refid="Server"/>
</Arg>
<Arg>
<Ref refid="sslContextFactory"/>
</Arg>
<Arg>
<Ref refid="proxyEngineSelector"/>
</Arg>
<Set name="Port">8443</Set>
</New>
</Arg>
</Call>
</Configure>

```

Configuring SPDY push

[SPDY push](#) allows the server to send multiple resources to the client for a single client request. This will reduce the amount of round-trips and can significantly improve page load times. A full page load from germany to <https://www.webtide.com> via SPDY and without push takes ~3s. The same request with push enabled takes only ~1s.

To enable push in Jetty the SPDY connector needs to be configured with an implementation of [Push-Strategy](#). For each request Jetty will call the [PushStrategy's](#) apply method which will return a Set with the resources to push.

ReferrerPushStrategy

The [ReferrerPushStrategy](#) is a [PushStrategy](#) implementation that will use the HTTP "referer" header to identify if a resource belongs to a main resource.

A step by step example of how this works:

- client requests index.html
- client parses index.html and requests style.css, image1.png and image2.png setting the referer header to index.html for these requests.
- [ReferrerPushStrategy](#) will use this information to associate the subresources to index .html
- The next request to index.html from another client will get all subresources pushed without further requests

This will also work for nested subresources. E.g. a pushed style.css might initiate further resource pushes for subresources referred to by the style.css stylesheet.



Note

The referrerPushPeriod setting will define the time that [ReferrerPushStrategy](#) will record subresources after the initial request. If this period has elapsed no further subresources will be recorded.

Configuring ReferrerPushStrategy

In the Jetty etc directory you will find jetty-spdy.xml file which can be modified to suit your needs. It contains a commented [ReferrerPushStrategy](#) configuration.

An example [ReferrerPushStrategy](#) configuration can look as follows:

```

<!-- ===== -->
<!-- Create a push strategy which can be used by reference by      -->
<!-- individual connection factories below.                      -->
<!--                                         -->
<!-- Consult the javadoc of o.e.j.spdy.server.http.ReferrerPushStrategy -->
<!-- for all configuration that may be set here.                  -->
<!-- ===== -->
<New id="pushStrategy" class="org.eclipse.jetty.spdy.server.http.ReferrerPushStrategy">
    <!-- Uncomment to blacklist browsers for this push strategy. If one of the
        blacklisted Strings occurs in the
            user-agent header sent by the client, push will be disabled for this browser.
        This is case insensitive" -->
    <!--
    <Set name="UserAgentBlacklist">
        <Array type="String">
            <Item>.*\firefox/14.*</Item>
            <Item>.*\firefox/15.*</Item>
            <Item>.*\firefox/16.*</Item>
        </Array>
    </Set>
    <!--

    <!-- Uncomment to override default file extensions to push -->
    <!--
    <Set name="PushRegexpes">
        <Array type="String">
            <Item>.*\.css</Item>
            <Item>.*\.js</Item>
            <Item>.*\.png</Item>
            <Item>.*\.jpg</Item>
            <Item>.*\.gif</Item>
        </Array>
    </Set>
    <!--
    <Set name="referrerPushPeriod">5000</Set>
    <Set name="maxAssociatedResources">32</Set>
</New>

```

Note the commented parts that let you restrict the User-Agents and file extensions.

Important Options:

referrerPushPeriod

If referrerPushPeriod has elapsed after the initial request to a mainresource, no more subresources will be added to the push cache.

maxAssociatedResources

The maximum amount of subresources being pushed for a single main resource.

Then you have to add the configured [ReferrerPushStrategy](#) to the connection factory as follows.



Note

In the default config provided with Jetty the pushStrategy argument is commented out!

```
<!-- SPDY/3 Connection factory -->
<Item>

<New class="org.eclipse.jetty.spdy.server.http.HTTPSPDYServerConnectionFactory">
    <Arg name="version" type="int">3</Arg>
    <Arg name="config"><Ref refid="sslHttpConfig" /></Arg>
    <Arg name="pushStrategy"><Ref refid="pushStrategy"/></Arg>
</New>
</Item>
```

See the javadocs for [ReferrerPushStrategy](#) and sources for further details if needed.



Note

Visit <https://www.webtide.com> with a browser that supports push (e.g. a recent chrome browser) to see it in action.

Try it!

To verify if your setup works fine you can use chrome and it's very useful chrome://net-internals/#spdy page. Open that page in a tab of your browser. Then make sure you load a main resource of your application to fill the push cache. Then reload the page bypassing the browser's cache (STRG+SHIFT+R or CMD+SHIFT+R on OSX).

Chrome's net-internals page should tell you how many resources have been pushed and actually pushed and claimed (actually needed and used by the browser)

-internals/#spdy

: (8473) Stop Reset

abled: true
 Alternate Protocol: true
 SPDY Always: false
 SPDY Over SSL: true
 Protocols: http/1.1,spdy/2,spdy/3

ns

[SPDY sessions](#)

Host	Proxy	ID	Protocol Negotiated	Active streams	Unclaimed pushed	Max	Initiated
apis.com:443	direct://	691	spdy/3	0	0	100	1
m:443	direct://	693	spdy/3	0	0	100	2
e.com:443	direct://	587	spdy/3	0	0	100	1

Another option is to enable debug logging for ReferrerPushStrategy or org.eclipse.jetty.spdy. Have a look at the Chapter 22, *Jetty Logging* for details.

Implement a custom SPDY PushStrategy

The ReferrerPushStrategy that is distributed with Jetty (the section called “Configuring SPDY push”) does a great job to automatically detect subresources to push for a given main resource. However there might be reasons to implement your own push strategy.

PushStrategy API

The interface that needs to be implemented is [PushStrategy](#). For each request Jetty will call the [apply](#) method, which will return a Set with the resources to push. This is the apply method's signature:

```
/**  
 * <p>Applies the SPDY push logic for the primary resource.</p>  
 *  
 * @param stream the primary resource stream  
 * @param requestHeaders the primary resource request headers  
 * @param responseHeaders the primary resource response headers  
 * @return a list of secondary resource URIs to push  
 */  
public Set<String> apply(Stream stream, Fields requestHeaders,  
Fields responseHeaders);
```

The parameters are the [Stream](#) for the primary resource request, the request and the response header fields. Based on this information the implementation has to decide which resources to push and return a `Set<String>` containing the URLs for the resources to push. Jetty will then open a push stream for each URL returned in that list and push the contents of that file. This is the only method that you need to implement.

ReferrerPushStrategy

For reference and as a working example use the [ReferrerPushStrategy](#) source code:

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/  
plain/jetty-spdy/spdy-http-server/src/main/java/org/eclipse/jetty/spdy/server/http/  
ReferrerPushStrategy.java
```

Chapter 15. ALPN

Table of Contents

..... 199

Introducing ALPN

The development of new web protocols such as SPDY and HTTP 2.0 raised the need of protocol negotiation within a Transport Layer Security (TLS) handshake. Two protocol negotiation solutions have been proposed: an older one, called NPN (see Chapter 16, *NPN*) and a newer one, called [ALPN](#) (Application Layer Protocol Negotiation).

ALPN is going to replace NPN very soon, and major browsers already support ALPN, as well as major servers such as Jetty.

The Jetty project provides an implementation of the TLS extension for ALPN for OpenJDK 7 and OpenJDK 8. ALPN allows the application layer to negotiate which protocol to use over the secure connection.

Any protocol can be negotiated by ALPN within a TLS connection. The protocols that are most commonly negotiated are SPDY (for browsers that support it) and, currently in an experimental way, HTTP 2.0. The ALPN implementation is therefore not SPDY-specific in any way. Jetty's ALPN implementation, although hosted under the umbrella of the Jetty project, is independent of Jetty (the Servlet Container); you can use the ALPN implementation in any other Java network server.

Starting the JVM

To enable ALPN support, start the JVM as follows:

```
java -Xbootclasspath/p:<path_to_alpn_boot_jar> ...
```

where `path_to_alpn_boot_jar` is the path on the file system for the ALPN Boot Jar file, for example, one at the Maven coordinates `org.mortbay.jetty.alpn:alpn-boot`.

Starting in OSGi

To use ALPN in an OSGi environment, in addition to putting the ALPN jar on the boot classpath for the container, you will also need to deploy the `jetty-osgi-alpn` jar. This jar contains a Fragment-Host directive that ensures the ALPN classes will be available from the system bundle.

You can download the [jetty-osgi-alpn jar](#) from Maven Central.

Understanding the ALPN API

Applications need to interact with ALPN TLS extension protocol negotiations. For example, server applications need to know whether the client supports ALPN, and client applications needs to know whether the server supports ALPN.

To implement this interaction, Jetty's ALPN implementation provides an API to applications, hosted at Maven coordinates `org.eclipse.jetty.alpn:alpn-api`. You need to declare this depen-

dency as provided, because the `alpn-boot` Jar already includes it (see the previous section), and it is therefore available from the boot classpath.

The API consists of a single class, `org.eclipse.jetty.alpn.ALPN`, and applications need to register instances of `SSLocket` or `SSLEngine` with a `ClientProvider` or `ServerProvider` (depending on whether the application is a client application or server application). Refer to ALPN Javadocs and to the examples below for further details about client and server provider methods.

Client Example

Example Client Example.

```
SSLContext sslContext = ...;
final SSLocket sslSocket =
    (SSLocket)context.getSocketFactory().createSocket("localhost", server.getLocalPort());

ALPN.put(sslSocket, new ALPN.ClientProvider()
{
    @Override
    public boolean supports()
    {
        return true;
    }

    @Override
    public List<String> protocols()
    {
        return Arrays.asList("spdy/3", "http/1.1");
    }

    @Override
    public void unsupported()
    {
        ALPN.remove(sslSocket);
    }

    @Override
    public void selected(String protocol)
    {
        ALPN.remove(sslSocket);
        System.out.println("Protocol Selected is: " + protocol);
    }
});
```

The ALPN implementation calls `ALPN.ClientProvider` methods `supports()`, `protocols()`, `unsupported()` and `selected(String)`, so that the client application can:

- decide whether to support ALPN.
- provide the protocols supported.
- know whether the server supports ALPN.
- know the protocol chosen by the server.

Server Example

The example for `SSLEngine` is identical, and you just need to replace the `SSLocket` instance with an `SSLEngine` instance.

Example Server Example.

```
final SSLocket sslSocket = ...;
```

```

ALPN.put(sslSocket, new ALPN.ServerProvider()
{
    @Override
    public void unsupported()
    {
        ALPN.remove(sslSocket);
    }

    @Override
    public String select(List<String> protocols)
    {
        ALPN.remove(sslSocket);
        return protocols.get(0);
    }
});

```

The ALPN implementation calls `ALPN.ServerProvider` methods `unsupported()`, and `select(List<String>)`, so that the server application can:

- know whether the client supports ALPN.
- select one of the protocols the client supports.

Implementation Details

It is important that implementations of `ALPN.ServerProvider` and `ALPN.ClientProvider` remove the `sslSocket` or `sslEngine` when the negotiation is complete, like shown in the examples above.

Failing to do so will cause a memory leak.

Unit Tests

You can write and run unit tests that use the ALPN implementation. The solution that we use with Maven is to specify an additional command line argument to the Surefire plugin:

```

<project>

<properties>
    <alpn-version>1.1.1.v20121030</alpn-version>
</properties>

<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <configuration>
                <argLine>
                    -Xbootclasspath/p:${settings.localRepository}/org/mortbay/jetty/alpn/
alpn-boot/${alpn-version}/alpn-boot-${alpn-version}.jar
                </argLine>
            </configuration>
        </plugin>
        ...
    </plugins>
</build>
...
</project>

```

Debugging

You can enable debug logging for the ALPN implementation in this way:

```
ALPN.debug = true;
```

Since the ALPN class is in the boot classpath, we chose not to use logging libraries because we do not want to override application logging library choices; therefore the logging is performed directly on `System.err`.

License Details

The ALPN implementation relies on modification of a few OpenJDK classes and on a few new classes that need to live in the `sun.security.ssl` package. These classes are released under the same GPLv2+exception license of OpenJDK.

The ALPN class and its nested classes are released under same license as the classes of the Jetty project.

Versions

The ALPN implementation, relying on modifications of OpenJDK classes, updates every time there are updates to the modified OpenJDK classes.

Table 15.1. ALPN vs. OpenJDK versions

OpenJDK version	ALPN version
1.7.0u40	7.0.0.v20140317
1.7.0u45	7.0.0.v20140317
1.7.0u51	7.0.0.v20140317
1.7.0u55	7.0.0.v20140317
1.7.0u60	7.0.0.v20140317
1.8.0	8.0.0.v20140317
1.8.0u05	8.0.0.v20140317

How to build ALPN

This section is for Jetty developers that need to update the ALPN implementation with the OpenJDK versions.

Clone the OpenJDK repository with the following command:

```
$ hg clone http://hg.openjdk.java.net/jdk7u/jdk7u jdk7u # OpenJDK 7
$ hg clone http://hg.openjdk.java.net/jdk8u/jdk8u jdk8u # OpenJDK 8
$ cd !$
$ ./get_source.sh
```

To update the source to a specific tag, use the following command:

```
$ ./make/scripts/hgforest.sh update <tag-name>
```

The list of OpenJDK tags can be obtained from these pages: [OpenJDK 7](#) / [OpenJDK 8](#).

Then you need to compare and incorporate the OpenJDK source changes into the modified OpenJDK classes at the [ALPN GitHub Repository](#), branch `openjdk7` for OpenJDK 7 and branch `master` for OpenJDK 8.

Chapter 16. NPN

Table of Contents

..... 203

Configuring NPN

The Jetty project provides an implementation of the Transport Layer Security (TLS) extension for Next Protocol Negotiation (NPN) for OpenJDK 7 (but not for OpenJDK 8 or greater - see Chapter 15, *ALPN* for using a TLS protocol negotiation extension with OpenJDK 8 or greater). NPN allows the application layer to negotiate which protocol to use over the secure connection.

NPN currently negotiates using SPDY as an application level protocol on port 443, and also negotiates the SPDY version. However, NPN is not SPDY-specific in any way. Jetty's NPN implementation, although hosted under the umbrella of the Jetty project, is independent of Jetty (the servlet container); you can use it in any other Java network server.

Starting the JVM

To enable NPN support, start the JVM as follows:

```
java -Xbootclasspath/p:<path_to_npn_boot_jar> ...
```

where `path_to_npn_boot_jar` is the path on the file system for the NPN Boot Jar file, for example, one at the Maven coordinates `org.mortbay.jetty.npn:npn-boot`.

Be aware that the current versions of the npn packages no longer align with Jetty versions. Look at the dates in those file paths before looking at the version number.

Starting in OSGi

To use NPN in an OSGi environment, in addition to putting the NPN jar on the boot classpath for the container, you will also need to deploy the `jetty-osgi-npn` jar. This jar contains a Fragment-Host directive that ensures the NPN classes will be available from the system bundle.

You can download the `jetty-osgi-npn` jar from maven central: <http://central.maven.org/maven2/org/eclipse/jetty/osgi/jetty-osgi-npn/>

Understanding the NPN API

Applications need to interact with NPN TLS extension protocol negotiations. For example, server applications need to know whether the client supports NPN, and client applications needs to know the list of protocols the server supports, and so on.

To implement this interaction, Jetty's NPN implementation provides an API to applications, hosted at Maven coordinates `org.eclipse.jetty.npn:npn-api`. You need to declare this dependency as provided, because the `npn-boot` Jar already includes it (see the previous section), and it is therefore available in the boot classpath.

The API consists of a single class, `org.eclipse.jetty.npn.NextProtoNego`, and applications need to register instances of `SSLocket` or `SSLEngine` with a `ClientProvider` or `ServerProvider`

(depending on whether the application is a client or server application). Refer to NextProtoNego Javadocs and to the examples below for further details about client and server provider methods.

Client Example

Example Client Example.

```
SSLContext sslContext = ...;
final SSLSocket sslSocket =
    (SSLSocket)context.getSocketFactory().createSocket("localhost", server.getLocalPort());

NextProtoNego.put(sslSocket, new NextProtoNego.ClientProvider()
{
    @Override
    public boolean supports()
    {
        return true;
    }

    @Override
    public void unsupported()
    {
        NextProtoNego.remove(sslSocket);
    }

    @Override
    public String selectProtocol(List<String> protocols)
    {
        NextProtoNego.remove(sslSocket);
        return protocols.get(0);
    }
});
```

The NPN implementation calls `NextProtoNego.ClientProvider` methods `supports()`, `unsupported()` and `selectProtocol(List<String>)`, so that the client application can:

- decide whether to support NPN.
- know whether the server supports NPN.
- select one of the protocols the server supports.

Server Example

The example for SSELengine is identical, and you just need to replace the `SSLSocket` instance with an `SSELengine` instance.

Example Server Example.

```
final SSLSocket sslSocket = ...;
NextProtoNego.put(sslSocket, new NextProtoNego.ServerProvider()
{
    @Override
    public void unsupported()
    {
        NextProtoNego.remove(sslSocket);
    }

    @Override
    public List<String> protocols()
    {
        return Arrays.asList("http/1.1");
    }

    @Override
    public void protocolSelected(String protocol)
```

```

    {
        NextProtoNego.remove(sslSocket);
        System.out.println("Protocol Selected is: " + protocol);
    });
}

```

The NPN implementation calls `NextProtoNego.ServerProvider` methods `unsupported()`, `protocols()` and `protocolSelected(String)`, so that the server application can:

- know whether the client supports NPN.
- provide the list of protocols the server supports.
- know which protocol the client chooses.

Implementation Details

It is common that the `NextProtoNego.ServerProvider` and the `NextProtoNego.ClientProvider` are implemented as (anonymous) inner classes, and that their methods' implementations require references to the the `sslSocket` (or `sslEngine`), either directly or indirectly.

Since the `NextProtoNego` class holds `[sslSocket/sslEngine, provider]` pairs in a `WeakHashMap`, if the value (that is, the provider implementation) holds a strong (even indirect) reference to the key, then the `WeakHashMap` entries are never removed, leading to a memory leak.

It is important that implementations of `NextProtoNego.ServerProvider` and `NextProtoNego.ClientProvider` remove the `sslSocket` or `sslEngine` when the negotiation is complete, like shown in the examples above.

Be aware that declaring the `SslConnection` as a final local variable and referencing it from within the anonymous `NextProtoNego.ServerProvider` class generates a hidden field in the anonymous inner class, that may cause a memory leak if the implementation does not call `NextProtoNego.remove()`.

Unit Tests

You can write and run unit tests that use the NPN implementation. The solution that we use with Maven is to specify an additional command line argument to the Surefire plugin:

```

<project>

<properties>
    <npm-version>1.1.1.v20121030</npm-version>
</properties>

<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <configuration>
                <argLine>
                    -Xbootclasspath/p:${settings.localRepository}/org/mortbay/jetty/npn/
                    npm-boot/${npm-version}/npm-boot-${npm-version}.jar
                </argLine>
            </configuration>
        </plugin>
    ...
    </plugins>
</build>
...
</project>

```

Debugging

You can enable debug logging for the NPN implementation in this way:

```
NextProtoNego.debug = true;
```

Since the NextProtoNego class is in the boot classpath, we chose not to use logging libraries because we do not want to override application logging library choices; therefore the logging is performed directly on `System.err`.

License Details

The NPN implementation relies on modification of a few OpenJDK classes and on a few new classes that need to live in the `sun.security.ssl` package. These classes are released under the same GPLv2+exception license of OpenJDK.

The NextProtoNego class is released under same license as the classes of the Jetty project.

Versions

The NPN implementation, relying on modifications of OpenJDK classes, updates every time there are updates to the modified OpenJDK classes.

Table 16.1. NPN vs. OpenJDK versions

NPN version	OpenJDK version
1.0.0.v20120402	1.7.0 - 1.7.0u2 - 1.7.0u3
1.1.0.v20120525	1.7.0u4 - 1.7.0u5
1.1.1.v20121030	1.7.0u6 - 1.7.0u7
1.1.3.v20130313	1.7.0u9 - 1.7.0u10 - 1.7.0u11
1.1.4.v20130313	1.7.0u13
1.1.5.v20130313	1.7.0u15 - 1.7.0u17 - 1.7.0u21 - 1.7.0u25
1.1.6.v20130911	1.7.0u40 - 1.7.0u45 - 1.7.0u51
1.1.7.v20140316	1.7.0u55 - 1.7.0u60

How to build NPN

This section is for Jetty developers that need to update the NPN implementation with the OpenJDK versions.

Clone the OpenJDK repository with the following command:

```
$ hg clone http://hg.openjdk.java.net/jdk7u/jdk7u jdk7u
$ cd jdk7u
$ ./get_source.sh
```

To update the source to a specific tag, use the following command:

```
$ ./make/scripts/hgforest.sh update <tag-name>
```

The list of OpenJDK tags can be obtained from [this page](#).

Then you need to compare and incorporate the OpenJDK source changes into the modified OpenJDK classes at the [NPN GitHub Repository](#).

Chapter 17. FastCGI Support

Table of Contents

FastCGI Introduction	208
Configuring Jetty for FastCGI	208

FastCGI Introduction

FastCGI is a network protocol primarily used by a *web server* to communicate to a *FastCGI server*. FastCGI servers are typically used to serve web content generated by dynamic web languages, primarily [PHP](#), but also Python, Ruby, Perl and others.

Web servers that supports FastCGI are, among others, [Apache](#) and [Nginx](#), and Jetty. Web servers typically act as proxies, converting HTTP requests that they receive from clients (browsers) to FastCGI requests that are forwarded to the FastCGI server. The FastCGI server spawns the dynamic web language interpreter, passing it the information contained in the FastCGI request and a dynamic web language script is executed, producing web content, typically HTML. The web content is then formatted into a FastCGI response that is returned to the web server, which converts it to a HTTP response that is then returned to the client.

The most well known FastCGI server is the [PHP FastCGI Process Manager](#), or `php-fpm`. In the following we will assume that `php-fpm` is used as FastCGI server.

Jetty can be configured to act as a web server that supports FastCGI, replacing the functionality that is normally provided by Apache or Nginx. This allows users to leverage Jetty features such as SPDY, the unique support that Jetty provides for SPDY Push, Jetty's scalability, and of course Jetty's native support for Java Web Standards such as Servlets, JSPs, etc.

With such configuration, users can deploy their Java Web Applications in Jetty, but also serve their [WordPress](#) site or blog or their [Drupal](#) site without having to install and manage multiple web servers.

Configuring Jetty for FastCGI

In this section you will see how to configure Jetty to serve WordPress via FastCGI.

The first step is to have WordPress installed in your server machine, for example under `/var/www/wordpress`. For more information about how to install WordPress, please refer to the [WordPress Installation Guide](#).

The second step is to install `php-fpm` and make sure it is configured to listen on a TCP socket; typically it is configured to listen to `localhost:9000`.

The third step is to install Jetty, for example under `/opt/jetty`. Refer to the section called “Downloading Jetty” for more information about how to install Jetty.

The fourth step is to create a Jetty base directory (see the section called “Managing Jetty Base and Jetty Home”), called in the following `$JETTY_BASE`, where you setup the configuration needed to support FastCGI in Jetty:

```
$ mkdir -p /usr/jetty/wordpress  
$ cd /usr/jetty/wordpress
```

Therefore `$JETTY_BASE=/usr/jetty/wordpress`.

The fifth step is to deploy the web application that provides the proxying of client requests to the FastCGI server, php-fpm. Typically this is done by deploying a *.war file in the \$JETTY_BASE/webapps directory, but in case of FastCGI there is really nothing of this web application that you have to write: all the work is already done for you by the Jetty developers. Therefore you just need to deploy a Jetty XML file that configures the web application directly. Copy and paste the following content as \$JETTY_BASE/webapps/jetty-wordpress.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN" "http://www.eclipse.org/jetty/configure_9_0.dtd">
<Configure class="org.eclipse.jetty.servlet.ServletContextHandler">

<New id="root" class="java.lang.String">
    <Arg>/var/www/wordpress</Arg>
</New>

<Set name="contextPath"/></Set>
<Set name="resourceBase"><Ref refid="root" /></Set>
<Set name="welcomFiles">
    <Array type="string"><Item>index.php</Item></Array>
</Set>

<Call name="addFilter">
    <Arg>org.eclipse.jetty.fcgi.server.proxy.TryFilesFilter</Arg>
    <Arg>/*</Arg>
    <Arg>
        <Call name="of" class="java.util.EnumSet">
            <Arg><Get name="REQUEST" class="javax.servlet.DispatcherType" /></Arg>
        </Call>
    </Arg>
    <Call name="setInitParameter">
        <Arg>files</Arg>
        <Arg>$path /index.php?p=$path</Arg>
    </Call>
</Call>

<Call name="addServlet">
    <Arg>
        <New class="org.eclipse.jetty.servlet.ServletHolder">
            <Arg>default</Arg>
            <Arg>
                <Call name="forName" class="java.lang.Class">
                    <Arg>org.eclipse.jetty.servlet.DefaultServlet</Arg>
                </Call>
            </Arg>
            <Call name="setInitParameter">
                <Arg>dirAllowed</Arg>
                <Arg>false</Arg>
            </Call>
        </New>
    </Arg>
    <Arg>/</Arg>
</Call>

<Call name="addServlet">
    <Arg>org.eclipse.jetty.fcgi.server.proxy.FastCGIProxyServlet</Arg>
    <Arg>*.php</Arg>
    <Call name="setInitParameter">
        <Arg>proxyTo</Arg>
        <Arg>http://localhost:9000</Arg>
    </Call>
    <Call name="setInitParameter">
        <Arg>prefix</Arg>
        <Arg>/</Arg>
    </Call>
    <Call name="setInitParameter">
        <Arg>scriptRoot</Arg>
        <Arg><Ref refid="root" /></Arg>
    </Call>
    <Call name="setInitParameter">
        <Arg>scriptPattern</Arg>
        <Arg>( .+?\.\.php )</Arg>
    </Call>
</Call>
```

```
</Configure>
```

Explanation of this file content:

- At line 6 it is specified the WordPress installation directory, in this example `/var/www/wordpress` (as defined in the first step).
- At line 9 it is specified the context path at which WordPress will be served, in this example at the root context path `/`.
- At line 10 it is specified the resource base of the context, also set to the WordPress installation directory. This allows Jetty to serve static resources directly from the WordPress installation directory.
- At line 12 it is specified the welcome file as `index.php`, so that Jetty can perform the proper redirects in case of URIs ending with the `/` character.
- At line 15 it is specified the `TryFilesFilter`, a Servlet Filter that has been inspired by the [try_files](#) functionality offered by Nginx. This filter tries to serve the resource from the file system first, and if the resource is not found it forwards the request as `index.php?p=$path`, which will match the proxy servlet defined below. Refer to the [TryFilesFilter](#) documentation for further information.
- At line 29 it is specified Jetty's `DefaultServlet` to serve static content such as CSS files, JavaScript files, etc. `DefaultServlet` will serve these files by looking in the resource base of the context, defined at line 10 (see above).
- At line 47 it is specified the `FastCGIProxyServlet`, a Servlet that proxies HTTP requests arriving from clients to FastCGI requests to the FastCGI server.
- At line 52 it is specified the TCP address of the FastCGI server (`php-fpm`), where HTTP requests are forwarded as FastCGI requests.
- At line 60 it is specified once again the WordPress installation directory, so that the `FastCGIProxyServlet` can pass this information to the FastCGI server.
- At line 64 it is specified a regular expression that matches request URIs performed to this servlet, in addition to the standard URL mapping defined by `Servlet` at line 49. Refer to the [FastCGIProxyServlet](#) documentation for further information.

Configuring Jetty to Proxy HTTP to FastCGI

In order to configure Jetty to listen for HTTP requests from clients and forward them to the FastCGI server as FastCGI requests, you need to specify a `start.ini` file that configures the Jetty modules that are needed for Jetty to proxy HTTP to FastCGI. The minimal `start.ini` file is the following:

```
# The 'deploy' module enables Jetty to look up in the 'webapps' directory for files to
# deploy.
--module=deploy

# The 'fcgi' module enables FastCGI support.
--module=fcgi

# The 'http' module enables a HTTP connector that listens for client requests on port
# 8080.
--module=http
jetty.port=8080
```

At this point, you can start Jetty (see Chapter 9, *Starting Jetty*), hit `http://localhost:8080` with your browser and enjoy WordPress:

```
$ cd $JETTY_BASE  
$ java -jar /opt/jetty/start.jar
```

Configuring Jetty to Proxy SPDY to FastCGI

In order to configure Jetty to listen for SPDY requests from clients that are SPDY enabled and forward them to the FastCGI server as FastCGI requests, you need to specify a `start.ini` file that configures the Jetty modules that are needed for Jetty to proxy SPDY to FastCGI.

Remember that for SPDY to work you will need to run over SSL, and you will need the NPN boot jar in the boot classpath, as explained in the section called “Configuring SPDY”.

Since now your site will run over SSL, you need to make sure that the WordPress URL is also configured so. If you have followed the steps of the the section called “Configuring Jetty to Proxy HTTP to FastCGI”, your WordPress site is served at `http://localhost:8080`. You will need to change that to be `https://localhost:8443` from the WordPress administration web interface, or follow the [WordPress instructions](#) to do so without using the administration web interface.

The minimal `start.ini` file is the following:

```
# The 'deploy' module enables Jetty to look up in the 'webapps' directory for files to  
# deploy.  
--module=deploy  
  
# The 'fcgi' module enables FastCGI support.  
--module=fcgi  
  
# The 'spdy' module enables a SPDY connector that listens for client requests on port  
# 8443.  
--module=spdy  
spdy.port=8443
```

Note that we specified the `spdy.port` to be `8443`.

At this point, you will need the NPN boot jar and the SSL certificate. In this example, you can obtain them very easily using the built-in feature of the Jetty start mechanism that downloads required dependencies, see Chapter 9, *Starting Jetty*. However, remember that the example SSL certificate that is downloaded will not be valid for your website and that it must not be used in production. It is provided only for testing and example purposes. To leverage the automatic download of required dependencies, issue this command:

```
$ cd $JETTY_BASE  
$ java -jar /opt/jetty/start.jar --create-files
```

At this point, you can start Jetty (see Chapter 9, *Starting Jetty*), hit `http://localhost:8080` with your browser and enjoy WordPress via SPDY using a SPDY enabled browser:

```
$ cd $JETTY_BASE  
$ java -Xbootclasspath/p:lib/npn/npn-boot-<version>.jar -jar /opt/jetty/start.jar
```

If you don't have a SPDY enabled browser, WordPress will still be available over plain HTTPS.

Chapter 18. Bundled Servlets, Filters, and Handlers

Table of Contents

Default Servlet	212
Proxy Servlet	213
Balancer Servlet	214
CGI Servlet	215
Quality of Service Filter	215
Denial of Service Filter	218
Gzip Filter	220
Cross Origin Filter	222
Resource Handler	223
Debug Handler	225
Statistics Handler	226
IP Access Handler	227
Moved Context Handler	229
Shutdown Handler	230
Default Handler	231
Error Handler	231

Jetty ships with a bundle of servlets that interact with the key classes. Most are in the org.eclipse.jetty.servlets package. These servlets and filters are among the principle elements of Jetty as a component-based infrastructure that holds and runs J2EE applications. As described, they play a major role in running and maintaining the Jetty server.

Also included are a number of Jetty specific handlers that allow access to internals of jetty that would not normally be exposed and are very useful testing environments and many production scenarios.

Default Servlet

Info

- Classname: org.eclipse.jetty.servlet.DefaultServlet
- Maven Artifact: org.eclipse.jetty:jetty-servlet
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/servlet/DefaultServlet.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/servlet/DefaultServlet.html>

Usage

The DefaultServlet implements the ResourceFactory interface and extends the HttpServlet abstract class. It is usually mapped to / and provides handling for static content, OPTION and TRACE methods for the context. The MOVE method is allowed if PUT and DELETE are allowed. See DefaultServlet javadoc.

Init Parameters

Jetty supports the following initParameters:

acceptRanges

If true, range requests and responses are supported.

dirAllowed

If true, directory listings are returned if no welcome file is found. Otherwise 403 Forbidden displays.

redirectWelcome

If true, welcome files are redirected rather than forwarded.

gzip

If set to true, then static content is served as gzip content encoded if a matching resource is found ending with ".gz".

resourceBase

Set to replace the context resource base.

aliases

If true, aliases of resources are allowed (that is, symbolic links and case variations) and may bypass security constraints.

maxCacheSize

Maximum total size of the cache or 0 for no cache.

maxCachedFileSize

Maximum size of a file to cache.

maxCachedFiles

Maximum number of files to cache.

useFileMappedBuffer

If set to true, mapped file buffer serves static content. Setting this value to false means that a direct buffer is used instead of a mapped file buffer. By default, this is set to true.

Proxy Servlet

Info

- Classname: `org.eclipse.jetty.proxy.ProxyServlet`
- Maven Artifact: `org.eclipse.jetty:jetty-proxy`
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/proxy/ProxyServlet.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/proxy/ProxyServlet.html>

Usage

An asynchronous servlet that forwards requests to another server either as a standard web reverse proxy (as defined by RFC2616) or as a transparent reverse proxy. Internally it uses the async jetty-client.

To facilitate JMX monitoring, the `HttpClient` instance is set as context attribute, prefixed with the servlet's name and exposed by the mechanism provided by `ContextHandler.MANAGED_ATTRIBUTES`.

Init Parameters

The following init parameters may be used to configure the servlet:

hostHeader

forces the host header to a particular value

viaHost

the name to use in the Via header: Via: http/1.1 <viaHost>

whiteList

comma-separated list of allowed proxy hosts

blackList

comma-separated list of forbidden proxy hosts

In addition, there are a number of init parameters that can be used to configure the HttpClient instance used internally for the proxy.

maxThreads

Default Value: 256

The max number of threads of HttpClient's Executor

maxConnections

Default Value: 32768

The max number of connections per destination. RFC 2616 suggests that 2 connections should be opened per each destination, but browsers commonly open 6 or more. If this HttpClient is used for load testing, it is common to have only one destination (the server to load test), and it is recommended to set this value to a high value (at least as much as the threads present in the executor).

idleTimeout

Default Value: 30000

The idle timeout in milliseconds that a connection can be idle, that is without traffic of bytes in either direction.

timeout

Default Value: 60000

The total timeout in milliseconds for the request/response conversation.

requestBufferSize

Default Value: 4096

The size of the request buffer the request is written into.

responseBufferSize

Default Value: 4096

The size of the response buffer the response is written into.

Balancer Servlet

Info

- Classname: org.eclipse.jetty.proxy.BalancerServlet
- Maven Artifact: org.eclipse.jetty:jetty-proxy
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/proxy/BalancerServlet.html>

- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/proxy/BalancerServlet.html>

Usage

The Balancer servlet allows for simple, sticky round robin load balancing leveraging the ProxyServlet that is distributed with Jetty.

In addition to the parameters for ProxyServlet, the following are available for the balancer servlet:

stickySessions

true if sessions should be sticky for subsequent requests

balancerMember.<name>.proxyTo

One or more of these are required and will be the locations that are used to proxy traffic to.

CGI Servlet

Info

- Classname: `org.eclipse.jetty.servlets.CGI`
- Maven Artifact: `org.eclipse.jetty:jetty-servlets`
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/servlets/CGI.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/servlets/CGI.html>

Usage

The CGI servlet class extends the abstract HttpServlet class. When the init parameter is called, the cgi bin directory is set with the cgibinResourceBase. Otherwise, it defaults to the resource base of the context. See CGI javadoc.

The cgi bin uses three parameters:

commandPrefix

The init parameter obtained when there is a prefix set to all commands directed to the method exec.

Path

An init parameter passed to the exec environment as a PATH. This must be run unpacked somewhere in the filesystem.

ENV_

An init parameter that points to an environment variable with the name stripped of the leading ENV_ and using the init parameter value.

Quality of Service Filter

Info

- Classname: `org.eclipse.jetty.servlets.QoSFilter`
- Maven Artifact: `org.eclipse.jetty:jetty-servlets`

- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/servlets/QoSFilter.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/servlets/QoSFilter.html>

Usage

Jetty supports Continuations, which allow non-blocking handling of HTTP requests, so that threads can be allocated in a managed way to provide application specific Quality of Service (QoS). The QoSFilter is a utility servlet filter that implements some QoS features.

Understanding the Problem

Waiting for Resources

Web applications frequently use JDBC Connection pools to limit the simultaneous load on the database. This protects the database from peak loads, but makes the web application vulnerable to thread starvation. Consider a thread pool with 20 connections, being used by a web application that typically receives 200 requests per second and each request holds a JDBC connection for 50ms. Such a pool can service on average $200*20*1000/50 = 400$ requests per second.

However, if the request rate rises above 400 per second, or if the database slows down (due to a large query) or becomes momentarily unavailable, the thread pool can very quickly accumulate many waiting requests. If, for example, the website is slashdotted or experiences some other temporary burst of traffic and the request rate rises from 400 to 500 requests per second, then 100 requests per second join those waiting for a JDBC connection. Typically, a web server's thread pool contains only a few hundred threads, so a burst or slow DB need only persist for a few seconds to consume the entire web server's thread pool. This is called thread starvation. The key issue with thread starvation is that it effects the entire web application, and potentially the entire web server. Even if the requests using the database are only a small proportion of the total requests on the web server, all requests are blocked because all the available threads are waiting on the JDBC connection pool. This represents non graceful degradation under load and provides a very poor quality of service.

Prioritizing Resources

Consider a web application that is under extreme load. This load might be due to a popularity spike (slashdot), usage burst (Christmas or close of business), or even a denial of service attack. During such periods of load, it is often desirable not to treat all requests as equals, and to give priority to high value customers or administrative users.

The typical behaviour of a web server under extreme load is to use all its threads to service requests and to build up a backlog of unserviced requests. If the backlog grows deep enough, then requests start to timeout and users experience failures as well as delays.

Ideally, the web application should be able to examine the requests in the backlog, and give priority to high value customers and administrative users. But with the standard blocking servlet API, it is not possible to examine a request without allocating a thread to that request for the duration of its handling. There is no way to delay the handling of low priority requests, so if the resources are to be reallocated, then the low priority requests must all be failed.

Applying the QoSFilter

The Quality of Service Filter (QoSFilter) uses Continuations to avoid thread starvation, prioritize requests and give graceful degradation under load, to provide a high quality of service. When you apply the filter to specific URLs within a web application, it limits the number of active requests being handled for those URLs. Any requests in excess of the limit are suspended. When a request completes handling the limited URL, one of the waiting requests resumes and can be handled. You can assign priorities to each suspended request, so that high priority requests resume before lower priority requests.

Required JARs

To use the QoS Filter, these JAR files must be available in WEB-INF/lib:

- \$JETTY_HOME/lib/ext/jetty-util.jar
- \$JETTY_HOME/lib/ext/jetty-servlets.jar—contains QoSFilter

Sample Configuration

Place the configuration in a webapp's web.xml or jetty-web.xml. The default configuration processes ten requests at a time, servicing more important requests first, and queuing up the rest. This example processes fifty requests at a time:

```
<filter>
  <filter-name>QoSFilter</filter-name>
  <filter-class>org.eclipse.jetty.servlets.QoSFilter</filter-class>
  <init-param>
    <param-name>maxRequests</param-name>
    <param-value>50</param-value>
  </init-param>
</filter>
```

Configuring QoS Filter Parameters

A semaphore polices the "maxRequests" limit. The filter waits a short time while attempting to acquire the semaphore. The "waitMs" init parameter controls the wait, avoiding the expense of a suspend if the semaphore is shortly available. If the semaphore cannot be obtained, Jetty suspends the request for the default suspend period of the container or the value set as the "suspendMs" init parameter.

The QoS filter uses the following init parameters:

maxRequests

the maximum number of requests to be serviced at a time. The default is 10.

maxPriority

the maximum valid priority that can be assigned to a request. A request with a high priority value is more important than a request with a low priority value. The default is 10.

waitMS

waitMS—length of time, in milliseconds, to wait while trying to accept a new request. Used when the maxRequests limit is reached. Default is 50 ms.

suspendMS

length of time, in milliseconds, that the request will be suspended if it is not accepted immediately. If not set, the container's default suspend period applies. Default is -1 ms.

managedAttr

If set to true, then this servlet is set as a ServletContext attribute with the filter name as the attribute name. This allows a context external mechanism (for example, JMX via ContextHandler.MANAGED_ATTRIBUTES) to manage the configuration of the filter.

Mapping to URLs

You can use the <filter-mapping> syntax to map the QoSFilter to a servlet, either by using the servlet name, or by using a URL pattern. In this example, a URL pattern applies the QoSFilter to every request within the web application context:

```
<filter-mapping>
    <filter-name>QoSFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Setting the Request Priority

Requests with higher values have a higher priority. The default request priorities assigned by the QoSFilter are:

- 2 -- For any authenticated request
- 1 -- For any request with a non-new valid session
- 0 -- For all other requests

To customize the priority, subclass QoSFilter and then override the `getPriority(ServletRequest request)` method to return an appropriate priority for the request. You can then use this subclass as your QoS filter. Here's a trivial example:

```
public class ParsePriorityQoSFilter extends QoSFilter
{
    protected int getPriority(ServletRequest request)
    {
        String p = ((HttpServletRequest)request).getParameter("priority");
        if (p!=null)
            return Integer.parseInt(p);
        return 0;
    }
}
```

Denial of Service Filter

Info

- Classname: `org.eclipse.jetty.servlets.DoSFilter`
- Maven Artifact: `org.eclipse.jetty:jetty-servlets`
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/servlets/DoSFilter.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/servlets/DoSFilter.html>

Usage

The Denial of Service (DoS) filter limits exposure to request flooding, whether malicious, or as a result of a misconfigured client. The DoS filter keeps track of the number of requests from a connection per second. If the requests exceed the limit, Jetty rejects, delays, or throttles the request, and sends a warning message. The filter works on the assumption that the attacker might be written in simple blocking style, so by suspending requests you are hopefully consuming the attacker's resources. The DoS filter is related to the QoS filter, using Continuations to prioritize requests and avoid thread starvation.

Using the DoS Filter

Jetty places throttled requests in a priority queue, giving priority first to authenticated users and users with an HttpSession, then to connections identified by their IP addresses. Connections with no way to identify them have lowest priority. To uniquely identify authenticated users, you should implement the extractUserId(ServletRequest request) function.

Required JARs

To use the DoS Filter, these JAR files must be available in WEB-INF/lib:

- \$JETTY_HOME/lib/ext/jetty-util.jar
- \$JETTY_HOME/lib/ext/jetty-servlets.jar

Sample Configuration

Place the configuration in a webapp's web.xml or jetty-web.xml. The default configuration allows 25 requests per connection at a time, servicing more important requests first, and queuing up the rest. This example allow 30 requests at a time:

```
<filter>
  <filter-name>DoSFilter</filter-name>
  <filter-class>org.eclipse.jetty.servlets.DoSFilter</filter-class>
  <init-param>
    <param-name>maxRequestsPerSec</param-name>
    <param-value>30</param-value>
  </init-param>
</filter>
```

Configuring DoS Filter Parameters

The following init parameters control the behavior of the filter:

maxRequestsPerSec

Maximum number of requests from a connection per second. Requests in excess of this are first delayed, then throttled. Default is 25.

delayMs

Delay imposed on all requests over the rate limit, before they are considered at all:

- 100 (ms) = Default
- -1 = Reject request
- 0 = No delay
- any other value = Delay in ms

maxWaitMs

Length of time, in ms, to blocking wait for the throttle semaphore. Default is 50 ms.

throttledRequests

Number of requests over the rate limit able to be considered at once. Default is 5.

throttleMs

Length of time, in ms, to async wait for semaphore. Default is 30000L.

maxRequestMs

Length of time, in ms, to allow the request to run. Default is 30000L.

maxIdleTrackerMs

Length of time, in ms, to keep track of request rates for a connection, before deciding that the user has gone away, and discarding it. Default is 30000L.

insertHeaders

If true, insert the DoSFilter headers into the response. Defaults to true.

trackSessions

If true, usage rate is tracked by session if a session exists. Defaults to true.

remotePort

If true and session tracking is not used, then rate is tracked by IP+port (effectively connection). Defaults to false.

ipWhitelist

A comma-separated list of IP addresses that will not be rate limited.

managedAttr

If set to true, then this servlet is set as a ServletContext attribute with the filter name as the attribute name. This allows a context external mechanism (for example, JMX via ContextHandler.MANAGED_ATTRIBUTES) to manage the configuration of the filter.

Gzip Filter

Info

- Classname: org.eclipse.jetty.servlets.GzipFilter
- Maven Artifact: org.eclipse.jetty:jetty-servlets
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/servlets/GzipFilter.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/servlets/GzipFilter.html>

Usage

The Jetty GzipFilter is a compression filter that you can apply to almost any dynamic resource (servlet). It fixes many of the bugs in commonly available compression filters—for example, it handles all ways to set content length. We have tested it with Jetty continuations and suspending requests. Some user-agents might be excluded from compression to avoid common browser bugs (yes, this means IE!). See GzipFilter javadoc.

For example simply add this to your web.xml or override your web.xml using jetty's the section called “Jetty override-web.xml” feature.

```
<filter>
  <filter-name>GzipFilter</filter-name>
  <filter-class>org.eclipse.jetty.servlets.GzipFilter</filter-class>
  <init-param>
    <param-name>mimeTypes</param-name>
    <param-value>text/html,text/plain,text/xml,application/xhtml+xml,text/css,application/javascript,image/svg+xml</param-value>
  </init-param>
```

```
</filter>
<filter-mapping>
    <filter-name>GzipFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Gzip Rules

GZIP Filter This filter will gzip or deflate the content of a response if:

- The filter is mapped to a matching path
- accept-encoding header is set to either gzip, deflate or a combination of those
- The response status code is ≥ 200 and < 300
- The content length is unknown or more than the minGzipSize initParameter or the minGzipSize is 0(default)
- The content-type is in the comma separated list of mimeTypes set in the mimeTypes initParameter or if no mimeTypes are defined the content-type is not "application/gzip"
- No content-encoding is specified by the resource

If both gzip and deflate are specified in the accept-encoding header, then gzip will be used.

Compressing the content can greatly improve the network bandwidth usage, but at a cost of memory and CPU cycles. If this filter is mapped for static content, then use of efficient direct NIO may be prevented, thus use of the gzip mechanism of the DefaultServlet is advised instead.

Init Parameters

This filter extends UserAgentFilter and if the excludedAgents initParameter is set to a comma separated list of user agents, then these agents will be excluded from gzip content.

bufferSize

The output buffer size which defaults to 8192. Be careful with this parameter as values ≤ 0 will lead to an IllegalArgumentException.

minGzipSize

Content will only be compressed if content length is either unknown or greater than minGzipSize.

deflateCompressionLevel

The compression level used for deflate compression. (0-9).

deflateNoWrap

The nowrap setting for deflate compression. Defaults to true. (true/false)

methods

Comma separated list of HTTP methods to compress. If not set, only GET requests are compressed.

mimeTypes

Comma separated list of mime types to compress. See description above.

excludedAgents

Comma separated list of user agents to exclude from compression. Does a String.contains(CharSequence) to check if the excluded agent occurs in the user-agent header. If it does -> no compression

excludeAgentPatterns

Same as excludedAgents, but accepts regex patterns for more complex matching.

excludePaths

Comma separated list of paths to exclude from compression. Does a String.startsWith(String) comparison to check if the path matches. If it does match -> no compression. To match subpaths use excludePathPatterns instead.

excludePathPatterns

Same as excludePath, but accepts regex patterns for more complex matching.

vary

Set to the value of the Vary header sent with responses that could be compressed. By default it is set to 'Vary: Accept-Encoding, User-Agent' since IE6 is excluded by default from the excludedAgents. If user-agents are not to be excluded, then this can be set to 'Vary: Accept-Encoding'. Note also that shared caches may cache copies of a resource that is varied by User-Agent - one per variation of the User-Agent, unless the cache does some normalization of the UA string.

Cross Origin Filter

Info

- Classname: `org.eclipse.jetty.servlets.CrossOriginFilter`
- Maven Artifact: `org.eclipse.jetty:jetty-servlets`
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/servlets/CrossOriginFilter.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/servlets/CrossOriginFilter.html>

Usage

HTTP requests made from a script are subject to well known restrictions, the most prominent being the same domain policy.

Firefox 3.5 introduced support for W3C's Access Control for Cross-Site Requests specification, which requires a compliant client (for example, Firefox 3.5) and a compliant server (via this servlet filter).

This filter implements the required bits to support the server-side contract of the specification, and will allow a compliant client to perform cross-domain requests via the standard XMLHttpRequest object. If the client does not issue a compliant cross-domain request, this filter does nothing, and its overhead is the check of the presence of the cross-domain HTTP header.

This is extremely useful in Cometd web applications where it is now possible to perform cross-domain long polling without using script injection (also known as the JSONP transport), and therefore removing all the downsides that the JSONP transport has (it's chattier, does not react quickly to failures, has a message size limit, uses GET instead of POST, etc.).

Setup

You will need to put the jetty-servlets.jar file onto your classpath. If you are creating a webapp, ensure that this jar is included in your webapp's WEB-INF/lib. Or, if you are running jetty embedded you will need to ensure that jetty-servlets.jar is on the execution classpath. You can download the jetty-servlets.jar from the Maven Central Repository at <http://central.maven.org/maven2/org/eclipse/jetty/jetty-servlets/>.

Configuration

This is a regular servlet filter that must be configured in web.xml.

It supports the following configuration parameters:

allowedOrigins

a comma separated list of origins that are allowed to access the resources. Default value is *, meaning all origins

allowedMethods

a comma separated list of HTTP methods that are allowed to be used when accessing the resources. Default value is GET,POST

allowedHeaders

a comma separated list of HTTP headers that are allowed to be specified when accessing the resources. Default value is X-Requested-With

preflightMaxAge

the number of seconds that preflight requests can be cached by the client. Default value is 1800 seconds, or 30 minutes

allowCredentials

a boolean indicating if the resource allows requests with credentials. Default value is false

A typical configuration could be:

```
<web-app>
    <filter>
        <filter-name>cross-origin</filter-name>
        <filter-class>org.eclipse.jetty.servlets.CrossOriginFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>cross-origin</filter-name>
        <url-pattern>/cometd/*</url-pattern>
    </filter-mapping>
</web-app>
```

Resource Handler

Info

- Classname: org.eclipse.jetty.server.handler.ResourceHandler
- Maven Artifact: org.eclipse.jetty:jetty-server
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/server/handler/ResourceHandler.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/server/handler/ResourceHandler.html>

Usage

This handler will serve static content and handle If-Modified-Since headers and is suitable for simple serving of static content.

Important



There is no caching done with this handler so if you are looking for a more featureful way of serving static content look to the the section called “Default Servlet”.

Note



Requests for resources that do not exist are let pass (Eg no 404's).

Improving the Look and Feel

The resource handler has a default stylesheet which you can change by calling `setStyleSheet(String location)` with the location of a file on the system that it can locate through the resource loading system. The default css is called jetty-dir.css and is located in the jetty-util package, pulled as a classpath resource from the jetty-util jar when requested through the ResourceHandler.

Embedded Example

The following is an example of a split fileserver, able to serve static content from multiple directory locations. Since this handler does not return 404's on content you are able to iteratively try multiple resource handlers to resolve content.

```
package org.eclipse.jetty.embedded;

import org.eclipse.jetty.server.Connector;
import org.eclipse.jetty.server.Handler;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.server.ServerConnector;
import org.eclipse.jetty.server.handler.ContextHandler;
import org.eclipse.jetty.server.handler.ContextHandlerCollection;
import org.eclipse.jetty.server.handler.ResourceHandler;
import org.eclipse.jetty.toolchain.test.MavenTestingUtils;
import org.eclipse.util.resource.Resource;

/**
 * A {@link ContextHandlerCollection} handler may be used to direct a request to
 * a specific Context. The URI path prefix and optional virtual host is used to
 * select the context.
 */
public class SplitFileServer {

    public static void main(String[] args) throws Exception {
        Server server = new Server();
        ServerConnector connector = new ServerConnector(server);
        connector.setPort(8090);
        server.setConnectors(new Connector[] { connector });
        ContextHandler context0 = new ContextHandler();
        context0.setContextPath("/");
        ResourceHandler rh0 = new ResourceHandler();

        rh0.setBaseResource(Resource.newResource(MavenTestingUtils.getTestResourceDir("dir0")));
        context0.setHandler(rh0);
        ContextHandler context1 = new ContextHandler();
        context1.setContextPath("/");
        ResourceHandler rh1 = new ResourceHandler();

        rh1.setBaseResource(Resource.newResource(MavenTestingUtils.getTestResourceDir("dir1")));
        context1.setHandler(rh1);
        ContextHandlerCollection contexts = new ContextHandlerCollection();
        contexts.setHandlers(new Handler[] { context0, context1 });
        server.setHandler(contexts);
        server.start();
        System.err.println(server.dump());
        server.join();
    }
}
```

Debug Handler

Info

- Classname: `org.eclipse.jetty.server.handler.DebugHandler`
- Maven Artifact: `org.eclipse.jetty:jetty-server`
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/server/handler/DebugHandler.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/server/handler/DebugHandler.html>

Usage

A simple handler that is useful to debug incoming traffic. It will log entry and exit points of http requests as well as the response code.

Usage in standard distribution

Simply include `jetty-debug.xml` in your `*.ini` configs. For example in `start.ini`.

Embedded usage

```
Server server = new Server(8080);
RolloverFileOutputStream outputStream = new RolloverFileOutputStream("MeinLogPfad/
yyyy_mm_dd.request.log", true,10);

DebugHandler debugHandler = new DebugHandler();
debugHandler.setOutputStream(outputStream);
debugHandler.setHandler(server.getHandler());

server.setHandler(debugHandler);
server.start();
```

Some example output

```
15:14:05.838:qtp551889550-13-selector-0 OPENED
HttpConnection@e910ee4{IDLE},g=HttpGenerator{s=START},p=HttpParser{s=START,0 of 0}
15:14:05.846:qtp551889550-57:http://0:0:0:0:0:0:1:8080/ REQUEST 0:0:0:0:0:0:1
GET __utma=111872281.10102721.1321534299.1369833564.1370447492.35;
__utzm=111872281.1321534299.1.1.utmcsrc=(direct)|utmccn=(direct)|utmcmd=(none);
_opt_vi_RPY720HZ=75E12E63-0CD0-4D6F-8383-C90D5C8397C7; Mozilla/5.0 (Macintosh; Intel Mac
OS X 10.8; rv:22.0) Gecko/20100101 Firefox/22.0
15:14:05.894:qtp551889550-57:http://0:0:0:0:0:0:1:8080/ RESPONSE 200 null
15:14:05.959:qtp551889550-59:http://0:0:0:0:0:0:1:8080/jetty.css REQUEST
0:0:0:0:0:0:1 GET __utma=111872281.10102721.1321534299.1369833564.1370447492.35;
__utzm=111872281.1321534299.1.1.utmcsrc=(direct)|utmccn=(direct)|utmcmd=(none);
_opt_vi_RPY720HZ=75E12E63-0CD0-4D6F-8383-C90D5C8397C7; visited=yes; Mozilla/5.0
(Macintosh; Intel Mac OS X 10.8; rv:22.0) Gecko/20100101 Firefox/22.0
15:14:05.962:qtp551889550-59:http://0:0:0:0:0:0:1:8080/jetty.css RESPONSE 200 null
15:14:06.052:qtp551889550-57:http://0:0:0:0:0:0:1:8080/images/jetty-header.jpg REQUEST
0:0:0:0:0:0:1 GET __utma=111872281.10102721.1321534299.1369833564.1370447492.35;
__utzm=111872281.1321534299.1.1.utmcsrc=(direct)|utmccn=(direct)|utmcmd=(none);
_opt_vi_RPY720HZ=75E12E63-0CD0-4D6F-8383-C90D5C8397C7; visited=yes; Mozilla/5.0
(Macintosh; Intel Mac OS X 10.8; rv:22.0) Gecko/20100101 Firefox/22.0
15:14:06.055:qtp551889550-57:http://0:0:0:0:0:0:1:8080/images/jetty-header.jpg RESPONSE
200 null
```

```
15:14:07.248:qtp551889550-59:http://0:0:0:0:0:0:0:1:8080/favicon.ico REQUEST
0:0:0:0:0:0:1 GET __utma=111872281.10102721.1321534299.1369833564.1370447492.35;
__utmz=111872281.1321534299.1.1.utmcsrc=(direct)|utmccn=(direct)|utmcmd=(none);
_opt_v1_RPY720HZ=75E12E63-0CD0-4D6F-8383-C90D5C8397C7; visited=yes; Mozilla/5.0
(Macintosh; Intel Mac OS X 10.8; rv:22.0) Gecko/20100101 Firefox/22.0
15:14:07.251:qtp551889550-59:http://0:0:0:0:0:0:1:8080/favicon.ico RESPONSE 404 text/
html;charset=ISO-8859-1
15:14:09.330:qtp551889550-57 CLOSED
HttpConnection@e910ee4{INTERESTED},g=HttpGenerator{s=START},p=HttpParser{s=START,0 of
-1}
```

Statistics Handler

Info

- Classname: `org.eclipse.jetty.server.handler.StatisticsHandler`
- Maven Artifact: `org.eclipse.jetty:jetty-server`
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/server/handler/StatisticsHandler.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/server/handler/StatisticsHandler.html>

Usage

Jetty currently has two levels of request statistic collection:

- Subclasses of `AbstractConnector` class optionally can collect statistics about connections as well as number of requests.
- The `StatisticsHandler` class may be used to collect request statistics.

In addition to that, subclasses of `AbstractSessionHandler` class optionally can collect session statistics.

`AbstractConnector` and `AbstractSessionHandler` statistics are turned off by default and must either be configured manually for each instance or turned on via JMX interface. The Statistics Handler is not included in default Jetty configuration, and needs to be configured manually.



Viewing Statistics

To view statistics, you have to be able to connect to Jetty using either JConsole or some other JMX agent. See the section called “Using Java Management Extensions (JMX)” for more information.

Connector statistics

Detailed statistics on connection duration and number of requests are only collated when a connection is closed. The current and maximum number of connections are the only “live” statistics. To learn how to turn on connector statistics please see Jetty Statistics tutorial, although this is not recommended and it is best to use a JMX agent to select statistics only when needed.

The following example shows how to turn on connector statistics in jetty xml. This example comes from within `jetty-http.xml`.

```
<Call name="addConnector">
```

```
<Arg>
<New class="org.eclipse.jetty.server.ServerConnector">
<Arg name="server"><Ref refid="Server" /></Arg>
<Arg name="factories">
<Array type="org.eclipse.jetty.server.ConnectionFactory">
<Item>
<New class="org.eclipse.jetty.server.HttpConnectionFactory">
<Arg name="config"><Ref refid="httpConfig" /></Arg>
</New>
</Item>
</Array>
</Arg>
<Set name="host"><Property name="jetty.host" /></Set>
<Set name="port"><Property name="jetty.port" default="8080" /></Set>
<Set name="idleTimeout">30000</Set>
<!-- Enable Connector Statistics -->
<Call name="addBean">
<Arg>
<New id="ConnectorStatistics" class="org.eclipse.jetty.server.ConnectorStatistics"/>
</Arg>
</Call>
</New>
</Arg>
</Call>
</Arg>
</Call>
```

Request Statistics

To collect request statistics a StatisticsHandler must be configured as one of the handlers of the server. Typically this can be done as the top level handler, but you may choose to configure a statistics handler for just one context by creating a context configuration file. Please note that `jetty-stats.xml` has to appear in the command line after the main Jetty configuration file as shown below. It should be able to be uncommented in the `start.ini` file.

```
$ java -jar start.jar OPTIONS=default etc/jetty.xml etc/jetty-stats.xml
```

Alternately, if you are making multiple changes to the Jetty configuration, you could include statistics handler configuration into your own jetty xml configuration. The following fragment shows how to configure a top level statistics handler:

```
<Get id="oldhandler" name="handler" />
<Set name="handler">
<New id="StatsHandler" class="org.eclipse.jetty.server.handler.StatisticsHandler">
<Set name="handler"><Ref refid="oldhandler" /></Set>
</New>
</Set>
```

Session Statistics

Session handling is built into Jetty for any servlet or webapp context. Detailed statistics on session duration are only collated when a session is closed. The current, minimum, and maximum number of sessions are the only "live" statistics. The session statistics are enabled by default and do not need to be configured.

IP Access Handler

Info

- Classname: org.eclipse.jetty.server.handler.IPAccessHandler
- Maven Artifact: org.eclipse.jetty:jetty-server
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/server/handler/IPAccessHandler.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/server/handler/IPAccessHandler.html>

Usage

Controls access to the wrapped handler by the real remote IP. Control is provided by white/black lists that include both internet addresses and URIs. This handler uses the real internet address of the connection, not one reported in the forwarded for headers, as this cannot be as easily forged.

Typically, the black/white lists will be used in one of three modes:

- Blocking a few specific IPs/URLs by specifying several black list entries.
- Allowing only some specific IPs/URLs by specifying several white lists entries.
- Allowing a general range of IPs/URLs by specifying several general white list entries, that are then further refined by several specific black list exceptions

An empty white list is treated as match all. If there is at least one entry in the white list, then a request must match a white list entry. Black list entries are always applied, so that even if an entry matches the white list, a black list entry will override it.

Internet addresses may be specified as absolute address or as a combination of four octet wildcard specifications (a.b.c.d) that are defined as follows.

- nnn - an absolute value (0-255)
- mmm-nnn - an inclusive range of absolute values, with following shorthand notations:
 - nnn- => nnn-255
 - -nnn => 0-nnn
 - - => 0-255
- a,b,... - a list of wildcard specifications

Internet address specification is separated from the URI pattern using the "|" (pipe) character. URI patterns follow the servlet specification for simple * prefix and suffix wild cards (e.g. /, /foo, /foo/bar, /foo/bar/*, *.baz).

Earlier versions of the handler used internet address prefix wildcard specification to define a range of the internet addresses (e.g. 127., 10.10., 172.16.1.). They also used the first "/" character of the URI pattern to separate it from the internet address. Both of these features have been deprecated in the current version.

Examples of the entry specifications are:

- 10.10.1.2 - all requests from IP 10.10.1.2
- 10.10.1.2|/foo/bar - all requests from IP 10.10.1.2 to URI /foo/bar

- 10.10.1.2|/foo/* - all requests from IP 10.10.1.2 to URIs starting with /foo/
- 10.10.1.2|*.html - all requests from IP 10.10.1.2 to URIs ending with .html
- 10.10.0-255.0-255 - all requests from IPs within 10.10.0.0/16 subnet
- 10.10.0.-255|/foo/bar - all requests from IPs within 10.10.0.0/16 subnet to URI /foo/bar
- 10.10.0-3,1,3,7,15|/foo/* - all requests from IPs addresses with last octet equal to 1,3,7,15 in subnet 10.10.0.0/22 to URIs starting with /foo/

Earlier versions of the handler used internet address prefix wildcard specification to define a range of the internet addresses (e.g. 127., 10.10., 172.16.1.). They also used the first "/" character of the URI pattern to separate it from the internet address. Both of these features have been deprecated in the current version.

Moved Context Handler

Info

- Classname: org.eclipse.jetty.server.handler.MovedContextHandler
- Maven Artifact: org.eclipse.jetty:jetty-server
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/server/handler/MovedContextHandler.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/server/handler/MovedContextHandler.html>

Usage

You can use the MovedContextHandler to relocate or redirect a context that has changed context path and/or virtual hosts.

You can configure it to *permanently* redirect the old URL to the new URL, in which case Jetty sends a Http Status code of 301 to the browser with the new URL. Alternatively, you can make it non-permanent, in which case Jetty sends a 302 Http Status code along with the new URL.

In addition, as with any other context, you can configure a list of virtual hosts, meaning that this context responds only to requests to one of the listed host names.

Suppose you have a context deployed at /foo, but that now you want to deploy at the root context / instead.

- First you reconfigure and redeploy the context on Jetty.
- Next you need a way to redirect all the browsers who have bookmarked /foo to the new path. You create a new context xml file in \$JETTY_HOMEcontexts and configure the MovedContextHandler to do the redirection from /foo to /.

Here's an example. This is a permanent redirection, which also preserves pathinfo and query strings on the redirect:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/configure_9_0.dtd">
```

```
<Configure class="org.eclipse.jetty.server.handler.MovedContextHandler">
  <Set name="contextPath">/foo</Set>
  <Set name="newContextURL"/></Set>
  <Set name="permanent">true</Set>
  <Set name="discardPathInfo">false</Set>
  <Set name="discardQuery">false</Set>

  <Set name="virtualHosts">
    <Array type="String">
      <Item>209.235.245.73</Item>
      <Item>127.0.0.73</Item>
      <Item>acme.org</Item>
      <Item>www.acme.org</Item>
      <Item>server.acme.org</Item>
    </Array>
  </Set>
</Configure>
```

Shutdown Handler

Info

- Classname: `org.eclipse.jetty.server.handler.ShutdownHandler`
- Maven Artifact: `org.eclipse.jetty:jetty-server`
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/server/handler/ShutdownHandler.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/server/handler/ShutdownHandler.html>

Usage

A handler that shuts the server down on a valid request. Used to do "soft" restarts from Java. If `_exitJvm` is set to true a hard `System.exit()` call is being made.

This is an example of how you can setup this handler directly with the Server, it can be added as a part of handler chain or collection as well.

```
Server server = new Server(8080);
HandlerList handlers = new HandlerList();
handlers.setHandlers(new Handler[] {
  someOtherHandler, new ShutdownHandler(server, "secret password") });
server.setHandler(handlers);
server.start();
```

And this is an example that you can use to call the shutdown handler from within java.

```
public static void attemptShutdown(int port, String shutdownCookie) {
  try {
    URL url = new URL("http://localhost:" + port + "/shutdown?token=" +
    shutdownCookie);
    HttpURLConnection connection = (HttpURLConnection)url.openConnection();
    connection.setRequestMethod("POST");
    connection.getResponseCode();
    logger.info("Shutting down " + url + ": " + connection.getResponseMessage());
  } catch (SocketException e) {
```

```
    logger.debug("Not running");
    // Okay - the server is not running
} catch (IOException e) {
    throw new RuntimeException(e);
}
}
```

Default Handler

Info

- Classname: `org.eclipse.jetty.server.handler.DefaultHandler`
- Maven Artifact: `org.eclipse.jetty:jetty-server`
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/server/handler/DefaultHandler.html>
- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/server/handler/DefaultHandler.html>

Usage

A simple handler that is useful to terminate handler chains with a clean fashion. As in the example below, if a resource to be served is not matched within the resource handler, the DefaultHandler will take care of producing a 404 page. This class is a useful template to either extend and embrace or simply provide a similar implementation for customizing to your needs. There is also an [ErrorHandler](#) that services errors related to the servlet api specification so it is best to not get the two confused.



flav.ico

The DefaultHandler will also handle serving out the flav.ico file should a request make it through all of the other handlers without being resolved.

```
Server server = new Server(8080);
HandlerList handlers = new HandlerList();
ResourceHandler resourceHandler = new ResourceHandler();
resourceHandler.setBaseResource(Resource.newResource("."));
handlers.setHandlers(new Handler[]
{ resourceHandler, new DefaultHandler() });
server.setHandler(handlers);
server.start();
```

Error Handler

Info

- Classname: `org.eclipse.jetty.server.handler.ErrorHandler`
- Maven Artifact: `org.eclipse.jetty:jetty-server`
- Javadoc: <http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/server/handler/ErrorHandler.html>

- Xref: <http://download.eclipse.org/jetty/stable-9/xref/org/eclipse/jetty/server/handler/ErrorHandler.html>

Usage

A handler that is used to report errors from servlet contexts and webapp contexts to report error conditions. Primarily handles setting the various servlet spec specific response headers for error conditions. Can be customized by extending, for more information on this see the section called “Creating Custom Error Pages”.

Chapter 19. Jetty Runner

Table of Contents

Use Jetty without an installed distribution 233

This chapter explains how to use the jetty-runner to run your webapps without needing an installation of jetty.

Use Jetty without an installed distribution

The idea of the jetty-runner is extremely simple – run a webapp directly from the command line using a single jar and as much default configuration as possible. Of course, if your webapp is not so straightforward, the jetty-runner has command line options which allow you to customize the execution environment.

Preparation

You will need the jetty-runner jar:

1. [Get](#) the jetty-runner jar, from [maven central](#)

Deploying a simple context

Let's assume we have a very simple webapp, that does not need any resources from its environment, nor any configuration apart from the defaults. Starting it is as simple as doing the following:

```
> java -jar jetty-runner.jar simple.war
```

This will start jetty on port 8080, and deploy the webapp to "/".

Your webapp does not have to be packed into a war, you can deploy a webapp that is a directory instead in the same way:

```
> java -jar jetty-runner.jar simple
```

In fact, the webapp does not have to be a war or even a directory, it can simply be a jetty [context xml](#) file that describes your webapp:

```
> java -jar jetty-runner.jar simple-context.xml
```

 **Note**

When using a context xml file, the application being deployed is not even required to be a fully-fledged webapp. It can simply be a Jetty [context](#).

Deploying multiple contexts

If you have more than one webapp that must be deployed, simply provide them all on the command line. You can control the context paths for them using the "--path" parameter. Here's an example of deploying 2 wars (although either or both of them could be unpacked directories instead):

```
> java -jar jetty-runner.jar --path /one my1.war --path /two my2.war
```

If you have context xml files that describe your webapps, you can fully configure your webapps in them, and hence you don't need to use the command line switches. Just provide the list of context files like so:

```
> java -jar jetty-runner.jar my-first-context.xml my-second-context.xml my-third-context.xml
```

 **Note**

The command line switches override configuration file settings. So, for example, you could set the context path for the webapp inside the context xml file, and use the --path switch to override it on the command line.

Changing the default port

By default the jetty-runner will listen on port 8080. You can easily change this on the command line using the "--port" command. Here's an example that runs our simple.war on port 9090:

```
> java -jar jetty-runner.jar --port 9090 simple.war
```

Using jetty.xml files

Instead of, or in addition to using command line switches, you can use one or more jetty.xml files to configure the environment for your webapps. Here's an example where we apply two different jetty.xml files:

```
> java -jar jetty-runner.jar --config jetty.xml --config jetty-https.xml simple.war
```

 **Note**

Switches on the command line take precedence over those defined in configuration files, so you can use the command line as overrides.

Full configuration reference

You can see the full set of configuration options using the --help switch:

```
> java -jar jetty-runner.jar --help
```

Here's what the output will look like:

```
Usage: java [-Djetty.home=dir] -jar jetty-runner.jar [--help|--version] [ server opts]
[[ context opts] context ...]

Server opts:
--version                                - display version and exit
--log file                                 - request log filename (with optional 'yyyy_mm_dd')
wildcard                                  - info/warn/debug log filename (with optional
'yyyy_mm_dd' wildcard)
--host name|ip                            - interface to listen on (default is all interfaces)
--port n                                    - port to listen on (default 8080)
--stop-port n                            - port to listen for stop command
--stop-key n                             - security string for stop command (required if --
stop-port is present)
[--jar file]*n                           - each tuple specifies an extra jar to be added to
the classloader
[--lib dir]*n                           - each tuple specifies an extra directory of jars to
be added to the classloader
[--classes dir]*n                         - each tuple specifies an extra directory of classes
to be added to the classloader
--stats [unsecure|realm.properties]        - enable stats gathering servlet context
[--config file]*n                         - each tuple specifies the name of a jetty xml
config file to apply (in the order defined)

Context opts:
[[--path /path] context]*n                - WAR file, web app dir or context xml file,
optionally with a context path
```

Printing the version:

Print out the version of jetty and then exit immediately.

```
> java -jar jetty-runner.jar --version
```

Configuring a request log:

Cause jetty to write a request log with the given name. If the file is prefixed with yyyy_mm_dd then the file will be automatically rolled over. Note that for finer grained configuration of the [request log](#), you will need to use a jetty xml file instead.

```
> java -jar jetty-runner.jar --log yyyy_mm_dd-requests.log my.war
```

Configuring the output log:

Redirect the output of jetty logging to the named file. If the file is prefixed with yyyy_mm_dd then the file will be automatically rolled over.

```
> java -jar jetty-runner.jar --out yyyy_mm_dd-output.log my.war
```

Configuring the interface for http:

Like jetty standalone, the default is for the connectors to listen on all interfaces on a machine. You can control that by specifying the name or ip address of the particular interface you wish to use with the --host argument:

```
> java -jar jetty-runner.jar --host 192.168.22.19 my.war
```

Configuring the port for http:

The default port number is 8080. To [configure a https connector](#), use a jetty xml config file instead.

```
> java -jar jetty-runner.jar --port 9090 my.war
```

Configuring stop:

You can configure a port number for jetty to listen on for a stop command, so you are able to stop it from a different terminal. This requires the use of a "secret" key, to prevent malicious or accidental termination. Use the --stop-port and --stop-key parameters as arguments to the jetty-runner:

```
> java -jar jetty-runner.jar --stop-port 8181 --stop-key abc123
```

Then, to stop jetty from a different terminal, you need to supply the same port and key information. For this you'll either need a local installation of jetty, the [jetty-maven-plugin](#), the [jetty-ant plugin](#), or write a custom class. Here's how to use a jetty installation to perform a stop:

```
> java -jar start.jar --stop-port 8181 --stop-key abc123 --stop
```

Configuring the container classpath:

With a local installation of jetty, you add jars and classes to the container's classpath by putting them in the \$JETTY_HOME/lib directory. With the jetty-runner, you can use the --lib, --jar and --classes arguments instead to achieve the same thing.

--lib adds the location of a directory which contains jars to add to the container classpath. You can add 1 or more. Here's an example of configuring 2 directories:

```
> java -jar jetty-runner.jar --lib /usr/local/external/lib --lib $HOME/external-other/lib my.war
```

--jar adds a single jar file to the container classpath. You can add 1 or more. Here's an example of configuring 3 extra jars:

```
> java -jar jetty-runner.jar --jar /opt/stuff/jars/jar1.jar --jar $HOME/jars/jar2.jar --jar /usr/local/proj/jars/jar3.jar my.war
```

--classes add the location of a directory containing classes to add to the container classpath. You can add 1 or more. Here's an example of configuring a single extra classes dir:

```
> java -jar jetty-runner.jar --classes /opt/stuff/classes my.war
```

Gathering statistics:

If statistics gathering is enabled, then they are viewable by surfing to the context /stats. You may optionally protect access to that context with a password. Here's an example of enabling statistics, with no password protection:

```
> java -jar jetty-runner.jar --stats unsecure my.war
```

If we wished to protect access to the /stats context, we would provide the location of a jetty realm configuration file containing authentication and authorization information. For example, we could use the following example realm file from the jetty distribution:

```
jetty: MD5:164c88b302622e17050af52c89945d44,user
admin: CRYPT:adpexzg3FUZAk,server-administrator,content-administrator,admin
other: OBF:1xmk1w26lu9rlwlclxmq,user
plain: plain,user
user: password,user
# This entry is for digest auth. The credential is a MD5 hash of
username:realmname:password
digest: MD5:6e120743ad67abfbc385bc2bb754e297,user
```

Assuming we've copied it into the local directory, we would apply it like so

```
> java -jar jetty-runner.jar --stats realm.properties my.war
```

After surfing to http://localhost:8080/ a few times, we can surf to the stats servlet on http://localhost:8080/stats to see the output:

```
Statistics:
Statistics gathering started 1490627ms ago
Requests:
Total requests: 9
Active requests: 1
Max active requests: 1
Total requests time: 63
Mean request time: 7.875
Max request time: 26
Request time standard deviation: 8.349764752888037

Dispatches:
Total dispatched: 9
Active dispatched: 1
Max active dispatched: 1
```

```
Total dispatched time: 63
Mean dispatched time: 7.875
Max dispatched time: 26
Dispatched time standard deviation: 8.349764752888037
Total requests suspended: 0
Total requests expired: 0
Total requests resumed: 0

Responses:

1xx responses: 0
2xx responses: 7
3xx responses: 1
4xx responses: 0
5xx responses: 0
Bytes sent total: 1453

Connections:

org.eclipse.jetty.server.ServerConnector@203822411
Protocols:http/1.1
Statistics gathering started 1490606ms ago
Total connections: 7
Current connections open: 1
Max concurrent connections open: 2
Total connections duration: 72883
Mean connection duration: 12147.166666666666
Max connection duration: 65591
Connection duration standard deviation: 23912.40292977684
Total messages in: 7
Total messages out: 7

Memory:

Heap memory usage: 49194840 bytes
Non-heap memory usage: 12611696 bytes
```

Chapter 20. Setuid

Table of Contents

Configuring Setuid 239

Configuring Setuid

On Unix based systems, port 80 is protected and can usually only be opened by the superuser root. As it is not desirable to run the server as root (for security reasons), the solution options are as follows:

- Start Jetty as the root user, and use Jetty's setuid mechanism to switch to a non-root user after startup.
- Configure the server to run as a normal user on port 8080 (or some other non protected port). Then, configure the operating system to redirect port 80 to 8080 using ipchains, iptables, ipfw or a similar mechanism.

The latter has traditionally been the solution, however Jetty 9 has Setuid feature.

If you are using Solaris 10, you may not need to use this feature, as Solaris provides a User Rights Management framework that can permit users and processes superuser-like abilities. Please refer to the Solaris documentation for more information.



Note

If the environment variable JETTY_USER is set for the startup process and jetty.sh is used, jetty-setuid will not work! So if you want to use jetty-setuid, make sure JETTY_USER is not set!

Configuring the Setuid feature

In the Jetty etc directory you will find the following jetty-setuid.xml file which can be modified to suit your needs.

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN" "http://
www.eclipse.org/jetty/configure.dtd">

<!-- ===== -->
<!-- Configure the Jetty SetUIDServer -->
<!-- This configuration file *must* be specified first in the list of -->
<!-- configuration files and should be used in combination with other -->
<!-- configuration files eg jetty.xml -->
<!-- ===== -->
<Configure id="Server" class="org.eclipse.jetty.setuid.SetUIDServer">
    <Set name="startServerAsPrivileged">false</Set>
    <Set name="umask">2</Set>
    <Set name="username">jetty</Set>
    <Set name="groupname">jetty</Set>
<!-- uncomment to change the limits on number of open file descriptors for root -->
<!--
    <Call name="setRLimitNoFiles">
        <Arg>
            <New class="org.eclipse.jetty.setuid.RLimit">
                <Set name="soft">20000</Set>
                <Set name="hard">40000</Set>
            </New>
        </Arg>
    </Call>
-->
```

```
</Call>
-->
</Configure>
```

Options:

startServerAsPrivileged

set this to true if you will start the server up as the root user

umask

the umask setting you would like the process to have, optionally you may remove this line to leave it unchanged.

username

the name of the user you would like the process to run under after starting, set to *jetty* by default

groupname

the name of the group you would like the process to run under after starting, set to *jetty* by default

Additionally if you would like to set the file descriptor limits in the process you can uncomment the appropriate section above and set the soft and hard values accordingly.

Enabling SetUID on startup

The `jetty-setuid.xml` file runs as a wrapper around the typical Jetty server configuration so you must set this xml file to be processed before any others. This is already configured yet commented out in the normal `start.ini` file in the root of the jetty-distribution.

Open the `start.ini` file and look for the following section:

```
=====
# Enable SetUID
# To enable setuid you must have the jetty-setuid.xml as the
# first xml file to be processed.
# The default user and group is 'jetty' and if you are
# starting as root you must change the run privileged to true
#-----
# OPTIONS=setuid
# etc/jetty-setuid.xml
=====
```

Uncomment the `OPTIONS` line which will set the setuid libraries to be loaded when Jetty starts along with the line following which will process the `jetty-setuid.xml` file when jetty starts up. Take care when modifying this file when the SetUID feature is in play as it *MUST* be the first xml file to be processed.

Supported Operating Systems

The Setuid feature leverages the the JNI setup with the jvm so part of the feature is C code compiled for the appropriate operating environment. By default we ship with .so files for both Linux and Mac OS X. The code for the entire SetUID feature is located in the [Jetty toolchain](#). The Linux file is built on a release machine, most typically an Ubuntu machine with a fairly standard setup. If the existing distributed binaries do not work you can look to this project and fiddle with the appropriate linker and compiler options until it works.

Chapter 21. Optimizing Jetty

Table of Contents

Garbage Collection	241
High Load	242
Limiting Load	244

There are many ways to optimize Jetty, which vary depending on the situation. Are you trying to optimize for number of requests within a given amount of time? Are you trying to optimize the serving of static content? Do you have a large bit of hardware that you want to give entirely over to Jetty to use to its heart's delight? Here are a few of the many different ways to optimize Jetty.

Garbage Collection

Tuning the JVM garbage collection (GC) can greatly improve Jetty performance. Specifically, you can avoid pauses while the system performs full garbage collections. Optimal tuning of the GC depends on the behaviour of the application and requires detailed analysis, however there are general recommendations.

Tuning Examples

These options are general to the Sun JVM, and work in a JDK 6 installation. They provide good information about how your JVM is running; based on that initial information, you can then tune more finely.

To print the implicit flags with which the JVM is configured:

```
-XX:+PrintCommandLineFlags
```

To disable explicit GC performed regularly by RMI:

```
-XX:+DisableExplicitGC
```

To print the date and time stamps of GC activity with details:

```
-XX:+PrintGCDetails \
-XX:+PrintGCDateStamps \
-XX:+PrintGCTimeStamps \
-XX:+PrintTenuringDistribution
```

To log GC details to a file:

```
-Xloggc:[path/to/gc.log]
```

```
To print GC activity with less detail:
```

```
-verbose:gc
```

```
To use the concurrent marksweep GC with full GC at 80% old generation full:
```

```
-XX:+UseConcMarkSweepGC \
-XX:CMSInitiatingOccupancyFraction=80
```

High Load

Configuring Jetty for high load, whether for load testing or for production, requires that the operating system, the JVM, Jetty, the application, the network and the load generation all be tuned.

Load Generation for Load Testing

The load generation machines must have their OS, JVM, etc., tuned just as much as the server machines.

The load generation should not be over the local network on the server machine, as this has unrealistic performance and latency as well as different packet sizes and transport characteristics.

The load generator should generate a realistic load:

- A common mistake is that load generators often open relatively few connections that are extremely busy sending as many requests as possible over each connection. This causes the measured throughput to be limited by request latency (see [Lies, Damned Lies and Benchmarks](#) for an analysis of such an issue).
- Another common mistake is to use TCP/IP for a single request, and to open many, many short-lived connections. This often results in accept queues filling and limitations due to file descriptor and/or port starvation.
- A load generator should model the traffic profile from the normal clients of the server. For browsers, this is often between two and six connections that are mostly idle and that are used in sporadic bursts with read times in between. The connections are typically long held HTTP/1.1 connections.
- Load generators should be written in asynchronous programming style, so that a limited number of threads does not restrict the maximum number of users that can be simulated. If the generator is not asynchronous, a thread pool of 2000 may only be able to simulate 500 or fewer users. The Jetty HttpClient is an ideal choice for building a load generator, as it is asynchronous and can simulate many thousands of connections (see the Cometd Load Tester for a good example of a realistic load generator).

Operating System Tuning

Both the server machine and any load generating machines need to be tuned to support many TCP/IP connections and high throughput.

Linux

Linux does a reasonable job of self-configuring TCP/IP, but there are a few limits and defaults that you should increase. You can configure most of them in `/etc/security/limits.conf` or via `sysctl`.

TCP Buffer Sizes

You should increase TCP buffer sizes to at least 16MB for 10G paths and tune the autotuning (although you now need to consider buffer bloat).

```
$ sysctl -w net.core.rmem_max=16777216
$ sysctl -w net.core.wmem_max=16777216
$ sysctl -w net.ipv4.tcp_rmem="4096 87380 16777216"
$ sysctl -w net.ipv4.tcp_wmem="4096 16384 16777216"
```

Queue Sizes

`net.core.somaxconn` controls the size of the connection listening queue. The default value is 128; if you are running a high-volume server and connections are getting refused at a TCP level, you need to increase this. This is a very tweakable setting in such a case: if you set it too high, resource problems occur as it tries to notify a server of a large number of connections, and many remain pending, but if you set it too low, refused connections occur.

```
$ sysctl -w net.core.somaxconn=4096
```

The `net.core.netdev_max_backlog` controls the size of the incoming packet queue for upper-layer (java) processing. The default (2048) may be increased and other related parameters (TODO MORE EXPLANATION) adjusted with:

```
$ sysctl -w net.core.netdev_max_backlog=16384
$ sysctl -w net.ipv4.tcp_max_syn_backlog=8192
$ sysctl -w net.ipv4.tcp_syncookies=1
```

Ports

If many outgoing connections are made (for example, on load generators), the operating system might run low on ports. Thus it is best to increase the port range, and allow reuse of sockets in TIME_WAIT:

```
$ sysctl -w net.ipv4.ip_local_port_range="1024 65535"
$ sysctl -w net.ipv4.tcp_tw_recycle=1
```

File Descriptors

Busy servers and load generators may run out of file descriptors as the system defaults are normally low. These can be increased for a specific user in `/etc/security/limits.conf`:

```
theusername      hard  nofile    40000
theusername      soft   nofile    40000
```

Congestion Control

Linux supports pluggable congestion control algorithms. To get a list of congestion control algorithms that are available in your kernel run:

```
$ sysctl net.ipv4.tcp_available_congestion_control
```

If cubic and/or htcp are not listed, you need to research the control algorithms for your kernel. You can try setting the control to cubic with:

```
$ sysctl -w net.ipv4.tcp_congestion_control=cubic
```

Mac OS

TBD

Windows

TBD

Network Tuning

Intermediaries such as nginx can use a non-persistent HTTP/1.0 connection. Make sure to use persistent HTTP/1.1 connections.

JVM Tuning

- Tune the [Garbage Collection](#)
- Allocate sufficient memory
- Use the -server option
- Jetty Tuning

Connectors

Acceptors

The standard rule of thumb for the number of Acceptors to configure is one per CPU on a given machine.

Low Resource Limits

Must not be configured for less than the number of expected connections.

Thread Pool

Configure with goal of limiting memory usage maximum available. Typically >50 and <500

Limiting Load

To achieve optimal fair handling for all users of a server, it can be necessary to limit the resources that each user/connection can utilize so as to maximize throughput for the server or to ensure that the entire server runs within the limitations of its runtime

Low Resources Monitor

An instance of [LowResourcesMonitor](#) may be added to a Jetty Server to monitor for low resources situations and to take action to limit the number of idle connections on the server. To configure the low resources monitor, you can uncomment the jetty-lowresources.xml line from the start.ini configuration file, which has the effect of including the following XML configuration:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<!-- ===== -->
<!-- Mixin the Low Resources Monitor -->
<!-- ===== -->

<Configure id="Server" class="org.eclipse.jetty.server.Server">
    <Call name="addBean">
        <Arg>
            <New class="org.eclipse.jetty.server.LowResourceMonitor">
                <Arg name="server"><Ref refid='Server' /></Arg>
                <Set name="period"><Property name="lowresources.period" default="1000"/></Set>
                <Set name="lowResourcesIdleTimeout"><Property
name="lowresources.lowResourcesIdleTimeout" default="200"/></Set>
                <Set name="monitorThreads"><Property name="lowresources.monitorThreads"
default="true"/></Set>
                <Set name="maxConnections"><Property name="lowresources.maxConnections"
default="0"/></Set>
                <Set name="maxMemory"><Property name="lowresources.maxMemory" default="0"/></Set>
                <Set name="maxLowResourcesTime"><Property name="lowresources.maxLowResourcesTime"
default="5000"/></Set>
            </New>
        </Arg>
    </Call>
</Configure>
```

The monitor is configured with a period in milliseconds at which it will scan the server looking for a low resources condition, which may be one of:

- If monitorThreads is configured as true and a connectors Executor is an instance of [ThreadPool](#), then its isLowOnThreads() method is used to detect low resources.
- If maxConnections is configured to a number >0 then if the total number of connections from all monitored connectors exceeds this value, then low resources state is entered.
- If the maxMemory field is configured to a number of bytes >0 then if the JVMs total memory minus its idle memory exceeds this value, then low resources state is entered.

Once low resources state is detected, then the monitor will iterate over all existing connections and set their IdleTimeout to its configured lowResourcesIdleTimeout in milliseconds. This allows the idle time of existing connections to be reduced so that the connection is quickly closed if no further request are received.

If the low resources state persists longer than the time in milliseconds configured for the maxLowResourcesTime field, the lowResourcesIdleTimeout is repeatedly applied so that new connections as well as existing connections will be limited.

Denial Of Service Filter

TBD (see [DoSFilter](#)).

Chapter 22. Jetty Logging

Table of Contents

Configuring Jetty Logging	246
Default Logging with Jetty's StdErrLog	247
Configuring Jetty Request Logs	249
Example: Logging with Apache Log4j	251
Example: Logging with Java's java.util.logging via Slf4j	252
Example: Logging with Java's java.util.logging via JavaUtilLog	254
Example: Logging with Logback	255
Example: Capturing Multiple Logging Frameworks with Slf4j	256
Example: Centralized Logging with Logback	259
Jetty Dump Tool	262

This chapter discusses various options for configuring logging.

Configuring Jetty Logging

Jetty provides logging via its own `org.eclipse.jetty.util.log.Logger` layer, and does not natively use any existing Java logging framework. All logging events, produced via the Jetty logging layer, have a name, a level, and a message. The name is a FQCN (fully qualified class name) similar to how all existing Java logging frameworks operate.

Jetty logging, however, has a slightly different set of levels that it uses internally:

WARN

For events serious enough to inform and log, but not fatal.

INFO

Informational events.

DEBUG

Debugging events (very noisy).

IGNORE

Exception events that you can safely ignore, but useful for some people. You might see this level as DEBUG under some Java logging framework configurations, where it retains the *ignore* phrase somewhere in the logging.



Note

Jetty logging produces no FATAL or SEVERE events.

Selecting the Log Framework

Configure the Jetty logging layer via the `org.eclipse.jetty.util.log.Log` class, following [these rules](#).

1. Load Properties

- First from a Classpath Resource called `jetty-logging.properties` (if found).
- Then from the `System.getProperties()`.

2. Determine the log implementation.

- If property `org.eclipse.jetty.util.log.class` is defined, load the class it defines as the logger implementation from the server classpath.
- If the class `org.slf4j.Logger` exists in server classpath, the Jetty implementation becomes `org.eclipse.jetty.util.log.Slf4jLog`.
- If no logger implementation is specified, default to `org.eclipse.jetty.util.log.StdErrLog`.



Note

You can create your own custom logging by providing an implementation of the [Jetty Logger API](#). For an example of a custom logger, see [JavaUtilLog.java](#).

The jetty-logging.properties file

By default, the internal Jetty Logging discovery mechanism will load logging specific properties from a classpath resource called `jetty-logging.properties` and then initialize the Logging from a combination of properties found in that file, along with any System Properties.

A typical `jetty-logging.properties` file will include at least the declaration of which logging implementation you want to use by defining a value for the `org.eclipse.jetty.util.log.class` property.

Examples for various logging frameworks can be found later in this documentation.

- Default Logging with Jetty's [StdErrLog](#)
- Using [Log4j](#) via [Slf4jLog](#)
- Using [Logback](#) via [Slf4jLog](#)
- Using [java.util.logging](#) via [Slf4jLog](#)
- Using [java.util.logging](#) via Jetty's [JavaUtilLog](#)
- Capturing [Multiple Logging Frameworks](#) via [Slf4jLog](#)
- [Centralized Logging with Logback and Slf4jLog](#)

Default Logging with Jetty's StdErrLog

If you do nothing to configure your own logging framework, then Jetty will default to using an internal `org.eclipse.jetty.util.log.StdErrLog` implementation. This will output all logging events to STDERR (aka `System.err`).

Simply use Jetty and you get `StdErrLog` based logging output.

Capturing Console Output to File

Included in the Jetty distribution is a logging module that is capable of performing simple capturing of all `STDOUT` and `STDERR` output to a file that is rotated daily.

To enable on the command line:

```
[my-base]$ java -jar /opt/jetty/start.jar --module=logging
```

You can also include the `--module=logging` command in your `/${jetty.base}/start.ini`

```
[my-base]$ java -jar /opt/jetty/start.jar --add-to-start=logging
```

The default configuration for logging output will create a file \${jetty.logs}/yyyy_mm_dd.stderrout.log which allows you to configure the output directory by setting the jetty.logs property to a path of your choice.

If you want a more advanced configuration of your logging output, consider using your logging library of choice.

Configuring StdErrLog

The recommended way to configure StdErrLog is to create a \${jetty.home}/resources/jetty-logging.properties file, specify the Log implementation to StdErrLog and then setup your logging levels.

```
# Configure Jetty for StdErrLog Logging
org.eclipse.jetty.util.log.class=org.eclipse.jetty.util.log.StrErrLog
# Overall Logging Level is INFO
org.eclipse.jetty.LEVEL=INFO
# Detail Logging for WebSocket
org.eclipse.jetty.websocket.LEVEL=DEBUG
```

There are a number of properties you can define in here that will affect the StdErrLog behavior.

<name>.LEVEL=<level>

Sets the logging level for all loggers within the name specified to the level, which can be (in increasing order of restriction) ALL, DEBUG, INFO, WARN, OFF. The name (or hierarchy) can be a specific fully qualified class or a package namespace, for example, org.eclipse.jetty.http.LEVEL=DEBUG is a package namespace approach to turn all loggers in the Jetty HTTP package to DEBUG level, and org.eclipse.jetty.io.ChanelEndPoint.LEVEL=ALL turns on all logging events for the specific class, including DEBUG, INFO, WARN (and even special internally ignored exception classes). If more than one system property specifies a logging level, the most specific one applies.

<name>.SOURCE=<boolean>

Named Logger specific, attempts to print the Java source file name and line number from where the logging event originated. Name must be a fully qualified class name (this configurable does not support package name hierarchy). Default is false. Be aware that this is a slow operation and has an impact on performance.

<name>.STACKS=<boolean>

Named Logger specific, controls the display of stacktraces. Name must be a fully qualified class name (this configurable does not support package name hierarchy).

Default is true.

org.eclipse.jetty.util.log.stderr.SOURCE=<boolean>

Special Global Configuration. Attempts to print the Java source file name and line number from where the logging event originated. Default is false.

org.eclipse.jetty.util.log.stderr.LONG=<boolean>

Special Global Configuration. When true, outputs logging events to STDERR using long form, fully qualified class names. When false, uses abbreviated package names.

Default is false.

Example when set to false:

```
2014-06-03 14:36:16.013:INFO:oejs.Server:main: jetty-9.2.0.v20140526
2014-06-03 14:36:16.028:INFO:oejdp.ScanningAppProvider:main: Deployment monitor
[file:/opt/jetty/demo-base/webapps/] at interval 1
2014-06-03 14:36:16.051:INFO:oejsh.ContextHandler:main: Started
o.e.j.s.h.MovedContextHandler@7d256e50{/oldContextPath,null,AVAILABLE}
```

```
2014-06-03 14:36:17.880:INFO:oejs.ServerConnector:main: Started  
ServerConnector@34f2d11a{HTTP/1.1}{0.0.0.0:8080}  
2014-06-03 14:36:17.888:INFO:oejs.Server:main: Started @257ms
```

Example when set to true:

```
2014-06-03 14:38:19.019:INFO:org.eclipse.jetty.server.Server:main:  
jetty-9.2.0.v20140526  
2014-06-03  
14:38:19.032:INFO:org.eclipse.jetty.deploy.providers.ScanningAppProvider:main:  
Deployment monitor [file:/opt/jetty/demo-base/webapps/] at interval 1  
2014-06-03 14:38:19.054:INFO:org.eclipse.jetty.server.handler.ContextHandler:main:  
Started o.e.j.s.h.MovedContextHandler@246d8660{/oldContextPath,null,AVAILABLE}  
2014-06-03 14:38:20.715:INFO:org.eclipse.jetty.server.ServerConnector:main: Started  
ServerConnector@59f625be{HTTP/1.1}{0.0.0.0:8080}  
2014-06-03 14:38:20.723:INFO:org.eclipse.jetty.server.Server:main: Started @243ms
```

Deprecated Parameters:

These parameters existed in prior versions of Jetty, and are no longer supported. They are included here for historical (and search engine) reasons.

org.eclipse.jetty.util.log.DEBUG

Formerly used to enable DEBUG level logging on any logger used within Jetty (not just Jetty's own logger).

Replaced with using the logger implementation specific configuration and level filtering.

org.eclipse.jetty.util.log.stderr.DEBUG

Formerly used to enable DEBUG level logging on the internal Jetty StdErrLog implementation.

Replaced with level specific equivalent: example: org.eclipse.jetty.LEVEL=DEBUG

DEBUG

Ancient debugging flag that turned on all debugging, even non-logging debugging.

Jetty no longer uses because many third party libraries employ this overly simple property name, which would generate far too much console output.

Configuring Jetty Request Logs

Request logs are a record of the requests that the server has processed. There is one entry per request received, and commonly in the standard NCSA format, so you can use tools like [Webalizer](#) to analyze them conveniently.

Constructing a Request Log Entry

A standard request log entry includes the client IP address, date, method, URL, result, size, referrer, and user agent, for example:

```
123.4.5.6 - - [27/Aug/2004:10:16:17 +0000]  
"GET /jetty/tut/XmlConfiguration.html HTTP/1.1"  
200 76793 "http://localhost:8080/jetty/tut/logging.html"  
"Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.6) Gecko/20040614 Firefox/0.8"
```

Implementing a Request Log

Jetty provides an implementation called *NCSAResponseLog* which supports the NCSA format in files that you can roll over on a daily basis.

The [Logback Project](#) offers [another implementation](#) of a RequestLog interface, providing rich and powerful HTTP-access log functionality.

If neither of these options meets your needs, you can implement a custom request logger by implementing Jetty's [RequestLog.java](#) interface and plugging it in similar to the NCSARequestLog, as shown below.

Configuring a Request Log

To configure a single request log for the entire Jetty Server instance:

```
<Set name="handler">
  <New id="Handlers" class="org.eclipse.jetty.server.handler.HandlerCollection">
    <Set name="handlers">
      <Array type="org.eclipse.jetty.server.Handler">
        <Item>

          <New id="Contexts" class="org.eclipse.jetty.server.handler.ContextHandlerCollection"/>
            </Item>
            <Item>

              <New id="DefaultHandler" class="org.eclipse.jetty.server.handler.DefaultHandler"/>
                </Item>
                <Item>

                  <New id="RequestLog" class="org.eclipse.jetty.server.handler.RequestLogHandler"/>
                    </Item>
                    </Array>
                  </Set>
                </New>
              </Set>
            <Ref refid="RequestLog">
              <Set name="requestLog">
                <New id="RequestLogImpl" class="org.eclipse.jetty.server.NCSARequestLog">
                  <Arg><SystemProperty name="jetty.logs" default=".logs"/>/yyyy_mm_dd.request.log</Arg>
                  <Set name="retainDays">90</Set>
                  <Set name="append">true</Set>
                  <Set name="extended">false</Set>
                  <Set name="LogTimeZone">GMT</Set>
                </New>
              </Set>
            </Ref>
```

The equivalent code is:

```
HandlerCollection handlers = new HandlerCollection();
ContextHandlerCollection contexts = new ContextHandlerCollection();
RequestLogHandler requestLogHandler = new RequestLogHandler();
handlers.setHandlers(new Handler[]{contexts,new DefaultHandler(),requestLogHandler});
server.setHandler(handlers);

NCSARequestLog requestLog = new NCSARequestLog("./logs/jetty-yyyy_mm_dd.request.log");
requestLog.setRetainDays(90);
requestLog.setAppend(true);
requestLog.setExtended(false);
requestLog.setLogTimeZone("GMT");
requestLogHandler.setRequestLog(requestLog);
```

This configures a request log in \$JETTY_HOME/logs with filenames including the date. Old log files are kept for 90 days before being deleted. Existing log files are appended to and the extended NCSA format is used in the GMT timezone.

To examine many more configuration options, see [NCSARequestLog.java](#)

Configuring a Separate Request Log For a Web Application

To configure a separate request log for a web application, add the following to the context XML file.

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  ...
```

```

<Set name="handler">
  <New id="RequestLog" class="org.eclipse.jetty.server.handler.RequestLogHandler">
    <Set name="requestLog">
      <New id="RequestLogImpl" class="org.eclipse.jetty.server.NCSARequestLog">
        <Set name="filename"><Property name="jetty.logs" default=".logs"/>/test-
yyyy_mm_dd.request.log</Arg>
        <Set name="filenameDateFormat">yyyy_MM_dd</Set>
        <Set name="LogTimeZone">GMT</Set>
        <Set name="retainDays">90</Set>
        <Set name="append">true</Set>
      </New>
    </Set>
  </New>
</Set>
...
</Configure>

```

Example: Logging with Apache Log4j

It is possible to have the Jetty Server logging configured so that Log4j controls the output of logging events produced by Jetty. This is accomplished by configuring Jetty for logging to [Apache Log4j](#) via [Slf4j](#) and the [Slf4j binding layer for Log4j](#).

Quick Setup of Log4j Logging using Jetty 9.2.1+

A convenient replacement logging module has been created to bootstrap your \${jetty.base} directory for logging with log4j.

```

[mybase]$ mkdir modules
[mybase]$ cd modules
[modules]$ curl -O https://raw.githubusercontent.com/jetty-project/logging-modules/
master/log4j-1.2/logging.mod
% Total    % Received % Xferd  Average Speed   Time     Time     Current
          Dload  Upload Total Spent   Left Speed
100  720  100  720    0     0  2188      0 ---:--- ---:--- 2188
[modules]$ cd ..
[mybase]$ java -jar /opt/jetty-dist/start.jar --add-to-start=logging
INFO: logging initialised in ${jetty.base}/start.ini (appended)
MKDIR: ${jetty.base}/logs
DOWNLOAD: http://central.maven.org/maven2/org/slf4j/slf4j-api/1.6.6/slf4j-api-1.6.6.jar
to lib/logging/slf4j-api-1.6.6.jar
DOWNLOAD: http://central.maven.org/maven2/org/slf4j/slf4j-log4j12/1.6.6/slf4j-
log4j12-1.6.6.jar to lib/logging/slf4j-log4j12-1.6.6.jar
DOWNLOAD: http://central.maven.org/maven2/log4j/log4j/1.2.17/log4j-1.2.17.jar to lib/
logging/log4j-1.2.17.jar
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/
log4j-1.2/log4j.properties to resources/log4j.properties
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/
log4j-1.2/jetty-logging.properties to resources/jetty-logging.properties
INFO: resources initialised transitively
INFO: resources enabled in ${jetty.base}/start.ini
[mybase]$ java -jar /opt/jetty-dist/start.jar

```

That's Cool! But what just happened?

The replacement logging.mod performs a number of tasks.

1. mybase is a \${jetty.base} directory
2. The jetty-distribution is unpacked (and untouched) into /opt/jetty-dist/ and becomes the \${jetty.home} directory for this demonstration.
3. The curl command downloads the replacement logging.mod and puts it into the \${jetty.base}/modules/ directory for use by mybase only.
4. The start.jar --add-to-start=logging command performs a number of steps to make the logging module available to the \${jetty.base} configuration.

- a. The **--module=logging** command is added to the `${jetty.base}/start.ini` configuration
 - b. Required `${jetty.base}` directories are created: `${jetty.base}/logs` and `${jetty.base}/resources`
 - c. Required libraries are downloaded (if not present already): slf4j-api, slf4j-log4j, and log4j itself.
The libraries are put in the `${jetty.base}/lib/logging/` directory.
 - d. Required configuration files are downloaded (if not present already): jetty-logging.properties, and log4j.properties
The configuration files are put in the `${jetty.base}/resources/` directory.
5. At this point you have your mybase configured so that the jetty server itself will log using log4j, using the log4j configuration found in `mybase/resources/log4j.properties`

You can verify the server classpath by using the **start.jar --list-config** command.

In essence, Jetty is now configured to emit its own logging events to slf4j, and slf4j itself is using the static log binder found in `slf4j-log4j12.jar`. Making all Jetty + Slf4j + Log4j events emitted by the Jetty server go to Log4j for routing (to console, file, syslog, etc...)

Example: Logging with Java's java.util.logging via Slf4j

It is possible to have the Jetty Server logging configured so that [java.util.logging](#) controls the output of logging events produced by Jetty.

This example demonstrates how to configuring Jetty for logging to [java.util.logging](#) via [Slf4j](#) and the [Slf4j binding layer for java.util.logging](#). If you want to use the built-in native JavaUtilLog implementation, see [example-logging-java-util-logging-native](#)

Quick Setup of java.util.logging (Slf4j) Logging using Jetty 9.2.1+

A convenient replacement logging module has been created to bootstrap your `${jetty.base}` directory for logging with `java.util.logging`.

```
[mybase]$ mkdir modules
[mybase]$ cd modules
[modules]$ curl -O https://raw.githubusercontent.com/jetty-project/logging-modules/master/java.util.logging-slf4j/logging.mod
% Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload Total   Spent   Left Speed
100  826  100  826    0     0  2468      0  --::-- --::-- 2473
[modules]$ cd ..
[mybase]$ java -jar /opt/jetty-dist/start.jar --add-to-start=logging
INFO: logging initialised in ${jetty.base}/start.ini (appended)
MKDIR: ${jetty.base}/logs
DOWNLOAD: http://central.maven.org/maven2/org/slf4j/slf4j-jdk14/1.6.6/slf4j-jdk14-1.6.6.jar to lib/logging/slf4j-jdk14-1.6.6.jar
DOWNLOAD: http://central.maven.org/maven2/org/slf4j/slf4j-api/1.6.6/slf4j-api-1.6.6.jar to lib/logging/slf4j-api-1.6.6.jar
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/java.util.logging-slf4j/jetty-logging.xml to etc/jetty-logging.xml
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/java.util.logging-slf4j/logging.properties to resources/logging.properties
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/java.util.logging-slf4j/jetty-logging.properties to resources/jetty-logging.properties
```

```
INFO: resources      initialised transitively
INFO: resources      enabled in      ${jetty.base}/start.ini
[mybase]$ java -jar /opt/jetty-dist/start.jar
```

That's Cool! But what just happened?

The replacement logging.mod performs a number of tasks.

1. mybase is a \${jetty.base} directory
2. The jetty-distribution is unpacked (and untouched) into /opt/jetty-dist/ and becomes the \${jetty.home} directory for this demonstration.
3. The curl command downloads the replacement logging.mod and puts it into the \${jetty.base}/modules/ directory for use by mybase only.
4. The start.jar --add-to-start=logging command performs a number of steps to make the logging module available to the \${jetty.base} configuration.
 - a. The --module=logging command is added to the \${jetty.base}/start.ini configuration
 - b. Required \${jetty.base} directories are created: \${jetty.base}/logs and \${jetty.base}/resources
 - c. Required libraries are downloaded (if not present already): slf4j-api, and slf4j-jdk14.

The libraries are put in the \${jetty.base}/lib/logging/ directory.

- d. Required configuration files are downloaded (if not present already): jetty-logging.properties, and logging.properties

The configuration files are put in the \${jetty.base}/resources/ directory.

- e. Required java.util.logging initialization commands are downloaded (if not present already): jetty-logging.xml

The xml file is put in the \${jetty.base}/etc/ directory.

5. At this point you have your mybase configured so that the jetty server itself will log using java.util.logging, using the java.util.logging configuration found in mybase/resources/logging.properties

You can verify the server classpath by using the start.jar --list-config command.

In essence, Jetty is now configured to emit its own logging events to slf4j, and slf4j itself is using the static log binder found in slf4j-jdk14.jar. Making all Jetty + Slf4j + java.util.logging events emitted by the Jetty server go to java.util.logging for routing (to console, file, etc...)

If you have any custom java.util.logging handlers that you want to use, put the implementation jar in your \${jetty.base}/lib/logging/ directory and reference them in the \${jetty.base}/resources/logging.properties file.



Note

java.util.logging is configured via the \${jetty.base}/resources/logging.properties file during a valid startup of Jetty. This means that if there is any startup errors that occur before java.util.logging is configured, they will likely be lost and/or not routed through your configuration. (Other logging frameworks are more reliable in that they always initialize and configure on first use, unlike java.util.logging)

While it is possible to configure `java.util.logging` sooner, even at JVM startup, the example demonstrated here does not show this technique. For more information consult the [official `java.util.logging` `LogManager` javadoc documentation from Oracle](#).

Example: Logging with Java's `java.util.logging` via `JavaUtilLog`

It is possible to have the Jetty Server logging configured so that `java.util.logging` controls the output of logging events produced by Jetty.

This example demonstrates how to configuring Jetty for logging to `java.util.logging` via Jetty's own `JavaUtilLog` implementation.

Important



While this is a valid setup, the Jetty project recommends always using the [slf4j to `java.util.logging` configuration](#) for memory and performance reasons. (this naive implementation is very non-performant and is not guaranteed to exist in the future)

Quick Setup of `java.util.logging` (native) Logging using Jetty 9.2.1+

A convenient replacement logging module has been created to bootstrap your `${jetty.base}` directory for logging with `java.util.logging`.

```
[mybase]$ mkdir modules
[mybase]$ cd modules
[modules]$ curl -O https://raw.githubusercontent.com/jetty-project/logging-modules/master/java.util.logging-native/logging.mod
% Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload Total   Spent   Left  Speed
100  623  100  623    0     0  1879      0 --:--:-- --:--:-- --:--:-- 1876
[modules]$ cd ..
[mybase]$ java -jar /opt/jetty-dist/start.jar --add-to-start=logging
INFO: logging initialised in ${jetty.base}/start.ini (appended)
MKDIR: ${jetty.base}/logs
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/java.util.logging-native/jetty-logging.xml to etc/jetty-logging.xml
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/java.util.logging-native/logging.properties to resources/logging.properties
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/java.util.logging-native/jetty-logging.properties to resources/jetty-logging.properties
INFO: resources initialised transitively
INFO: resources enabled in ${jetty.base}/${jetty.base}
[mybase]$ java -jar /opt/jetty-dist/start.jar
```

That's Cool! But what just happened?

The replacement `logging.mod` performs a number of tasks.

1. `mybase` is a `${jetty.base}` directory
2. The jetty-distribution is unpacked (and untouched) into `/opt/jetty-dist/` and becomes the `${jetty.home}` directory for this demonstration.
3. The `curl` command downloads the replacement `logging.mod` and puts it into the `${jetty.base}/modules/` directory for use by `mybase` only.
4. The `start.jar --add-to-start=logging` command performs a number of steps to make the logging module available to the `${jetty.base}` configuration.

a. The **--module=logging** command is added to the \${jetty.base}/start.ini configuration

b. Required \${jetty.base} directories are created: \${jetty.base}/logs and \${jetty.base}/resources

c. Required configuration files are downloaded (if not present already): jetty-logging.properties, and logging.properties

The configuration files are put in the \${jetty.base}/resources/ directory.

d. Required java.util.logging initialization commands are downloaded (if not present already): jetty-logging.xml

The xml file is put in the \${jetty.base}/etc/ directory.

5. At this point you have your mybase configured so that the jetty server itself will log using java.util.logging, using the java.util.logging configuration found in mybase/resources/logging.properties

You can verify the server classpath by using the **start.jar --list-config** command.

In essence, Jetty is now configured to use org.eclipse.jetty.util.log.JavaUtilLog, which emit its own logging events to java.util.logging. Making all Jetty + java.util.logging events emitted by the Jetty server go to java.util.logging for routing (to console, file, etc...)

If you have any custom java.util.logging handlers that you want to use, put the implementation jar in your \${jetty.base}/lib/logging/ directory and reference them in the \${jetty.base}/resources/logging.properties file.



Note

java.util.logging is configured via the \${jetty.base}/resources/logging.properties file during a valid startup of Jetty. This means that if there is any startup errors that occur before java.util.logging is configured, they will likely be lost and/or not routed through your configuration. (Other logging frameworks are more reliable in that they always initialize and configure on first use, unlike java.util.logging)

While it is possible to configure java.util.logging sooner, even at JVM startup, the example demonstrated here does not show this technique. For more information consult the [official java.util.logging LogManager javadoc documentation from Oracle](#).

Example: Logging with Logback

It is possible to have the Jetty Server logging configured so that Logback controls the output of logging events produced by Jetty. This is accomplished by configuring Jetty for logging to [Logback](#), which uses [Slf4j](#) and the [Logback Implementation for Slf4j](#).

Quick Setup of Logback Logging using Jetty 9.2.1+

A convenient replacement logging module has been created to bootstrap your \${jetty.base} directory for logging with logback.

```
[mybase]$ mkdir modules
[mybase]$ cd modules
[modules]$ curl -O https://raw.githubusercontent.com/jetty-project/logging-modules/master/logback/logging.mod
% Total    % Received % Xferd  Average Speed   Time     Time     Time Current

```

```
Dload Upload Total Spent Left Speed
100 742 100 742 0 0 2196 0 ---:--- ---:--- ---:--- 2201
[mybase]$ cd ..
[mybase]$ java -jar /opt/jetty-dist/start.jar --add-to-start=logging
INFO: logging initialised in ${jetty.base}/start.ini (appended)
mkdir: ${jetty.base}/logs
DOWNLOAD: http://central.maven.org/maven2/org/slf4j/slf4j-api/1.6.6/slf4j-api-1.6.6.jar
to lib/logging/slf4j-api-1.6.6.jar
DOWNLOAD: http://central.maven.org/maven2/ch/qos/logback/logback-core/1.0.7/logback-
core-1.0.7.jar to lib/logging/logback-core-1.0.7.jar
DOWNLOAD: http://central.maven.org/maven2/ch/qos/logback/logback-classic/1.0.7/logback-
classic-1.0.7.jar to lib/logging/logback-classic-1.0.7.jar
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/logback/
logback.xml to resources/logback.xml
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/logback/
jetty-logging.properties to resources/jetty-logging.properties
[mybase]$ java -jar /opt/jetty-dist/start.jar
```

That's Cool! But what just happened?

The replacement `logging.mod` performs a number of tasks.

1. `mybase` is a `${jetty.base}` directory
2. The jetty-distribution is unpacked (and untouched) into `/opt/jetty-dist/` and becomes the `${jetty.home}` directory for this demonstration.
3. The `curl` command downloads the replacement `logging.mod` and puts it into the `${jetty.base}/modules/` directory for use by `mybase` only.
4. The `start.jar --add-to-start=logging` command performs a number of steps to make the logging module available to the `${jetty.base}` configuration.
 - a. The `--module=logging` command is added to the `${jetty.base}/start.ini` configuration
 - b. Required `${jetty.base}` directories are created: `${jetty.base}/logs` and `${jetty.base}/resources`
 - c. Required libraries are downloaded (if not present already): `slf4j-api`, `logback-core`, and `logback-classic`.

The libraries are put in the `${jetty.base}/lib/logging/` directory.

- d. Required configuration files are downloaded (if not present already): `jetty-logging.properties`, and `logback.xml`

The configuration files are put in the `${jetty.base}/resources/` directory.

5. At this point you have your `mybase` configured so that the jetty server itself will log using logback, using the logback configuration found in `mybase/resources/logback.xml`

You can verify the server classpath by using the `start.jar --list-config` command.

In essence, Jetty is now configured to emit its own logging events to slf4j, and slf4j itself is using the static log binder found in `logback-classic.jar`. Making all Jetty + Slf4j + Logback events emitted by the Jetty server go to Logback for routing (to console, file, syslog, etc...)

Example: Capturing Multiple Logging Frameworks with Slf4j

This page describes how to configure Jetty for capturing multiple logging frameworks logging events into a single logging implementation handled by Slf4j.

When using Slf4j, you can configure a single logging solution for the variety of logging libraries available in common use. With careful setup, you can support all of the following logging APIs at the same time, with a single configuration file to control the output of events produced by these APIs.

Logging APIs that Slf4j supports:

- Slf4j API
- Logback API
- Apache Log4j 1.2
- JDK 1.4 Logging (aka `java.util.logging`)
- Apache Commons Logging

To accomplish this you must make some careful choices, starting with a single underlying logging framework. This decision guides the rest of your choices about JARs to place on the Server classpath.

Table 22.1. Slf4j Logging Grid

Logging API	Slf4j Binding Jar	Slf4j Adapter Jar	Underlying Logging Framework
Logback API	n/a	logback-classic.jar	logback-core.jar
Log4j	log4j-over-slf4j.jar	slf4j-log4j12.jar	log4j.jar
JDK 1.4 Logging	jul-to-slf4j.jar	slf4j-jdk14.jar	(Core Java Classlib)
Commons Logging	jcl-over-slf4j.jar	slf4j-jcl.jar	commons-logging.jar

Logging API

The Logging API that you are either capturing events from and/or using to write out those events (for example, to disk).

Slf4j Binding JAR

Special JARs, created and maintained by the Slf4j project, that pretend to be the various Logging API implementation classes, but instead just route that Logging API's events to Slf4j to handle.

There MAY be multiple Slf4j binding JARs present on the classpath at the same time.

For a single logging API, if you choose to use the Slf4j binding JAR, then you MUST NOT include the SLF4j adapter JAR or underlying logging framework in the classpath as well.

Slf4j Adapter Jar

These JARs are created and maintained by the Slf4j project and route Slf4j logging events to a specific underlying logging framework.

There MUST NOT be multiple Slf4j adapter JARs present on the classpath at the same time.

Logging events that these adapter JARs capture can come from direct use of the Slf4j API or via one of the Slf4j binding JAR implementations.

Underlying Logging Framework

This is the last leg of your configuration, the implementation that processes, filters, and outputs the logging events to the console, logging directory on disk, or whatever else the underlying logging framework supports (like Socket, SMTP, Database, or even SysLog in the case of Logback).

Caution



There MUST NOT be multiple underlying logging frameworks on the classpath. If there are, the Slf4j framework fails to load.



Note

Some third party libraries provide their own implementations of common logging APIs; be careful not to accidentally include an underlying logging framework.

For example, if you are using SpringSource you likely have a `com.springsource.org.apache.log4j.jar` along with a `log4j.jar`, which have the same classes in them. In this example, use the `com.springsource.org.apache.log4j.jar` version and exclude the `log4j.jar`, as the SpringSource version includes extra metadata suitable for using SpringSource.

The following sections use Logback as the underlying Logging framework. This requires using `logback-classic.jar` and `logback-core.jar`, and excluding any other Slf4j adapter JAR or underlying logging framework.

It also requires including the other Slf4j binding JARs in the classpath, along with some special initialization for `java.util.logging`.

Quick Setup of Multiple Logging capture to Logback using Jetty 9.2.1+

A convenient replacement logging module has been created to bootstrap your `${jetty.base}` directory for capturing all Jetty server logging from multiple logging frameworks into a single logging output file managed by logback.

```
[mybase]$ mkdir modules
[mybase]$ cd modules
[modules]$ curl -O https://raw.githubusercontent.com/jetty-project/logging-modules/master/capture-all/logging.mod
% Total    % Received % Xferd  Average Speed   Time     Time      Time  Current
          Dload  Upload   Total Spent  Left Speed
100 1293  100 1293    0     0  3693      0 --:--:-- --:--:-- --:--:-- 3694
[modules]$ cd ..
[mybase]$ java -jar /opt/jetty-dist/start.jar --add-to-start=logging
INFO: logging initialised in ${jetty.base}/start.ini (appended)
MKDIR: ${jetty.base}/logs
DOWNLOAD: http://central.maven.org/maven2/org/slf4j/slf4j-api/1.6.6/slf4j-api-1.6.6.jar
to lib/logging/slf4j-api-1.6.6.jar
DOWNLOAD: http://central.maven.org/maven2/org/slf4j/log4j-over-slf4j/1.6.6/log4j-over-
slf4j-1.6.6.jar to lib/logging/log4j-over-slf4j-1.6.6.jar
DOWNLOAD: http://central.maven.org/maven2/org/slf4j/jul-to-slf4j/1.6.6/jul-to-
slf4j-1.6.6.jar to lib/logging/jul-to-slf4j-1.6.6.jar
DOWNLOAD: http://central.maven.org/maven2/org/slf4j/jcl-over-slf4j/1.6.6/jcl-over-
slf4j-1.6.6.jar to lib/logging/jcl-over-slf4j-1.6.6.jar
DOWNLOAD: http://central.maven.org/maven2/ch/qos/logback/logback-core/1.0.7/logback-
core-1.0.7.jar to lib/logging/logback-core-1.0.7.jar
DOWNLOAD: http://central.maven.org/maven2/ch/qos/logback/logback-classic/1.0.7/logback-
classic-1.0.7.jar to lib/logging/logback-classic-1.0.7.jar
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/capture-
all/logback.xml to resources/logback.xml
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/capture-
all/jetty-logging.properties to resources/jetty-logging.properties
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/capture-
all/jetty-logging.xml to etc/jetty-logging.xml
INFO: resources initialised transitively
INFO: resources enabled in ${jetty.base}/start.ini
[mybase]$ java -jar /opt/jetty-dist/start.jar
```

That's cool and all, but tell me what that just did.

The replacement `logging.mod` performs a number of tasks.

1. `mybase` is a `${jetty.base}` directory

2. The jetty-distribution is unpacked (and untouched) into `/opt/jetty-dist/` and becomes the `${jetty.home}` directory for this demonstration.
3. The `curl` command downloads the replacement `logging.mod` and puts it into the `${jetty.base}/modules/` directory for use by mybase only.
4. The `start.jar --add-to-start=logging` command performs a number of steps to make the logging module available to the `${jetty.base}` configuration.
 - a. The `--module=logging` command is added to the `${jetty.base}/start.ini` configuration
 - b. Required `${jetty.base}` directories are created: `${jetty.base}/logs` and `${jetty.base}/resources`
 - c. Required libraries are downloaded (if not present already):
 - `slf4j-api.jar` - API jar for Slf4j (used by most of the rest of the jars)
 - `log4j-over-slf4j.jar` - Slf4j jar that captures all log4j emitted logging events
 - `jul-to-slf4j.jar` - Slf4j jar that captures all `java.util.logging` events
 - `jcl-over-slf4j.jar` - Slf4j jar that captures all commons-logging events
 - `logback-classic.jar` - the Slf4j adapter jar that routes all of the captured logging events to logback itself.
 - `logback-core.jar` - the logback implementation jar, that handles all of the filtering and output of the logging events.

These libraries are put in the `${jetty.base}/lib/logging/` directory.

- d. Required configuration files are downloaded (if not present already): `jetty-logging.properties`, and `logback.xml`

The configuration files are put in the `${jetty.base}/resources/` directory.

- e. Required `java.util.logging` initialization commands are downloaded (if not present already): `jetty-logging.xml`

The xml file is put in the `${jetty.base}/etc/` directory.

5. At this point you have your mybase configured so that the jetty server itself will log using slf4j, and all other logging events from other Jetty Server components (such as database drivers, security layers, jsp, mail, and other 3rd party server components) are routed to logback for filtering and output.

You can verify the server classpath by using the `start.jar --list-config` command.

In essence, Jetty is now configured to emit its own logging events to slf4j, and various slf4j bridge jars are acting on behalf of log4j, java.util.logging, and commons-logging, routing all of the logging events to logback (a slf4j adapter) for routing (to console, file, etc...)

Example: Centralized Logging with Logback

The term *Centralized Logging* refers to a forced logging configuration for the Jetty Server and all web applications that are deployed on the server. It routes all logging events from the web applications to a single configuration on the Server side.

The example below shows how to accomplish this with Jetty and Slf4j, using Logback to manage the final writing of logs to disk.

Important



This mechanism forces all webapps to use the server's configuration for logging, something that isn't 100% appropriate for all webapps.

An example would be having Jenkins-CI deployed as an webapp, if you force its logging configuration to the server side, you lose the ability on [Jenkins-CI](#) to see the logs from the various builds (as now those logs are actually going to the main server log)

This configuration is essentially the multiple logger configuration with added configuration to the deployers to force a WebAppClassLoader change to use the server classpath over the webapps classpath for the logger specific classes.

The technique used by this configuration is to provide an [AppLifecycle.Binding](#) against the ["deploying" node](#) that modifies the [WebApplicationContext.addSystemClass\(String\)](#) for the common logging classes. (See [org.eclipse.jetty.logging.CentralizedWebAppLoggingBinding](#) for actual implementation)

Quick Setup of Centralized Logging using Jetty 9.2.1+

A convenient replacement logging module has been created to bootstrap your \${jetty.base} directory for capturing all Jetty server logging from multiple logging frameworks into a single logging output file managed by logback.

```
[mybase]$ mkdir modules
[mybase]$ cd modules
[modules]$ curl -O https://raw.githubusercontent.com/jetty-project/logging-modules/master/capture-all/logging.mod
  % Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload   Total Spent  Left Speed
100  1416  100  1416    0     0  4241      0 --:--:-- --:--:-- 4252
[master]$ curl -O https://raw.githubusercontent.com/jetty-project/logging-modules/master/centralized/webapp-logging.mod
  % Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload   Total Spent  Left Speed
100   660  100   660    0     0  2032      0 --:--:-- --:--:-- 2037
[modules]$ cd ..
[mybase]$ java -jar /opt/jetty-dist/start.jar --add-to-start=logging,webapp-logging
INFO: logging initialised in ${jetty.base}/start.ini (appended)
MKDIR: ${jetty.base}/logs
DOWNLOAD: http://central.maven.org/maven2/org/slf4j/slf4j-api/1.6.6/slf4j-api-1.6.6.jar
to lib/logging/slf4j-api-1.6.6.jar
DOWNLOAD: http://central.maven.org/maven2/org/slf4j/log4j-over-slf4j/1.6.6/log4j-over-
slf4j-1.6.6.jar to lib/logging/log4j-over-slf4j-1.6.6.jar
DOWNLOAD: http://central.maven.org/maven2/org/slf4j/jul-to-slf4j/1.6.6/jul-to-
slf4j-1.6.6.jar to lib/logging/jul-to-slf4j-1.6.6.jar
DOWNLOAD: http://central.maven.org/maven2/org/slf4j/jcl-over-slf4j/1.6.6/jcl-over-
slf4j-1.6.6.jar to lib/logging/jcl-over-slf4j-1.6.6.jar
DOWNLOAD: http://central.maven.org/maven2/ch/qos/logback/logback-core/1.0.7/logback-
core-1.0.7.jar to lib/logging/logback-core-1.0.7.jar
DOWNLOAD: http://central.maven.org/maven2/ch/qos/logback/logback-classic/1.0.7/logback-
classic-1.0.7.jar to lib/logging/logback-classic-1.0.7.jar
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/capture-
all/logback.xml to resources/logback.xml
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/capture-
all/jetty-logging.properties to resources/jetty-logging.properties
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/logging-modules/master/capture-
all/jetty-logging.xml to etc/jetty-logging.xml
INFO: resources initialised transitively
INFO: resources enabled in ${jetty.base}/start.ini
INFO: webapp-logging initialised in ${jetty.base}/start.ini (appended)
DOWNLOAD: http://central.maven.org/maven2/org/eclipse/jetty/jetty-webapp-logging/9.0.0/
jetty-webapp-logging-9.0.0.jar to lib/webapp-logging/jetty-webapp-logging-9.0.0.jar
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/jetty-webapp-logging/master/
src/main/config/etc/jetty-webapp-logging.xml to etc/jetty-webapp-logging.xml
DOWNLOAD: https://raw.githubusercontent.com/jetty-project/jetty-webapp-logging/master/
src/main/config/etc/jetty-mdc-handler.xml to etc/jetty-mdc-handler.xml
INFO: deploy initialised transitively
INFO: deploy enabled in ${jetty.base}/start.ini
INFO: logging initialised transitively
```

```
INFO: resources      initialised transitively
INFO: resources      enabled in      ${jetty.base}/start.ini
[mybase]$ java -jar /opt/jetty-dist/start.jar
```

That's cool and all, but tell me what that just did.

The replacement `logging.mod` performs a number of tasks.

1. `mybase` is a `${jetty.base}` directory
2. The jetty-distribution is unpacked (and untouched) into `/opt/jetty-dist/` and becomes the `${jetty.home}` directory for this demonstration.
3. The `curl` command downloads the replacement `logging.mod` and puts it into the `${jetty.base}/modules/` directory for use by `mybase` only.
4. The `start.jar --add-to-start=logging,webapp-logging` command performs a number of steps to make the logging module available to the `${jetty.base}` configuration.
 - a. Several entries are added to the `${jetty.base}/start.ini` configuration
 - i. `--module=logging` is added to enable the logging module
 - ii. `--module=webapp-logging` is added to enable the webapp-logging module
 - b. Required `${jetty.base}` directories are created: `${jetty.base}/logs` and `${jetty.base}/resources`
 - c. Required logging libraries are downloaded (if not present already):
 - `slf4j-api.jar` - API jar for Slf4j (used by most of the rest of the jars)
 - `log4j-over-slf4j.jar` - Slf4j jar that captures all log4j emitted logging events
 - `jul-to-slf4j.jar` - Slf4j jar that captures all `java.util.logging` events
 - `jcl-over-slf4j.jar` - Slf4j jar that captures all commons-logging events
 - `logback-classic.jar` - the Slf4j adapter jar that routes all of the captured logging events to logback itself.
 - `logback-core.jar` - the logback implementation jar, that handles all of the filtering and output of the logging events.

These libraries are put in the `${jetty.base}/lib/logging/` directory.

- d. Required webapp-logging library are downloaded (if not present already):
 - `jetty-webapp-logging.jar` - the jetty side deployment manager app-lifecycle bindings for modifying the WebAppClassloaders of deployed webapps.

This library is put in the `${jetty.base}/lib/webapp-logging/` directory.

- e. Required configuration files are downloaded (if not present already): `jetty-logging.properties`, and `logback.xml`

The configuration files are put in the `${jetty.base}/resources/` directory.

- f. Required initialization commands are downloaded (if not present already): `jetty-logging.xml`, `jetty-webapp-logging.xml`, and `jetty-mdc-handler.xml`

These xml files are put in the `${jetty.base}/etc/` directory.

5. At this point you have your mybase configured so that the jetty server itself will log using slf4j, and all other logging events from other Jetty Server components (such as database drivers, security layers, jsp, mail, and other 3rd party server components) are routed to logback for filtering and output.

All webapps deployed via the DeploymentManager have their WebAppClassLoader modified to use server side classes and configuration for all logging implementations.

You can verify the server classpath by using the **start.jar --list-config** command.

In essence, Jetty is now configured to emit its own logging events to slf4j, and various slf4j bridge jars are acting on behalf of log4j, java.util.logging, and commons-logging, routing all of the logging events to logback (a slf4j adapter) for routing (to console, file, etc...)

Jetty Dump Tool

The dump feature in Jetty provides a good snapshot of the status of the threadpool, select sets, class-loaders, and so forth. To get maximum detail from the dump, you need to `setDetailDump(true)` on any `QueuedThreadPools` you are using. You can do this by a direct call if you are embedding Jetty, or in `jetty.xml`.

Configuring the Dump Feature in `jetty.xml`

You can request that Jetty do a dump immediately after starting and just before stopping by calling the appropriate setters on the Server instance. You can accomplish this in `jetty.xml` with:

```
<Set name="dumpAfterStart">true</Set>
<Set name="dumpBeforeStop">true</Set>
```

Extra ThreadPool Information

You can get additional detail from the `QueuedThreadPool` if `setDetailedDump(true)` is called on the thread pool instance. Do this in `jetty.xml` as follows:

```
<Configure id="Server" class="org.eclipse.jetty.server.Server">
  <!-- ===== -->
  <!-- Server Thread Pool
       -->
  <!-- ===== -->
  <Set name="ThreadPool">
    <!-- Default queued blocking threadpool -->
    <New class="org.eclipse.jetty.util.thread.QueuedThreadPool">
      <Set name="minThreads">10</Set>
      <Set name="maxThreads">200</Set>
      <Set name="detailedDump">true</Set>
    </New>
  </Set>
```

Using the Dump Feature via JMX

The dump method is on the Server instance and many of its nested components (Handlers, Connectors, and so forth). Dumps may be obtained by calling these methods either in code or via JMX (see the section called “Using Java Management Extensions (JMX)”).

The Server MBean has a `dump()` method, which dumps everything, plus a `dumpStdErr()` operation that dumps to stderr rather than replying to jconsole.

Examining a Jetty Distribution Dump

This is a dump of the stock jetty-distribution with extra threadpool information:

```

org.eclipse.jetty.server.Server@76f08fe1 - STARTING
+- qtp1062680061{STARTED,10<=i<=200,i=1,q=0} - STARTED
|   +- 12 qtp1062680061-12-selector-0 RUNNABLE
|   |   +- sun.nio.ch.KQueueArrayWrapper.kevent0(Native Method)
|   |   +- sun.nio.ch.KQueueArrayWrapper.poll(KQueueArrayWrapper.java:159)
|   |   +- sun.nio.ch.KQueueSelectorImpl.doSelect(KQueueSelectorImpl.java:103)
|   |   +- sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:102)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.run(SelectorManager.java:435)
|   |   +- org.eclipse.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:596)
|   |   +- org.eclipse.util.thread.QueuedThreadPool
$3.run(QueuedThreadPool.java:527)
|   |   +- java.lang.Thread.run(Thread.java:722)
+- 13 qtp1062680061-13-selector-6 RUNNABLE
|   |   +- sun.nio.ch.KQueueArrayWrapper.kevent0(Native Method)
|   |   +- sun.nio.ch.KQueueArrayWrapper.poll(KQueueArrayWrapper.java:159)
|   |   +- sun.nio.ch.KQueueSelectorImpl.doSelect(KQueueSelectorImpl.java:103)
|   |   +- sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:102)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.run(SelectorManager.java:435)
|   |   +- org.eclipse.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:596)
|   |   +- org.eclipse.util.thread.QueuedThreadPool
$3.run(QueuedThreadPool.java:527)
|   |   +- java.lang.Thread.run(Thread.java:722)
+- 14 qtp1062680061-14-selector-5 RUNNABLE
|   |   +- sun.nio.ch.KQueueArrayWrapper.kevent0(Native Method)
|   |   +- sun.nio.ch.KQueueArrayWrapper.poll(KQueueArrayWrapper.java:159)
|   |   +- sun.nio.ch.KQueueSelectorImpl.doSelect(KQueueSelectorImpl.java:103)
|   |   +- sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:102)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.run(SelectorManager.java:435)
|   |   +- org.eclipse.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:596)
|   |   +- org.eclipse.util.thread.QueuedThreadPool
$3.run(QueuedThreadPool.java:527)
|   |   +- java.lang.Thread.run(Thread.java:722)
+- 15 qtp1062680061-15-acceptor-0-ServerConnector@3d0f282{HTTP/1.1}{0.0.0.0:9090}
BLOCKED
|   |   +- sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:210)
|   |   +- org.eclipse.server.ServerConnector.accept(ServerConnector.java:284)
|   |   +- org.eclipse.server.AbstractConnector
$Acceptor.run(AbstractConnector.java:460)
|   |   +- org.eclipse.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:596)
|   |   +- org.eclipse.util.thread.QueuedThreadPool
$3.run(QueuedThreadPool.java:527)
|   |   +- java.lang.Thread.run(Thread.java:722)
+- 16 qtp1062680061-16-selector-1 RUNNABLE
|   |   +- sun.nio.ch.KQueueArrayWrapper.kevent0(Native Method)
|   |   +- sun.nio.ch.KQueueArrayWrapper.poll(KQueueArrayWrapper.java:159)
|   |   +- sun.nio.ch.KQueueSelectorImpl.doSelect(KQueueSelectorImpl.java:103)
|   |   +- sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:102)

```

```

|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.run(SelectorManager.java:435)
|   |
|   +- org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:596)
|   |   +- org.eclipse.jetty.util.thread.QueuedThreadPool
$3.run(QueuedThreadPool.java:527)
|   |   +- java.lang.Thread.run(Thread.java:722)
|   +- 17 qtp1062680061-17-selector-2 RUNNABLE
|   |   +- sun.nio.ch.KQueueArrayWrapper.kevent0(Native Method)
|   |   +- sun.nio.ch.KQueueArrayWrapper.poll(KQueueArrayWrapper.java:159)
|   |   +- sun.nio.ch.KQueueSelectorImpl.doSelect(KQueueSelectorImpl.java:103)
|   |   +- sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:102)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.run(SelectorManager.java:435)
|   |
|   +- org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:596)
|   |   +- org.eclipse.jetty.util.thread.QueuedThreadPool
$3.run(QueuedThreadPool.java:527)
|   |   +- java.lang.Thread.run(Thread.java:722)
|   +- 18 qtp1062680061-18-selector-3 RUNNABLE
|   |   +- sun.nio.ch.KQueueArrayWrapper.kevent0(Native Method)
|   |   +- sun.nio.ch.KQueueArrayWrapper.poll(KQueueArrayWrapper.java:159)
|   |   +- sun.nio.ch.KQueueSelectorImpl.doSelect(KQueueSelectorImpl.java:103)
|   |   +- sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:102)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.run(SelectorManager.java:435)
|   |
|   +- org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:596)
|   |   +- org.eclipse.jetty.util.thread.QueuedThreadPool
$3.run(QueuedThreadPool.java:527)
|   |   +- java.lang.Thread.run(Thread.java:722)
|   +- 19 qtp1062680061-19-selector-4 RUNNABLE
|   |   +- sun.nio.ch.KQueueArrayWrapper.kevent0(Native Method)
|   |   +- sun.nio.ch.KQueueArrayWrapper.poll(KQueueArrayWrapper.java:159)
|   |   +- sun.nio.ch.KQueueSelectorImpl.doSelect(KQueueSelectorImpl.java:103)
|   |   +- sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:102)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.run(SelectorManager.java:435)
|   |
|   +- org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:596)
|   |   +- org.eclipse.jetty.util.thread.QueuedThreadPool
$3.run(QueuedThreadPool.java:527)
|   |   +- java.lang.Thread.run(Thread.java:722)
|   +- 20 qtp1062680061-20-selector-7 RUNNABLE
|   |   +- sun.nio.ch.KQueueArrayWrapper.kevent0(Native Method)
|   |   +- sun.nio.ch.KQueueArrayWrapper.poll(KQueueArrayWrapper.java:159)
|   |   +- sun.nio.ch.KQueueSelectorImpl.doSelect(KQueueSelectorImpl.java:103)
|   |   +- sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
|   |   +- sun.nio.ch.SelectorImpl.select(SelectorImpl.java:102)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.run(SelectorManager.java:435)
|   |
|   +- org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:596)
|   |   +- org.eclipse.jetty.util.thread.QueuedThreadPool
$3.run(QueuedThreadPool.java:527)
|   |   +- java.lang.Thread.run(Thread.java:722)
|   +- 21 qtp1062680061-21-acceptor-1-ServerConnector@3d0f282{HTTP/1.1}{0.0.0.0:9090}
RUNNABLE
|   |   +- sun.nio.ch.ServerSocketChannelImpl.accept0(Native Method)
|   |   +- sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:226)

```

```

| | | +- org.eclipse.jetty.server.ServerConnector.accept(ServerConnector.java:284)
| | +- org.eclipse.jetty.server.AbstractConnector
$Acceptor.run(AbstractConnector.java:460)
| |
| +- org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:596)
| | +- org.eclipse.jetty.util.thread.QueuedThreadPool
$3.run(QueuedThreadPool.java:527)
| | +- java.lang.Thread.run(Thread.java:722)
| +- 49 qtp1062680061-49-acceptor-2-ServerConnector@3d0f282{HTTP/1.1}{0.0.0.0:9090}
BLOCKED
| | +- sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:210)
| | +- org.eclipse.jetty.server.ServerConnector.accept(ServerConnector.java:284)
| | +- org.eclipse.jetty.server.AbstractConnector
$Acceptor.run(AbstractConnector.java:460)
| |
| +- org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:596)
| | +- org.eclipse.jetty.util.thread.QueuedThreadPool
$3.run(QueuedThreadPool.java:527)
| | +- java.lang.Thread.run(Thread.java:722)
| +- 50 qtp1062680061-50-acceptor-3-ServerConnector@3d0f282{HTTP/1.1}{0.0.0.0:9090}
BLOCKED
| | +- sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:210)
| | +- org.eclipse.jetty.server.ServerConnector.accept(ServerConnector.java:284)
| | +- org.eclipse.jetty.server.AbstractConnector
$Acceptor.run(AbstractConnector.java:460)
| |
| +- org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:596)
| | +- org.eclipse.jetty.util.thread.QueuedThreadPool
$3.run(QueuedThreadPool.java:527)
| | +- java.lang.Thread.run(Thread.java:722)
| +- 52 qtp1062680061-52 TIMED_WAITING IDLE
+= org.eclipse.jetty.util.thread.ScheduledExecutorScheduler@725f5 - STARTED
+= org.eclipse.jetty.server.handler.HandlerCollection@58b37561 - STARTED
| += org.eclipse.jetty.server.handler.ContextHandlerCollection@64c6e290 - STARTED
| | += org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | += o.e.j.w.WebAppContext@7ea8b1c{/async-rest,[file:/private/var/folders/
br/kbs2g3753c54wmv4j31pnw5r000gn/T/jetty-0.0.0-9090-async-rest.war-_async-
any-/webapp/, jarfile:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r000gn/T/
jetty-0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/example-async-rest-
jar-9.0.2.v20130417.jar!/_META-INF/resources/],AVAILABLE}{/async-rest.war} - STARTED
| | | += org.eclipse.jetty.server.session.SessionHandler@6dfb8d2e - STARTED
| | | | += org.eclipse.jetty.server.session.HashSessionManager@6cb83869 -
STARTED
| | | | | += org.eclipse.jetty.security.ConstraintSecurityHandler@2848c90e -
STARTED
| | | | | | += org.eclipse.jetty.security.DefaultAuthenticatorFactory@52b12fef
| | | | | | += org.eclipse.jetty.servlet.ServletHandler@46bac287 - STARTED
| | | | | |
default@5c13d641==org.eclipse.jetty.servlet.DefaultServlet,0,true - STARTED
| | | | | | +- maxCacheSize=256000000
| | | | | | +- etags=true
| | | | | | +- dirAllowed=true
| | | | | | +- gzip=true
| | | | | | +- maxCachedFileSize=200000000
| | | | | | +- redirectWelcome=false
| | | | | | +- acceptRanges=true
| | | | | | +- welcomeServlets=false
| | | | | | +- aliases=false
| | | | | | +- useFileMappedBuffer=true
| | | | | | +- maxCachedFiles=2048
| | | | | | +- [/]=>default
| | | | | | += jsp@19c47==org.apache.jasper.servlet.JspServlet,0,true -
STARTED
| | | | | | | | +- logVerbosityLevel=DEBUG
| | | | | | | | +- fork=false
| | | | | | | | +- com.sun.appserv.jsp.classpath=/Users/jesse/
Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-xml-9.0.2.v20130417.jar:/-
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/servlet-
api-3.0.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
http-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
lib/jetty-continuation-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/lib/jetty-server-9.0.2.v20130417.jar:/Users/jesse/Desktop/
jetty-distribution-9.0.2.v20130417/lib/jetty-security-9.0.2.v20130417.jar:/Users/jesse/
Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-servlet-9.0.2.v20130417.jar:/-
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
webapp-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
lib/jetty-deploy-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-

```

```
distribution-9.0.2.v20130417/lib/jetty-client-9.0.2.v20130417.jar:/Users/jesse/Desktop/
jetty-distribution-9.0.2.v20130417/lib/jetty-jmx-9.0.2.v20130417.jar:/Users/jesse/
Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/com.sun.el-2.2.0.v201303151357.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
javax.el-2.2.0.v201303151357.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
lib/jsp/javax.servlet.jsp.jstl-1.2.0.v201105211821.jar:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/lib/jsp/javax.servlet.jsp-2.2.0.v201112011158.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
org.apache.jasper.glassfish-2.2.2.v201112011158.jar:/Users/
jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
org.apache.taglibs.standard.glassfish-1.2.0.v201112081803.jar:/Users/jesse/Desktop/
jetty-distribution-9.0.2.v20130417/lib/jsp/org.eclipse.jdt.core-3.8.2.v20130121.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/resources:/Users/jesse/Desktop/
jetty-distribution-9.0.2.v20130417/lib/websocket/websocket-api-9.0.2.v20130417.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/websocket/websocket-
common-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
lib/websocket/websocket-server-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/lib/websocket/websocket-servlet-9.0.2.v20130417.jar:/Users/
jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-util-9.0.2.v20130417.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-io-9.0.2.v20130417.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/start.jar:/Library/Java/
JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/Home/jre/lib/ext/dnsns.jar:/Library/
Java/JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/Home/jre/lib/ext/locatedata.jar:/
Library/Java/JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/Home/jre/lib/ext/
sunec.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/Home/jre/lib/ext/
sunjce_provider.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/
Home/jre/lib/ext/sunpkcs11.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0_17.jdk/
Contents/Home/jre/lib/ext/zipfs.jar:/opt/local/lib/libsvnjavahl-1.0.dylib:/System/
Library/Java/Extensions/AppleScriptEngine.jar:/System/Library/Java/Extensions/
dns_sd.jar:/System/Library/Java/Extensions/j3daudio.jar:/System/Library/Java/Extensions/
j3dcore.jar:/System/Library/Java/Extensions/j3dutils.jar:/System/Library/Java/Extensions/
jai_codec.jar:/System/Library/Java/Extensions/jai_core.jar:/System/Library/Java/
Extensions/libAppleScriptEngine.jnilib:/System/Library/Java/Extensions/libJ3D.jnilib:/
System/Library/Java/Extensions/libJ3DAudio.jnilib:/System/Library/Java/Extensions/
libJ3DUtils.jnilib:/System/Library/Java/Extensions/libmllib_jai.jnilib:/System/Library/
Java/Extensions/libQTJNative.jnilib:/System/Library/Java/Extensions/mlibwrapper_jai.jar:/
System/Library/Java/Extensions/MRJToolkit.jar:/System/Library/Java/Extensions/
QTJava.zip:/System/Library/Java/Extensions/vecmath.jar:/usr/lib/java/libjdns_sd.jnilib
| | | | | | | +- scratchdir=/private/var/folders/br/
| | | | | | | +- xpoweredBy=false
| | | | | | | +- [* .jsp, *.jspx, *.jspx, *.xsp, *.JSP, *.JSPF, *.JSPX,
*.XSP]=>jsp
| | | | | | | +=
SerialRestServlet@461411d==org.eclipse.jetty.example.asyncrest.SerialRestServlet,-1,false
- STARTED
| | | | | | | +- [/testSerial]=>SerialRestServlet
| | | | | | | +=
AsyncRestServlet@73eb9bd5==org.eclipse.jetty.example.asyncrest.AsyncRestServlet,-1,false
- STARTED
| | | | | | | +- [/testAsync]=>AsyncRestServlet
| | | | | | | +- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | | | | | | +- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | | | | | | +- HashLoginService[Test Realm] - STARTED
| | | | | | | +- org.eclipse.jetty.security.DefaultIdentityService@d2539a6
| | | | | | |
org.eclipse.jetty.security.authentication.BasicAuthenticator@7b239469
| | | | | | | +> HashLoginService[Test Realm] - STARTED
| | | | | | | +- org.eclipse.jetty.security.DefaultIdentityService@d2539a6
| | | | | | |
org.eclipse.jetty.security.authentication.BasicAuthenticator@7b239469
| | | | | | | +> []
| | | | | | | +- /={TRACE={RoleInfo,F,C[]}}
| | | | | | | +- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
|= org.eclipse.jetty.servlet.ErrorPageErrorHandler@3c121009 - STARTED
| | | | | | | +- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | | | | | | +- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | | | | | |
| | | | | | | +> WebAppClassLoader=Async REST Webservice Example@52934ea0
| | | | | | | +- file:/private/var/folders/br/kbs2g3753c54wmv4j3lpnw5r0000gn/T/
jetty-0.0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/classes/
| | | | | | | +- file:/private/var/folders/br/kbs2g3753c54wmv4j3lpnw5r0000gn/T/
jetty-0.0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/example-async-rest-
jar-9.0.2.v20130417.jar
```

```

| | | | +- file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/jetty-
client-9.0.2.v20130417.jar
| | | | +- file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/jetty-
http-9.0.2.v20130417.jar
| | | | +- file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/jetty-
io-9.0.2.v20130417.jar
| | | | +- file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/jetty-
util-9.0.2.v20130417.jar
| | | | +- file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/T/
jetty-0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/jetty-util-
ajax-9.0.2.v20130417.jar
| | | | +- startJarLoader@7194b34a
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-xml-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
servlet-api-3.0.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-http-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-continuation-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-server-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-security-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-servlet-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-webapp-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-deploy-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-client-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-jmx-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/com.sun.el-2.2.0.v201303151357.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/javax.el-2.2.0.v201303151357.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/javax.servlet.jsp.jstl-1.2.0.v201105211821.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/javax.servlet.jsp-2.2.0.v201112011158.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/org.apache.jasper.glassfish-2.2.2.v201112011158.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/org.apache.taglibs.standard.glassfish-1.2.0.v201112081803.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/org.eclipse.jdt.core-3.8.2.v20130121.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
resources/
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
websocket/websocket-api-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
websocket/websocket-common-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
websocket/websocket-server-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
websocket/websocket-servlet-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-util-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-io-9.0.2.v20130417.jar
| | | | +- sun.misc.Launcher$AppClassLoader@19d1b44b
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
start.jar
| | | | +- sun.misc.Launcher$ExtClassLoader@1693b52b
| | | | +> javax.servlet.context.tempdir=/private/var/folders/br/
kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-9090-async-rest.war-_async-rest-any-
| | | | +> org.apache.catalina.jsp_classpath=/private/var/folders/br/
kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-9090-async-rest.war-_async-rest-any-
/webapp/WEB-INF/classes:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/T/
jetty-0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/example-async-
rest-jar-9.0.2.v20130417.jar:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/

```

```

T/jetty-0.0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/jetty-
client-9.0.2.v20130417.jar:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/jetty-
http-9.0.2.v20130417.jar:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/jetty-
io-9.0.2.v20130417.jar:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/jetty-
util-9.0.2.v20130417.jar:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/T/
jetty-0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/jetty-util-
ajax-9.0.2.v20130417.jar
| | | +> org.eclipse.jetty.server.webapp.ContainerIncludeJarPattern=.*/servlet-
api-[^/]*\.jar$ 
| | | +> com.sun.jsp.taglibraryCache={}
| | | +> com.sun.jsp.tagFileJarUrlsCache={}
| += o.e.j.s.h.MovedContextHandler@5e0c8d24{/oldContextPath,null,AVAILABLE} -
STARTED
| | | += org.eclipse.jetty.server.handler.MovedContextHandler$Redirector@2a4200d3
- STARTED
| | | | +~ org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | | | +~ org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | | |
| | | +> No ClassLoader
| | | +> org.eclipse.jetty.server.webapp.ContainerIncludeJarPattern=.*/servlet-
api-[^/]*\.jar$ 
| | | += o.e.j.w.WebAppContext@6f01ba6f{/,file:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/webapps/ROOT/,AVAILABLE}{/ROOT} - STARTED
| | | | += org.eclipse.jetty.server.session.SessionHandler@5a770658 - STARTED
| | | | | += org.eclipse.jetty.server.session.HashSessionManager@746a95ae -
STARTED
| | | | | += org.eclipse.jetty.security.ConstraintSecurityHandler@1890e38 -
STARTED
| | | | | | +- org.eclipse.jetty.security.DefaultAuthenticatorFactory@6242c657
| | | | | | += org.eclipse.jetty.servlet.ServletHandler@debac27 - STARTED
| | | | | |
default@5c13d641==org.eclipse.jetty.servlet.DefaultServlet,0,true - STARTED
| | | | | | +- maxCacheSize=256000000
| | | | | | +- etags=true
| | | | | | +- dirAllowed=true
| | | | | | +- gzip=true
| | | | | | +- maxCachedFileSize=200000000
| | | | | | +- redirectWelcome=false
| | | | | | +- acceptRanges=true
| | | | | | +- welcomeServlets=false
| | | | | | +- aliases=false
| | | | | | +- useFileMappedBuffer=true
| | | | | | +- maxCachedFiles=2048
| | | | | | +- [/]=>default
| | | | | | += jsp@19c47==org.apache.jasper.servlet.JspServlet,0,true -
STARTED
| | | | | | | | +- logVerbosityLevel=DEBUG
| | | | | | | | +- fork=false
| | | | | | | | +- com.sun.appserv.jsp.classpath=/Users/jesse/
Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-xml-9.0.2.v20130417.jar:/-
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/servlet-
api-3.0.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
http-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/-
lib/jetty-continuation-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/lib/jetty-server-9.0.2.v20130417.jar:/Users/jesse/Desktop/-
jelly-distribution-9.0.2.v20130417/lib/jetty-security-9.0.2.v20130417.jar:/Users/jesse/-
Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-servlet-9.0.2.v20130417.jar:/-
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
webapp-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/-
lib/jetty-deploy-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/lib/jetty-client-9.0.2.v20130417.jar:/Users/jesse/Desktop/-
jelly-distribution-9.0.2.v20130417/lib/jetty-jmx-9.0.2.v20130417.jar:/Users/jesse/-
Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/com.sun.el-2.2.0.v201303151357.jar:/-
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
javax.el-2.2.0.v201303151357.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/-
lib/jsp/javax.servlet.jsp.jstl-1.2.0.v201105211821.jar:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/lib/jsp/javax.servlet.jsp-2.2.0.v201112011158.jar:/-
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
org.apache.jasper.glassfish-2.2.2.v201112011158.jar:/Users/jesse/-
Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
org.apache.taglibs.standard.glassfish-1.2.0.v201112081803.jar:/Users/jesse/Desktop/-
jelly-distribution-9.0.2.v20130417/lib/jsp/org.eclipse.jdt.core-3.8.2.v20130121.jar:/-
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/resources:/Users/jesse/Desktop/-
jelly-distribution-9.0.2.v20130417/lib/websocket/websocket-api-9.0.2.v20130417.jar:/

```



```

| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/javax.servlet.jsp-2.2.0.v201112011158.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/org.apache.jasper.glassfish-2.2.2.v201112011158.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/org.apache.taglibs.standard.glassfish-1.2.0.v201112081803.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/org.eclipse.jdt.core-3.8.2.v20130121.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
resources/
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
websocket/websocket-api-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
websocket/websocket-common-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
websocket/websocket-server-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
websocket/websocket-servlet-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-util-9.0.2.v20130417.jar
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-io-9.0.2.v20130417.jar
| | | | +- sun.misc.Launcher$AppClassLoader@19d1b44b
| | | | +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
start.jar
| | | | +- sun.misc.Launcher$ExtClassLoader@1693b52b
| | | | +> javax.servlet.context.tempdir=/private/var/folders/br/
kbs2g3753c54wmv4j31pnw5r000gn/T/jetty-0.0.0-9090-ROOT-_any-
| | | | +> org.eclipse.jetty.server.webapp.ContainerIncludeJarPattern=.*/servlet-
api-[^/]*\.jar$ 
| | | | +> com.sun.jsp.taglibraryCache={}
| | | | +> com.sun.jsp.tagFileJarUrlsCache={}
| | | += o.e.j.s.h.ContextHandler@7b2dffdf{/javadoc,file:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/javadoc,AVAILABLE} - STARTED
| | | | += org.eclipse.jetty.server.handler.ResourceHandler@8f9c8a7 - STARTED
| | | | +~ org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | | | +~ org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | | |
| | | | +> No ClassLoader
| | | | +> org.eclipse.jetty.server.webapp.ContainerIncludeJarPattern=.*/servlet-
api-[^/]*\.jar$ 
| | | | += o.e.j.w.WebAppContext@716d9094{/test,file:/private/var/folders/
br/kbs2g3753c54wmv4j31pnw5r000gn/T/jetty-0.0.0-9090-test.war-_test-any-
webapp/,AVAILABLE}{/test.war} - STARTED
| | | | | += org.eclipse.jetty.server.session.SessionHandler@336abd81 - STARTED
| | | | | += org.eclipse.jetty.server.session.HashSessionManager@1246f8d0 - 
STARTED
| | | | | | += org.eclipse.jetty.security.ConstraintSecurityHandler@7179290f - 
STARTED
| | | | | | | += org.eclipse.jetty.security.DefaultAuthenticatorFactory@17d41d12
| | | | | | | += org.eclipse.jetty.servlet.ServletHandler@5034037e - STARTED
| | | | | | | | += 
default@5c13d641==org.eclipse.jetty.servlet.DefaultServlet,0,true - STARTED
| | | | | | | | | +- maxCacheSize=256000000
| | | | | | | | | +- etags=true
| | | | | | | | | +- dirAllowed=true
| | | | | | | | | +- gzip=true
| | | | | | | | | +- maxCachedFileSize=200000000
| | | | | | | | | +- redirectWelcome=false
| | | | | | | | | +- acceptRanges=true
| | | | | | | | | +- welcomeServlets=false
| | | | | | | | | +- aliases=false
| | | | | | | | | +- useFileMappedBuffer=true
| | | | | | | | | +- maxCachedFiles=2048
| | | | | | | | | |+- [ / ]=>default
| | | | | | | | | += jsp@19c47==org.apache.jasper.servlet.JspServlet,0,true -
STARTED
| | | | | | | | | | +- logVerbosityLevel=DEBUG
| | | | | | | | | | +- fork=false
| | | | | | | | | | +- com.sun.appserv.jsp.classpath=/Users/jesse/
Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-xml-9.0.2.v20130417.jar:/ 
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/servlet-
api-3.0.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
http-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
lib/jetty-continuation-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
distribution-9.0.2.v20130417/lib/jetty-server-9.0.2.v20130417.jar:/Users/jesse/Desktop/
jetty-distribution-9.0.2.v20130417/lib/jetty-security-9.0.2.v20130417.jar:/Users/jesse/
```

```

Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-servlet-9.0.2.v20130417.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
webapp-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
lib/jetty-deploy-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/lib/jetty-client-9.0.2.v20130417.jar:/Users/jesse/Desktop/
jetty-distribution-9.0.2.v20130417/lib/jetty-jmx-9.0.2.v20130417.jar:/Users/jesse/
Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/com.sun.el-2.2.0.v201303151357.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
javax.el-2.2.0.v201303151357.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
lib/jsp/javax.servlet.jsp.jstl-1.2.0.v201105211821.jar:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/lib/jsp/javax.servlet.jsp-2.2.0.v201112011158.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
org.apache.jasper.glassfish-2.2.2.v201112011158.jar:/Users/
jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
org.apache.taglibs.standard.glassfish-1.2.0.v201112081803.jar:/Users/jesse/Desktop/
jetty-distribution-9.0.2.v20130417/lib/jsp/org.eclipse.jdt.core-3.8.2.v20130121.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/resources:/Users/jesse/Desktop/
jetty-distribution-9.0.2.v20130417/lib/websocket/websocket-api-9.0.2.v20130417.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/websocket/websocket-
common-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
lib/websocket/websocket-server-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/lib/websocket/websocket-servlet-9.0.2.v20130417.jar:/Users/
jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-util-9.0.2.v20130417.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-io-9.0.2.v20130417.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/start.jar:/Library/Java/
JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/Home/jre/lib/ext/dnsns.jar:/Library/
Java/JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/Home/jre/lib/ext/locatedata.jar:/
Library/Java/JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/Home/jre/lib/ext/
sunec.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/Home/jre/lib/
ext/sunjce_provider.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/
Home/jre/lib/ext/sunpkcs11.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0_17.jdk/
Contents/Home/jre/lib/ext/zipfs.jar:/opt/local/lib/libsvnjavahl-1.0.dylib:/System/
Library/Java/Extensions/AppleScriptEngine.jar:/System/Library/Java/Extensions/
dns_sd.jar:/System/Library/Java/Extensions/j3daudio.jar:/System/Library/Java/Extensions/
j3dcore.jar:/System/Library/Java/Extensions/j3dutils.jar:/System/Library/Java/Extensions/
jai_codec.jar:/System/Library/Java/Extensions/jai_core.jar:/System/Library/Java/
Extensions/libAppleScriptEngine.jnilib:/System/Library/Java/Extensions/libJ3D.jnilib:/
System/Library/Java/Extensions/libJ3DAudio.jnilib:/System/Library/Java/Extensions/
libJ3DUtils.jnilib:/System/Library/Java/Extensions/libmllib_jai.jnilib:/System/Library/
Java/Extensions/libQTJNative.jnilib:/System/Library/Java/Extensions/mlibwrapper_jai.jar:/
System/Library/Java/Extensions/MRJToolkit.jar:/System/Library/Java/Extensions/
QTJava.zip:/System/Library/Java/Extensions/vecmath.jar:/usr/lib/java/libjdns_sd.jnilib
| | | | | | | | +- scratchdir=/private/var/folders/br/
kbs2g3753c54wmv4j3lpnw5r000gn/T/jetty-0.0.0-9090-test.war-_test-any-/jsp
| | | | | | | | +- xpoweredBy=false
| | | | | | | | +- [*.jsp, *.jspx, *.jpx, *.xsp, *.JSP, *.JSPF, *.JSPX,
*.XSP]=>jsp
| | | | | | | | +- QoSFilter - STARTED
| | | | | | | | | +- managedAttr=true
| | | | | | | | | +- maxRequests=10000
| | | | | | | | +- [//*]/[]==0=>QoSFilter
| | | | | | | |= MultiPart - STARTED
| | | | | | | | | +- deleteFiles=true
| | | | | | | | +- [/dump/*]/[]==0=>MultiPart
| | | | | | | |= GzipFilter - STARTED
| | | | | | | | | +- bufferSize=8192
| | | | | | | | | +- excludedAgents=MSIE 6.0
| | | | | | | | | +- userAgent=(?:Mozilla[^\\()*(compatible;\s*+([^\;]*);.*|
(?::.*?([^\s]+/[^\s]+).*)|+- mimeTypes=text/plain,application/xml
| | | | | | | | | +- uncheckedReader=true
| | | | | | | | | +- cacheSize=1024
| | | | | | | | | +- minGzipSize=2048
| | | | | | | | +- [/dump/gzip/*, *.txt]/[]==0=>GzipFilter
| | | | | | |= Login@462ff49==com.acme.LoginServlet,1,true - STARTED
| | | | | | +- [/login/*]=>Login
| | | | | |= Hello@42628b2==com.acme.HelloWorld,1,true - STARTED
| | | | | +- [/hello/*]=>Hello
| | | | |= Dump@20ae14==com.acme.Dump,1,true - STARTED
| | | | | | +- servlet-override-example=a servlet value
| | | | +- [/dump/*, *.dump]=>Dump
| | | |= Session@d9891a76==com.acme.SessionDump,5,true - STARTED
| | | +- [/session/*]=>Session
| | |= Cookie@78a4f684==com.acme.CookieDump,1,true - STARTED
| | +- [/cookie/*]=>Cookie
| |= Dispatch@14d3a89a==com.acme.DispatchServlet,1,true - STARTED
| +- [/dispatch/*]=>Dispatch

```

```

+-- CGI@10465==org.eclipse.jetty.servlets.CGI,1,true - STARTED
+- [/cgi-bin/*]=>CGI
+= Chat@200778==com.acme.ChatServlet,1,true - STARTED
+- [/chat/*]=>Chat
+= WSChat@99274454==com.acme.WebSocketChatServlet,1,true -
STARTED
+- [/ws/*]=>WSChat
+= Rewrite@a4dac96c==com.acme.RewriteServlet,-1,false - STARTED
+- [/rewritten/*, /redirected/*]=>Rewrite
+= SecureMode@d45951da==com.acme.SecureModeServlet,1,true -
STARTED
+- [/secureMode/*]=>SecureMode
+-
foo.jsp@d7583f1f==org.apache.jasper.servlet.JspServlet,-1,false - STARTED
+- [/jsp/foo/]=>foo.jsp
+- [* .more]=>Dump
+- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
+= RegoTest@dafcdlad==com.acme.RegoTest,-1,false - STARTED
+- [/rego/*]=>RegoTest
+= RegoTest2@849d6425==com.acme.RegoTest,-1,false - STARTED
+- [/rego2/*]=>RegoTest2
+- TestFilter - STARTED
|   +- remote=false
|   +- [*]/[]==31=>TestFilter
+= HashLoginService[Test Realm] - STARTED
+-
org.eclipse.jetty.security.authentication.FormAuthenticator@1fa291f2
+- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
+- org.eclipse.jetty.security.DefaultIdentityService@41917d6d
|
+- HashLoginService[Test Realm] - STARTED
+- org.eclipse.jetty.security.DefaultIdentityService@41917d6d
+-
org.eclipse.jetty.security.authentication.FormAuthenticator@1fa291f2
+- [server-administrator, *, admin, user]
+- /rego2/*={*=RoleInfo,C[server-administrator]}
+- *.htm={*=RoleInfo,C[server-administrator, *, admin, user]}
+- /dump/auth/ssl/*={*=RoleInfo[]}
+- /dump/auth/noaccess/*={*=RoleInfo,F,C[]}
+- /auth/*={*=RoleInfo,F,C[]}
+- /dump/auth/admin/*={*=RoleInfo,C[admin]}
+- /dump/auth/relax/*={GET={RoleInfo[]}, HEAD={RoleInfo[]}}
+- /rego/*={*=RoleInfo,C[admin]}
+- /dump/auth/*={*=RoleInfo,C[server-administrator, *, admin,
user]}
|
+- /=TRACE={RoleInfo,F,C[]}
+- /auth/relax.txt={GET={RoleInfo[]}, HEAD={RoleInfo[]}}
+- /auth2/*={*=RoleInfo,C[server-administrator, *, admin, user]}
+- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
+= org.eclipse.jetty.servlet.ErrorPageErrorHandler@24bf7a86 - STARTED
|   +- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
+- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
+- org.eclipse.jetty.servlets.QoSFilter@6df3d1f5
|
+- WebAppClassLoader=Test WebApp@3e2f3adb
|   +- file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/T/
jetty-0.0.0.0-9090-test.war-_test-any-/webapp/WEB-INF/classes/
|   |   +- file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0.0-9090-test.war-_test-any-/webapp/WEB-INF/lib/jetty-
continuation-9.0.2.v20130417.jar
|   |   +- file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/T/
jetty-0.0.0.0-9090-test.war-_test-any-/webapp/WEB-INF/lib/jetty-http-9.0.2.v20130417.jar
|   |   +- file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/T/
jetty-0.0.0.0-9090-test.war-_test-any-/webapp/WEB-INF/lib/jetty-io-9.0.2.v20130417.jar
|   |   +- file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0.0-9090-test.war-_test-any-/webapp/WEB-INF/lib/jetty-
servlets-9.0.2.v20130417.jar
|   |   +- file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/T/
jetty-0.0.0.0-9090-test.war-_test-any-/webapp/WEB-INF/lib/jetty-util-9.0.2.v20130417.jar
|   |   +- file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0.0-9090-test.war-_test-any-/webapp/WEB-INF/lib/websocket-
api-9.0.2.v20130417.jar
|   |   +- file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0.0-9090-test.war-_test-any-/webapp/WEB-INF/lib/socket-
servlet-9.0.2.v20130417.jar
|   |   +- startJarLoader@7194b34a

```

```

|   |   |   |
|   |   |       +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |           jetty-xml-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   servlet-api-3.0.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jetty-http-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jetty-continuation-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jetty-server-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jetty-security-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jetty-servlet-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jetty-webapp-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jetty-deploy-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jetty-client-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jetty-jmx-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jsp/com.sun.el-2.2.0.v201303151357.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jsp/javax.el-2.2.0.v201303151357.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jsp/javax.servlet.jsp.jstl-1.2.0.v201105211821.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jsp/javax.servlet.jsp-2.2.0.v201112011158.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jsp/org.apache.jasper.glassfish-2.2.2.v201112011158.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jsp/org.apache.taglibs.standard.glassfish-1.2.0.v201112081803.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jsp/org.eclipse.jdt.core-3.8.2.v20130121.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
|   |   |                   resources/
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   websocket/websocket-api-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   websocket/websocket-common-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   websocket/websocket-server-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   websocket/websocket-servlet-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jetty-util-9.0.2.v20130417.jar
|   |   |               +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
|   |   |                   jetty-io-9.0.2.v20130417.jar
|   |   |               +- sun.misc.Launcher$AppClassLoader@19d1b44b
|   |   |                   +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
|   |   |                       start.jar
|   |   |               +- sun.misc.Launcher$ExtClassLoader@1693b52b
|   |   |               +>
|   |   |                   org.eclipse.jetty.server.context.ManagedAttributes=QoSFilter,TransparentProxy.ThreadPool,TransparentProxy
|   |   |               +> context-override-example=a context value
|   |   |               +> javax.servlet.context.tempdir=/private/var/folders/br/
|   |   |                   kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-9090-test.war-_test-any-
|   |   |               +> org.apache.catalina.jsp_classpath=/private/var/folders/br/
|   |   |                   kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-9090-test.war-_test-any-/webapp/WEB-INF/
|   |   |                   classes:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-9090-
|   |   |                   test.war-_test-any-/webapp/WEB-INF/lib/jetty-continuation-9.0.2.v20130417.jar:/private/
|   |   |                   var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-9090-
|   |   |                   test.war-_test-any-/webapp/WEB-INF/lib/jetty-http-9.0.2.v20130417.jar:/private/
|   |   |                   var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-9090-test.war-
|   |   |                   _test-any-/webapp/WEB-INF/lib/jetty-io-9.0.2.v20130417.jar:/private/var/folders/
|   |   |                   br/kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-9090-test.war-_test-any-
|   |   |                   /webapp/WEB-INF/lib/jetty-servlets-9.0.2.v20130417.jar:/private/var/folders/
|   |   |                   br/kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-9090-test.war-_test-any-
|   |   |                   /webapp/WEB-INF/lib/jetty-util-9.0.2.v20130417.jar:/private/var/folders/br/
|   |   |                   kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-9090-test.war-_test-any-
|   |   |                   /webapp/WEB-INF/lib/websocket-api-9.0.2.v20130417.jar:/private/var/folders/br/
|   |   |                   kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-9090-test.war-_test-any-/webapp/WEB-INF/
|   |   |                   lib/websocket-servlet-9.0.2.v20130417.jar
|   |   |               +> org.eclipse.jetty.server.webapp.ContainerIncludeJarPattern=.*/servlet-
|   |   |                   api-[^/]*.jar$
```

```

| | | +> QoSFilter=org.eclipse.jetty.servlets.QoSFilter@6df3d1f5
| | | +> com.sun.jsp.taglibraryCache={}
| | | +> com.sun.jsp.tagFileJarUrlsCache={}
| | += o.e.j.w.WebAppContext@4ac92718{/proxy,file:/private/var/folders/br/
kbs2g3753c54wmv4j31pnw5r000gn/T/jetty-0.0.0-9090-xref-proxy.war-_xref-proxy-any-
/webapp/,AVAILABLE}{/xref-proxy.war} - STARTED
| | | += org.eclipse.jetty.server.session.SessionHandler@5c25bf03 - STARTED
| | | | += org.eclipse.jetty.server.session.HashSessionManager@33053093 -
STARTED
| | | | += org.eclipse.jetty.security.ConstraintSecurityHandler@3bab0b5a -
STARTED
| | | | +- org.eclipse.jetty.security.DefaultAuthenticatorFactory@11ad5296
| | | | +- org.eclipse.jetty.servlet.ServletHandler@a08feeb - STARTED
| | | | | += default@5c13d641==org.eclipse.jetty.servlet.DefaultServlet,0,true - STARTED
| | | | | | +- maxCacheSize=256000000
| | | | | | +- etags=true
| | | | | | +- dirAllowed=true
| | | | | | +- gzip=true
| | | | | | +- maxCachedFileSize=200000000
| | | | | | +- redirectWelcome=false
| | | | | | +- acceptRanges=true
| | | | | | +- welcomeServlets=false
| | | | | | +- aliases=false
| | | | | | +- useFileMappedBuffer=true
| | | | | | +- maxCachedFiles=2048
| | | | | | +- [/]=>default
| | | | | | += jsp@19c47==org.apache.jasper.servlet.JspServlet,0,true -
STARTED
| | | | | | | +- logVerbosityLevel=DEBUG
| | | | | | | +- fork=false
| | | | | | | +- com.sun.appserv.jsp.classpath=/Users/jesse/
Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-xml-9.0.2.v20130417.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/servlet-
api-3.0.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
http-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
lib/jetty-continuation-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/lib/jetty-server-9.0.2.v20130417.jar:/Users/jesse/Desktop/
jetty-distribution-9.0.2.v20130417/lib/jetty-security-9.0.2.v20130417.jar:/Users/jesse/
Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-servlet-9.0.2.v20130417.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
webapp-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
lib/jetty-deploy-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/lib/jetty-client-9.0.2.v20130417.jar:/Users/jesse/Desktop/
jetty-distribution-9.0.2.v20130417/lib/jetty-jmx-9.0.2.v20130417.jar:/Users/jesse/
Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/com.sun.el-2.2.0.v201303151357.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
javax.el-2.2.0.v201303151357.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
lib/jsp/javax.servlet.jsp.jstl-1.2.0.v201105211821.jar:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/lib/jsp/javax.servlet.jsp-2.2.0.v201112011158.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
org.apache.jasper.glassfish-2.2.2.v201112011158.jar:/Users/
jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
org.apache.taglibs.standard.glassfish-1.2.0.v201112081803.jar:/Users/jesse/Desktop/
jetty-distribution-9.0.2.v20130417/lib/jsp/org.eclipse.jdt.core-3.8.2.v20130121.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/resources:/Users/jesse/Desktop/
jetty-distribution-9.0.2.v20130417/lib/websocket/websocket/websocket-api-9.0.2.v20130417.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/websocket/websocket-
common-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
lib/websocket/websocket-server-9.0.2.v20130417.jar:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/lib/websocket/websocket-servlet-9.0.2.v20130417.jar:/Users/
jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-util-9.0.2.v20130417.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-io-9.0.2.v20130417.jar:/
Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/start.jar:/Library/Java/
JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/Home/jre/lib/ext/dnsns.jar:/Library/
Java/JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/Home/jre/lib/ext/locatedata.jar:/
Library/Java/JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/Home/jre/lib/ext/
sunec.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/Home/jre/lib/
ext/sunjce_provider.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0_17.jdk/Contents/
Home/jre/lib/ext/sunpkcs11.jar:/Library/Java/JavaVirtualMachines/jdk1.7.0_17.jdk/
Contents/Home/jre/lib/ext/zips.jar:/opt/local/lib/libsvnjavahl-1.0.dylib:/System/
Library/Java/Extensions/AppleScriptEngine.jar:/System/Library/Java/Extensions/
dns_sd.jar:/System/Library/Java/Extensions/j3daudio.jar:/System/Library/Java/Extensions/
j3dcore.jar:/System/Library/Java/Extensions/j3dutils.jar:/System/Library/Java/Extensions/
jai_codec.jar:/System/Library/Java/Extensions/jai_core.jar:/System/Library/Java/
Extensions/libAppleScriptEngine.jnilib:/System/Library/Java/Extensions/libJ3D.jnilib:/
System/Library/Java/Extensions/libJ3DAudio.jnilib:/System/Library/Java/Extensions/

```

```

libJ3DUtils.jnilib:/System/Library/Java/Extensions/libmlib_jai.jnilib:/System/Library/
Java/Extensions/libQTJNative.jnilib:/System/Library/Java/Extensions/mlibwrapper_jai.jar:/-
System/Library/Java/Extensions/MRJToolkit.jar:/System/Library/Java/Extensions/
QTJava.zip:/System/Library/Java/Extensions/vecmath.jar:/usr/lib/java/libjdns_sd.jnilib
| | | | | +- scratchdir=/private/var/folders/br/
kbs2g3753c54wmv4j31pnw5r000gn/T/jetty-0.0.0.0-9090-xref-proxy.war-_xref-proxy-any-/jsp
| | | | | +- xpoweredBy=false
| | | | | +- [*.*jsp, *.jspx, *.jspx, *.xsp, *.JSP, *.JSPF, *.JSPX,
*.XSP]=>jsp
| | | | | += XrefTransparentProxy@b0222797==org.eclipse.jetty.proxy.ProxyServlet$Transparent,1,true -
STARTED
| | | | | | +- proxyTo=http://download.eclipse.org/jetty/stable-9
| | | | | | +- hostHeader=download.eclipse.org
| | | | | | +- [/xref/*]=>XrefTransparentProxy
| | | | | += JavadocTransparentProxy@8ab9c012==org.eclipse.jetty.proxy.ProxyServlet
$Transparent,1,true - STARTED
| | | | | | +- proxyTo=http://download.eclipse.org/jetty/stable-9
| | | | | | +- hostHeader=download.eclipse.org
| | | | | | +- [/apidocs/*]=>JavadocTransparentProxy
| | | | | | |~ org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | | | | | |~ org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | | | | | +- HashLoginService[Test Realm] - STARTED
| | | | | | +- org.eclipse.jetty.security.DefaultIdentityService@d2539a6
| | | | | |
| | | | | org.eclipse.jetty.security.authentication.BasicAuthenticator@5497fb72
| | | | | | |
| | | | | | |~ HashLoginService[Test Realm] - STARTED
| | | | | | |~ org.eclipse.jetty.security.DefaultIdentityService@d2539a6
| | | | | |
| | | | | org.eclipse.jetty.security.authentication.BasicAuthenticator@5497fb72
| | | | | | |
| | | | | | |~ []
| | | | | | |~ /={TRACE={RoleInfo,F,C[]}}
| | | | | | |~ org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | | | | | |= org.eclipse.jetty.servlet.ErrorPageErrorHandler@321f8d38 - STARTED
| | | | | | |~ org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | | | | | |~ org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
| | | | | |
| | | | | |+- WebAppClassLoader=Transparent Proxy WebApp@3570713d
| | | | | | |~ file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r000gn/T/jetty-
0.0.0.0-9090-xref-proxy.war-_xref-proxy-any-/webapp/WEB-INF/classes/
| | | | | | |~ file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r000gn/T/jetty-
0.0.0.0-9090-xref-proxy.war-_xref-proxy-any-/webapp/WEB-INF/lib/jetty-
client-9.0.2.v20130417.jar
| | | | | | |~ file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r000gn/T/jetty-
0.0.0.0-9090-xref-proxy.war-_xref-proxy-any-/webapp/WEB-INF/lib/jetty-
http-9.0.2.v20130417.jar
| | | | | | |~ file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r000gn/T/jetty-
0.0.0.0-9090-xref-proxy.war-_xref-proxy-any-/webapp/WEB-INF/lib/jetty-
io-9.0.2.v20130417.jar
| | | | | | |~ file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r000gn/T/jetty-
0.0.0.0-9090-xref-proxy.war-_xref-proxy-any-/webapp/WEB-INF/lib/jetty-
proxy-9.0.2.v20130417.jar
| | | | | | |~ file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r000gn/T/jetty-
0.0.0.0-9090-xref-proxy.war-_xref-proxy-any-/webapp/WEB-INF/lib/jetty-
util-9.0.2.v20130417.jar
| | | | | | |~ startJarLoader@7194b34a
| | | | | | | |~ file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-xml-9.0.2.v20130417.jar
| | | | | | | |~ file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
servlet-api-3.0.jar
| | | | | | | |~ file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-http-9.0.2.v20130417.jar
| | | | | | | |~ file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-continuation-9.0.2.v20130417.jar
| | | | | | | |~ file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-server-9.0.2.v20130417.jar
| | | | | | | |~ file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-security-9.0.2.v20130417.jar
| | | | | | | |~ file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-servlet-9.0.2.v20130417.jar
| | | | | | | |~ file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-webapp-9.0.2.v20130417.jar
| | | | | | | |~ file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-deploy-9.0.2.v20130417.jar

```

```

|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-client-9.0.2.v20130417.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-jmx-9.0.2.v20130417.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/com.sun.el-2.2.0.v201303151357.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/javax.el-2.2.0.v201303151357.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/javax.servlet.jsp.jstl-1.2.0.v201105211821.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/javax.servlet.jsp-2.2.0.v201112011158.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/org.apache.jasper.glassfish-2.2.2.v201112011158.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/org.apache.taglibs.standard.glassfish-1.2.0.v201112081803.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jsp/org.eclipse.jdt.core-3.8.2.v20130121.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
resources/
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
websocket/websocket-api-9.0.2.v20130417.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
websocket/websocket-common-9.0.2.v20130417.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
websocket/websocket-server-9.0.2.v20130417.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
websocket/websocket-servlet-9.0.2.v20130417.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-util-9.0.2.v20130417.jar
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/
jetty-io-9.0.2.v20130417.jar
|   |           +- sun.misc.Launcher$AppClassLoader@19d1b44b
|   |           +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
start.jar
|   |           +- sun.misc.Launcher$ExtClassLoader@1693b52b
|   |           +> javax.servlet.context.tempdir=/private/var/folders/br/
kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-9090-xref-proxy.war-_xref-proxy-any-
|   |           +> org.apache.catalina.jsp_classpath=/private/var/folders/br/
kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-9090-xref-proxy.war-_xref-proxy-
any-/webapp/WEB-INF/classes:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0-9090-xref-proxy.war-_xref-proxy-any-/webapp/WEB-INF/lib/jetty-
client-9.0.2.v20130417.jar:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0-9090-xref-proxy.war-_xref-proxy-any-/webapp/WEB-INF/lib/jetty-
http-9.0.2.v20130417.jar:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0-9090-xref-proxy.war-_xref-proxy-any-/webapp/WEB-INF/lib/jetty-
io-9.0.2.v20130417.jar:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0-9090-xref-proxy.war-_xref-proxy-any-/webapp/WEB-INF/lib/jetty-
proxy-9.0.2.v20130417.jar:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0-9090-xref-proxy.war-_xref-proxy-any-/webapp/WEB-INF/lib/jetty-
util-9.0.2.v20130417.jar
|   |           +> org.eclipse.jetty.server.webapp.ContainerIncludeJarPattern=.*/servlet-
api-[^/]*\.jar$
|   |           +>
JavadocTransparentProxy.HttpClient=org.eclipse.jetty.client.HttpClient@580f016d
|   |           +>
XrefTransparentProxy.HttpClient=org.eclipse.jetty.client.HttpClient@70c7e52b
|   |           +> com.sun.jsp.taglibraryCache={}
|   |           +> com.sun.jsp.tagFileJarUrlsCache={}
|= org.eclipse.jetty.server.handler.DefaultHandler@4de4926a - STARTED
|   |           +- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
|= org.eclipse.jetty.server.handler.RequestLogHandler@3dc087a2 - STARTED
|   |           +- org.eclipse.jetty.server.AsyncNCSARequestLog@108a1cf6 - STARTED
|   |           +- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
|   |           +- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
+- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
|   |           +- [/rego/
* ]=>RegoTest=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=RegoTest,id=0
|   |
|   |           +- org.eclipse.jetty.security.DefaultAuthenticatorFactory@17d41d12=org.eclipse.jetty.security:context=test,
|   |           +- org.eclipse.jetty.server.session.HashSessionManager@1246f8d0=org.eclipse.jetty.server.session:context=tes
|   |           +- org.eclipse.jetty.security.ConstraintSecurityHandler@1890e38=org.eclipse.jetty.security:context=ROOT,type=
|   |           +- WSChat@99274454==com.acme.WebSocketChatServlet,1,true=org.eclipse.jetty.servlet:context=test,type=servle

```

```

|   ++
| org.eclipse.jetty.deploy.DeploymentManager@c8e4be2=org.eclipse.jetty.deploy:type=deploymentmanager,id=0
|   ++
| org.eclipse.jetty.jmx.MBeanContainer@644a5ddd=org.eclipse.jetty.jmx:type=mbeancontainer,id=0
|   +- [/dump/gzip/*, *.txt]
[ ]==0=>GzipFilter=org.eclipse.jetty.servlet:context=test,type=filtermapping,name=GzipFilter,id=0
|   ++
Hello@42628b2==com.acme.HelloWorld,1,true=org.eclipse.jetty.servlet:context=test,type=servletholder,name=
|   +- [/]=>default=org.eclipse.jetty.servlet:context=xref-
proxy,type=servletmapping,name=default,id=0
|   +- [/login/
* ]=>Login=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=Login,id=0
|   ++
org.eclipse.jetty.server.handler.DefaultHandler@4de4926a=org.eclipse.jetty.server.handler:type=defaulthan-
|   ++
org.eclipse.jetty.server.session.SessionHandler@5c25bf03=org.eclipse.jetty.server.session:context=xref-
proxy,type=sessionhandler,id=0
|   +- [/ws/
* ]=>WSChat=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=WSChat,id=0
|   +- o.e.j.w.WebAppContext@6f01ba6f{/file:/Users/jesse/Desktop/
jetty-distribution-9.0.2.v20130417/webapps/ROOT/,AVAILABLE}{{/
ROOT}=org.eclipse.jetty.webapp:context=ROOT,type=webappcontext,id=0
|   +- o.e.j.w.WebAppContext@7ea88b1c{/async-rest,[file:/private/var/folders/br/
kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0.0-9090-async-rest.war-_async-rest-
any-/webapp/,jar:file:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/
T/jetty-0.0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/example-
async-rest-jar-9.0.2.v20130417.jar!/META-INF/resources/],AVAILABLE}{/async-
rest.war}=org.eclipse.jetty.webapp:context=async-rest,type=webappcontext,id=0
|   +- ServerConnector@3d0f282{HTTP/1.1}
{0.0.0.0:9090}=org.eclipse.jetty.server:context=HTTP/1.1@3d0f282,type=serverconnector,id=0
|   ++
org.eclipse.jetty.security.DefaultAuthenticatorFactory@6242c657=org.eclipse.jetty.security:context=ROOT,t
|   +- JavadocTransparentProxy@8ab9c012==org.eclipse.jetty.proxy.ProxyServlet
$Transparent,1,true=org.eclipse.jetty.servlet:context=xref-
proxy,type=servletholder,name=JavadocTransparentProxy,id=0
|   +- [/dump/*
*.dump]=>Dump=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=Dump,id=0
|   +- [/jsp/
foo/]=>foo.jsp=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=foo.jsp,id=0
|   ++
org.eclipse.jetty.servlet.ServletHandler@46bac287=org.eclipse.jetty.servlet:context=async-
rest,type=servlethandler,id=0
|   ++
GzipFilter=org.eclipse.jetty.servlet:context=test,type=filterholder,name=GzipFilter,id=0
|   +- o.e.j.w.WebAppContext@4ac92718{/proxy,file:/private/var/folders/br/
kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0.0-9090-xref-proxy.war-_xref-proxy-
any-/webapp/,AVAILABLE}{/xref-proxy.war}=org.eclipse.jetty.webapp:context=xref-
proxy,type=webappcontext,id=0
|   ++
qtp1062680061{STARTED,10<=13<=200,i=1,q=0}=org.eclipse.jetty.util.thread:type=queuedthreadpool,id=0
|   ++
org.eclipse.jetty.server.session.HashSessionManager@33053093=org.eclipse.jetty.server.session:context=xre
proxy,type=hashsessionmanager,id=0
|   ++
org.eclipse.jetty.security.DefaultAuthenticatorFactory@52b12fef=org.eclipse.jetty.security:context=async-
rest,type=defaultauthenticatorfactory,id=0
|   ++
Login@462ff49==com.acme.LoginServlet,1,true=org.eclipse.jetty.servlet:context=test,type=servletholder,nam
|   ++
org.eclipse.jetty.security.authentication.BasicAuthenticator@7b239469=org.eclipse.jetty.security.authenti
rest,type=basicauthenticator,id=0
|   ++
MultiPart=org.eclipse.jetty.servlet:context=test,type=filterholder,name=MultiPart,id=0
|   ++
default@5c13d641==org.eclipse.jetty.servlet.DefaultServlet,0,true=org.eclipse.jetty.servlet:context=xref-
proxy,type=servletholder,name=default,id=0
|   ++
default@5c13d641==org.eclipse.jetty.servlet.DefaultServlet,0,true=org.eclipse.jetty.servlet:context=async-
rest,type=servletholder,name=default,id=0
|   ++
default@5c13d641==org.eclipse.jetty.servlet.DefaultServlet,0,true=org.eclipse.jetty.servlet:context=test,
|   ++
org.eclipse.jetty.server.session.HashSessionManager@746a95ae=org.eclipse.jetty.server.session:context=ROO
|   ++
RegoTest2@849d6425==com.acme.RegTest,-1,false=org.eclipse.jetty.servlet:context=test,type=servletholder,n

```

```

|   +- org.eclipse.jetty.server.ServerConnector
$ServerConnectorManager@6f0ac4be=org.eclipse.jetty.server:context=HTTP/1.1@3d0f282,type=serverconnector
$serverconnectormanager,id=0
|   +- [ / ]=>default=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=default,id=0
|   +- SecureMode@d45951da==com.acme.SecureModeServlet,1,true=org.eclipse.jetty.servlet:context=test,type=servle
|   +- org.eclipse.jetty.security.authentication.BasicAuthenticator@6b733b94=org.eclipse.jetty.security.authenti
|   +- org.eclipse.jetty.server.session.SessionHandler@6dfb8d2e=org.eclipse.jetty.server.session:context=async-
rest,type=sessionhandler,id=0
|   +- org.eclipse.jetty.security.DefaultIdentityService@41917d6d=org.eclipse.jetty.security:context=test,type=c
|   +- jsp@19c47==org.apache.jasper.servlet.JspServlet,0,true=org.eclipse.jetty.servlet:context=xref-
proxy,type=servletholder,name=jsp,id=0
|   +- jsp@19c47==org.apache.jasper.servlet.JspServlet,0,true=org.eclipse.jetty.servlet:context=ROOT,type=servle
|   +- jsp@19c47==org.apache.jasper.servlet.JspServlet,0,true=org.eclipse.jetty.servlet:context=async-
rest,type=servletholder,name=jsp,id=0
|   +- jsp@19c47==org.apache.jasper.servlet.JspServlet,0,true=org.eclipse.jetty.servlet:context=test,type=servle
|   +- [ /* ]/
[ ]==31=>TestFilter=org.eclipse.jetty.servlet:context=test,type=filtermapping,name=TestFilter,id=0
|   +- org.eclipse.jetty.server.session.HashSessionManager@6cb83869=org.eclipse.jetty.server.session:context=asy
rest,type=hashsessionmanager,id=0
|   +- org.eclipse.jetty.io.ArrayByteBufferPool@30ad8942=org.eclipse.jetty.io:context=HTTP/1.1@3d0f282,type=arr
|   +- [ /cgi-bin/
* ]=>CGI=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=CGI,id=0
|   +- org.eclipse.jetty.server.handler.HandlerCollection@58b37561=org.eclipse.jetty.server.handler:type=handler
|   +- Session@d9891a76==com.acme.SessionDump,5,true=org.eclipse.jetty.servlet:context=test,type=servletholder,n
|   +- org.eclipse.jetty.servlet.ServletHandler@a08feeb=org.eclipse.jetty.servlet:context=xref-
proxy,type=servlethandler,id=0
|   +- org.eclipse.util.thread.ScheduledExecutorScheduler@725f5=org.eclipse.util.thread:type=schedul
|   +- [ /* ]/
[ ]==0=>QoSFilter=org.eclipse.jetty.servlet:context=test,type=filtermapping,name=QoSFilter,id=0
|   +- org.eclipse.server.session.SessionHandler@5a770658=org.eclipse.jetty.server.session:context=ROOT,ty
|   +- org.eclipse.server.session.SessionHandler@336abd81=org.eclipse.jetty.server.session:context=test,ty
|   +- o.e.j.s.h.ContextHandler@7b2dffdf{/javadoc,file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/
javadoc,AVAILABLE}=org.eclipse.jetty.server.handler:context=javadoc,type=contexthandler,id=0
|   +- org.eclipse.servlets.QoSFilter@6df3d1f5=org.eclipse.servlets:context=test,type=qosfilter,id=0
|   +- [* .more]=>Dump=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=Dump,id=1
|   +- Dump@20ae14==com.acme.Dump,1,true=org.eclipse.jetty.servlet:context=test,type=servletholder,name=Dump,id=
|   +- HttpConnectionFactory@5e47b1b9{HTTP/1.1}=org.eclipse.jetty.server:context=HTTP/1.1@3d0f282,type=httpconn
|   +- org.eclipse.jetty.servlet.ServletHandler@debac27=org.eclipse.jetty.servlet:context=ROOT,type=servlethand
|   +- [ *.jsp, *.jspx, *.jspx, *.xsp, *.JSP, *.JSPF,
*.JSPX, *.XSP]=>jsp=org.eclipse.jetty.servlet:context=xref-
proxy,type=servletmapping,name=jsp,id=0
|   +- org.eclipse.jetty.server.handler.MovedContextHandler
$Redirector@2a4200d3=org.eclipse.jetty.server.handler:context=oldContextPath,type=movedcontexthandler
$redirector,id=0
|   +- TestFilter=org.eclipse.jetty.servlet:context=test,type=filterholder,name=TestFilter,id=0
|   +- Rewrite@a4dac96c==com.acme.RewriteServlet,-1,false=org.eclipse.jetty.servlet:context=test,type=servlethol
|   +- [ /dispatch/
* ]=>Dispatch=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=Dispatch,id=0
|   +- [ /testSerial ]=>SerialRestServlet=org.eclipse.jetty.servlet:context=async-
rest,type=servletmapping,name=SerialRestServlet,id=0
|   +- org.eclipse.jetty.servlet.ErrorPageErrorHandler@24bf7a86=org.eclipse.jetty.servlet:context=test,type=err

```

```

|   +- [ /secureMode/
* ]=>SecureMode=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=SecureMode,id=0
|   +- [ / ]=>default=org.eclipse.jetty.servlet:context=async-
rest,type=servletmapping,name=default,id=0
|   +-+
Dispatch@14d3a89a==com.acme.DispatchServlet,1,true=org.eclipse.jetty.servlet:context=test,type=servlethol
|   +-+
org.eclipse.jetty.server.handler.ContextHandlerCollection@64c6e290=org.eclipse.jetty.server.handler:type=
|   +-+
org.eclipse.jetty.security.ConstraintSecurityHandler@2848c90e=org.eclipse.jetty.security:context=async-
rest,type=constraintsecurityhandler,id=0
|   +-+
[ /rego2/
* ]=>RegoTest2=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=RegoTest2,id=0
|   +-+
[ /rewritten/*, /redirected/
* ]=>Rewrite=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=Rewrite,id=0
|   +-+
org.eclipse.jetty.servlet.ServletHandler@5034037e=org.eclipse.jetty.servlet:context=test,type=servlethand
|   +-+
org.eclipse.jetty.servlet.ErrorPageErrorHandler@3c121009=org.eclipse.jetty.servlet:context=async-
rest,type=errorpageerrorhandler,id=0
|   +-+
sun.nio.ch.ServerSocketChannelImpl[/0:0:0:0:0:0:0:0:9090]=sun.nio.ch:context=HTTP/1.1@3d0f282,type=server
|   +-+
org.eclipse.jetty.security.ConstraintSecurityHandler@7179290f=org.eclipse.jetty.security:context=test,type=
|   +-+
org.eclipse.jetty.server.session.HashSessionIdManager@289eb857=org.eclipse.jetty.server.session:type=hash
|   +-+
org.eclipse.jetty.security.authentication.BasicAuthenticator@5497fb72=org.eclipse.jetty.security.authenti
proxy,type=basicauthenticator,id=0
|   +-+
org.eclipse.jetty.security.DefaultAuthenticatorFactory@11ad5296=org.eclipse.jetty.security:context=xref-
proxy,type=defaultauthenticatorfactory,id=0
|   +-+
[ /dump/*/
[]==0=>MultiPart=org.eclipse.jetty.servlet:context=test,type=filtermapping,name=MultiPart,id=0
|   +-+
o.e.j.s.h.MovedContextHandler@5e0c8d24{
oldContextPath,null,AVAILABLE}=org.eclipse.jetty.server.handler:context=oldContextPath,type=movedcontextha
|   +-+
QoSFilter=org.eclipse.jetty.servlet:context=test,type=filterholder,name=QoSFilter,id=0
|   +-+
org.eclipse.jetty.security.authentication.FormAuthenticator@1fa291f2=org.eclipse.jetty.security.authenti
|   +-+
o.e.j.w.WebAppContext@716d9094{/test,file:/
private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/T/
jetty-0.0.0-9090-test.war-_test-any-/webapp/,AVAILABLE}
test.war=org.eclipse.jetty.webapp:context=test,type=webappcontext,id=0
|   +-+
[ / ]=>default=org.eclipse.jetty.servlet:context=ROOT,type=servletmapping,name=default,id=0
|   +-+
[ /hello/
* ]=>Hello=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=Hello,id=0
|   +-+
[ /chat/
* ]=>Chat=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=Chat,id=0
|   +-+
[ /testAsync]=>AsyncRestServlet=org.eclipse.jetty.servlet:context=async-
rest,type=servletmapping,name=AsyncRestServlet,id=0
|   +-+
org.eclipse.jetty.security.DefaultIdentityService@d2539a6=org.eclipse.jetty.security:context=async-
rest,type=defaultidentityservice,id=0
|   +-+
org.eclipse.jetty.server.handler.RequestLogHandler@3dc087a2=org.eclipse.jetty.server.handler:type=request-
log
|   +-+
org.eclipse.jetty.servlet.ErrorPageErrorHandler@321f8d38=org.eclipse.jetty.servlet:context=xref-
proxy,type=errorpageerrorhandler,id=0
|   +-+
org.eclipse.jetty.server.handler.ResourceHandler@8f9c8a7=org.eclipse.jetty.server.handler:context=javadoc
|   +-+
CGI@10465==org.eclipse.jetty.servlets.CGI,1,true=org.eclipse.jetty.servlet:context=test,type=servletholde
|   +-+
SerialRestServlet@461411d==org.eclipse.jetty.example.asyncrest.SerialRestServlet,-1,false=org.eclipse.jett
rest,type=servletholder,name=SerialRestServlet,id=0
|   +-+
HashLoginService[Test
Realm]=org.eclipse.jetty.security:type=hashloginservice,id=0
|   +-+
AsyncRestServlet@73eb9bd5==org.eclipse.jetty.example.asyncrest.AsyncRestServlet,-1,false=org.eclipse.jett
rest,type=servletholder,name=AsyncRestServlet,id=0
|   +-+
org.eclipse.jetty.server.Server@76f08fe1=org.eclipse.jetty.server:type=server,id=0
|   +-+
org.eclipse.jetty.servlet.ErrorPageErrorHandler@3c41a9ce=org.eclipse.jetty.servlet:context=ROOT,type=err

```

```

|   +- [/apidocs/*]=>JavadocTransparentProxy=org.eclipse.jetty.servlet:context=xref-
proxy,type=servletmapping,name=JavadocTransparentProxy,id=0
|   +- Chat@200778==com.acme.ChatServlet,1,true=org.eclipse.jetty.servlet:context=test,type=servletholder,name=C
|   +- [/cookie/
* ]=>Cookie=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=Cookie,id=0
|   +- [/session/
* ]=>Session=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=Session,id=0
|   +- org.eclipse.jetty.deploy.providers.WebAppProvider@7b26b7df=org.eclipse.jetty.deploy.providers:type=webapp
|   +- org.eclipse.jetty.server.AsyncNCSARequestLog@108a1cf6=org.eclipse.jetty.server:type=asyncncksarequestlog,i
|   +- [* .jsp, *.jspx, *.xsp, *.JSP, *.JSPF, *.JSXP,
*.XSP]=>jsp=org.eclipse.jetty.servlet:context=test,type=servletmapping,name=jsp,id=0
|   +- [* .jsp, *.jspx, *.xsp, *.JSP, *.JSPF, *.JSXP,
*.XSP]=>jsp=org.eclipse.jetty.servlet:context=ROOT,type=servletmapping,name=jsp,id=0
|   +- HashLoginService[Test
Realm]=org.eclipse.jetty.security:context=test,type=hashloginservice,id=0
|   +- [* .jsp, *.jspx, *.xsp, *.JSP, *.JSPF,
*.JSXP, *.XSP]=>jsp=org.eclipse.jetty.servlet:context=async-
rest,type=servletmapping,name=jsp,id=0
|   +- foo.jsp@d7583f1f==org.apache.jasper.servlet.JspServlet,-1,false=org.eclipse.jetty.servlet:context=test,ty
|   +- RegoTest@dafcd1ad==com.acme.RegoTest,-1,false=org.eclipse.jetty.servlet:context=test,type=servletholder,n
|   +- [/xref/*]=>XrefTransparentProxy=org.eclipse.jetty.servlet:context=xref-
proxy,type=servletmapping,name=XrefTransparentProxy,id=0
|   +- org.eclipse.jetty.security.ConstraintSecurityHandler@3bab0b5a=org.eclipse.jetty.security:context=xref-
proxy,type=constraintsecurityhandler,id=0
|   +- HttpConfiguration@703b16bb{32768,8192/8192,https://:8443,
[]}=org.eclipse.jetty.server:context=HTTP/1.1@3d0f282,type=httpconfiguration,id=0
|   +- org.eclipse.jetty.util.log.Log@dda4f7b=org.eclipse.jetty.util.log:type=log,id=0
|   +- Cookie@78a4f684==com.acme.CookieDump,1,true=org.eclipse.jetty.servlet:context=test,type=servletholder,n
|   +- XrefTransparentProxy@b0222797==org.eclipse.jetty.proxy.ProxyServlet
$Transparent,1,true=org.eclipse.jetty.servlet:context=xref-
proxy,type=servletholder,name=XrefTransparentProxy,id=0
+- org.eclipse.jetty.util.log.Log@dda4f7b
+= ServerConnector@3d0f282{HTTP/1.1}{0.0.0.0:9090} - STARTED
|   +- org.eclipse.jetty.server.Server@76f08fe1 - STARTING
|   +- qtp1062680061{STARTED,10<13<=200,i=1,q=0} - STARTED
|   +- org.eclipse.jetty.util.thread.ScheduledExecutorScheduler@725f5 - STARTED
|   +- org.eclipse.jetty.io.ArrayByteBufferPool@30ad8942
|   +- HttpConnectionFactory@5e47b1b9{HTTP/1.1} - STARTED
|   |   +- HttpConfiguration@703b16bb{32768,8192/8192,https://:8443,[]}
|   |   +- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
|   +- org.eclipse.jetty.server.ServerConnector$ServerConnectorManager@6f0ac4be -
STARTED
|   |   +- org.eclipse.jetty.io.SelectorManager$ManagedSelector@61454787 keys=0
selected=0 id=0
|   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   |   +- sun.nio.ch.KQueueSelectorImpl@a0c508b keys=0
|   |   +- org.eclipse.jetty.io.SelectorManager$ManagedSelector@2e7bdad4 keys=0
selected=0 id=1
|   |   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   |   +- sun.nio.ch.KQueueSelectorImpl@5825168 keys=0
|   |   +- org.eclipse.jetty.io.SelectorManager$ManagedSelector@2ea85ab keys=0
selected=0 id=2
|   |   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   |   +- sun.nio.ch.KQueueSelectorImpl@6faa85f6 keys=0
|   |   +- org.eclipse.jetty.io.SelectorManager$ManagedSelector@244112c0 keys=0
selected=0 id=3
|   |   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   |   +- sun.nio.ch.KQueueSelectorImpl@10c6f695 keys=0
|   |   +- org.eclipse.jetty.io.SelectorManager$ManagedSelector@7666b8cd keys=0
selected=0 id=4
|   |   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   |   +- sun.nio.ch.KQueueSelectorImpl@17836c59 keys=0
|   |   +- org.eclipse.jetty.io.SelectorManager$ManagedSelector@353e531e keys=0
selected=0 id=5

```

```

|   |   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   |   +- sun.nio.ch.KQueueSelectorImpl@2095f259 keys=0
|   |   |   +- org.eclipse.jetty.io.SelectorManager$ManagedSelector@5459c1c5 keys=0
selected=0 id=6
|   |   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   |   +- sun.nio.ch.KQueueSelectorImpl@142c7195 keys=0
|   |   |   +- org.eclipse.jetty.io.SelectorManager$ManagedSelector@71d4f78b keys=0
selected=0 id=7
|   |   |   +- org.eclipse.jetty.io.SelectorManager
$ManagedSelector.select(SelectorManager.java:459)
|   |   |   +- sun.nio.ch.KQueueSelectorImpl@16bdab45 keys=0
|   |   +- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
|   |   +- sun.nio.ch.ServerSocketChannelImpl[/0:0:0:0:0:0:0:0:9090]
+= org.eclipse.jetty.deploy.DeploymentManager@c8e4be2 - STARTED
|   +- org.eclipse.jetty.deploy.providers.WebAppProvider@7b26b7df - STARTED
|   +- org.eclipse.jetty.jmx.MBeanContainer@644a5ddd
+- HashLoginService[Test Realm] - STARTED
+- org.eclipse.jetty.server.session.HashSessionIdManager@289eb857 - STARTED
|
+> startJarLoader@7194b34a
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
xml-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/servlet-
api-3.0.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
http-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
continuation-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
server-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
security-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
servlet-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
webapp-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
deploy-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
client-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
jmx-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
com.sun.el-2.2.0.v201303151357.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
javax.el-2.2.0.v201303151357.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
javax.servlet.jsp.jstl-1.2.0.v201105211821.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
javax.servlet.jsp-2.2.0.v201112011158.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
org.apache.jasper.glassfish-2.2.2.v201112011158.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
org.apache.taglibs.standard.glassfish-1.2.0.v201112081803.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jsp/
org.eclipse.jdt.core-3.8.2.v20130121.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/resources/
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/websocket/
websocket-api-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/websocket/
websocket-common-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/websocket/
websocket-server-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/websocket/
websocket-servlet-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
util-9.0.2.v20130417.jar
    +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/lib/jetty-
io-9.0.2.v20130417.jar
    +- sun.misc.Launcher$AppClassLoader@19d1b44b
        +- file:/Users/jesse/Desktop/jetty-distribution-9.0.2.v20130417/start.jar
        +- sun.misc.Launcher$ExtClassLoader@1693b52b
2013-04-29 14:38:39.422:INFO:oejs.Server:Thread-2: Graceful shutdown
org.eclipse.jetty.server.Server@76f08fe1 by Mon Apr 29 14:38:44 CDT 2013

```

```
2013-04-29 14:38:39.429:INFO:oejs.ServerConnector:Thread-2: Stopped
    ServerConnector@3d0f282{HTTP/1.1}{0.0.0.0:9090}
2013-04-29 14:38:39.444:INFO:oejs1.ELContextCleaner:Thread-2: javax.el.BeanELResolver
    purged
2013-04-29 14:38:39.444:INFO:oejsh.ContextHandler:Thread-2: stopped
    o.e.j.w.WebAppContext@4ac92718{/proxy,file:/private/var/folders/br/
kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0.0-9090-xref-proxy.war-_xref-proxy-any-
/webapp/,UNAVAILABLE}{/xref-proxy.war}
2013-04-29 14:38:39.447:INFO:oejs1.ELContextCleaner:Thread-2: javax.el.BeanELResolver
    purged
2013-04-29 14:38:39.447:INFO:oejsh.ContextHandler:Thread-2: stopped
    o.e.j.w.WebAppContext@716d9094{/test,file:/private/var/folders/br/
kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0.0-9090-test.war-_test-any-
/webapp/,UNAVAILABLE}{/test.war}
2013-04-29 14:38:39.455:INFO:oejsh.ContextHandler:Thread-2: stopped
    o.e.j.s.h.ContextHandler@7b2dffdf{/javadoc,file:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/javadoc,UNAVAILABLE}
2013-04-29 14:38:39.456:INFO:oejs1.ELContextCleaner:Thread-2: javax.el.BeanELResolver
    purged
2013-04-29 14:38:39.456:INFO:oejsh.ContextHandler:Thread-2: stopped
    o.e.j.w.WebAppContext@6f01ba6f{/,file:/Users/jesse/Desktop/jetty-
distribution-9.0.2.v20130417/webapps/ROOT/,UNAVAILABLE}{/ROOT}
2013-04-29 14:38:39.456:INFO:oejsh.ContextHandler:Thread-2: stopped
    o.e.j.s.h.MovedContextHandler@5e0c8d24{/oldContextPath,null,UNAVAILABLE}
2013-04-29 14:38:39.457:INFO:oejs1.ELContextCleaner:Thread-2: javax.el.BeanELResolver
    purged
2013-04-29 14:38:39.457:INFO:oejsh.ContextHandler:Thread-2: stopped
    o.e.j.w.WebAppContext@7ea88b1c{/async-rest,[file:/private/var/folders/br/
kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0.0-9090-async-rest.war-_async-rest-
any-/webapp/, jar:/private/var/folders/br/kbs2g3753c54wmv4j31pnw5r0000gn/T/
jetty-0.0.0.0-9090-async-rest.war-_async-rest-any-/webapp/WEB-INF/lib/example-async-rest-
jar-9.0.2.v20130417.jar!/_META-INF/resources/],UNAVAILABLE}{/async-rest.war}
```

Part IV. Jetty Development Guide

Table of Contents

23. Maven and Jetty	285
Using Maven	285
Configuring the Jetty Maven Plugin	289
Files Scanned by the Jetty Maven Plugin	307
Jetty Jspc Maven Plugin	307
24. Using Ant with Jetty	312
Using the Ant Jetty Plugin	312
25. Handlers	322
Rewrite Handler	322
Writing Custom Handlers	324
26. Embedding	328
Jetty Embedded HelloWorld	328
Embedding Jetty	329
Embedded Examples	334
27. Debugging	339
Options	339
Enable remote debugging	339
Debugging With Eclipse	340
Debugging With IntelliJ	342
28. Frameworks	347
Spring Setup	347
OSGI	348
Weld	362
Metro	364
29. HTTP Client	366
Introduction	366
API Usage	367
Other Features	373
30. WebSocket Introduction	376
What Jetty provides	376
WebSocket APIs	377
31. Jetty Websocket API	378
Jetty WebSocket API Usage	378
WebSocket Events	378
WebSocket Session	379
Send Messages to Remote Endpoint	379
Using WebSocket Annotations	384
Using WebSocketListener	385
Using the WebSocketAdapter	386
Jetty WebSocket Server API	387
Jetty WebSocket Client API	389
32. Java Websocket API	391
Java WebSocket Client API Usage	391
Java WebSocket Server API	391

Chapter 23. Maven and Jetty

Table of Contents

Using Maven	285
Configuring the Jetty Maven Plugin	289
Files Scanned by the Jetty Maven Plugin	307
Jetty Jspc Maven Plugin	307

This chapter explains how to use Jetty with Maven and the Jetty Maven plugin.

Using Maven

[Apache Maven](#) is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

It is an ideal tool to build a web application project, and such projects can use the [jetty-maven-plugin](#) to easily run the web application and save time in development. You can also use Maven to build, test and run a project which embeds Jetty.

First we'll have a look at a very simple HelloWorld java application that embeds Jetty, then a simple webapp which makes use of the [jetty-maven-plugin](#) to speed up the development cycle.

Using Embedded Jetty with Maven

To understand the basic operations of building and running against Jetty, first review

- [embedding with Jetty](#)
- [Jetty HelloWorld example](#)

Maven uses convention over configuration, so it is best to use the project structure Maven recommends. You can use [archetypes](#) to quickly setup Maven projects, but we will set up the structure manually for this simple tutorial example:

```
> mkdir JettyMavenHelloWorld  
> cd JettyMavenHelloWorld  
> mkdir -p src/main/java/org/example
```

Creating the HelloWorld Class

Use an editor to create the file `src/main/java/org/example/HelloWorld.java` with the following contents:

```
package org.example;  
  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import javax.servlet.ServletException;  
import java.io.IOException;  
import org.eclipse.jetty.server.Server;  
import org.eclipse.jetty.server.Request;  
import org.eclipse.jetty.server.handler.AbstractHandler;  
  
public class HelloWorld extends AbstractHandler  
{  
    public void handle(String target,  
                      Request baseRequest,  
                      HttpServletRequest request,
```

```
        HttpServletResponse response)
    throws IOException, ServletException
{
    response.setContentType("text/html;charset=utf-8");
    response.setStatus(HttpServletResponse.SC_OK);
    baseRequest.setHandled(true);
    response.getWriter().println("<h1>Hello World</h1>");
}

public static void main(String[] args) throws Exception
{
    Server server = new Server(8080);
    server.setHandler(new HelloWorld());

    server.start();
    server.join();
}
}
```

Creating the POM Descriptor

The pom.xml file declares the project name and its dependencies. Use an editor to create the file pom.xml with the following contents:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>org.example</groupId>
    <artifactId>hello-world</artifactId>
    <version>0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>Jetty HelloWorld</name>

    <properties>
        <!-- Adapt this to a version found on
            http://central.maven.org/maven2/org/eclipse/jetty/jetty-maven-plugin/
        -->
        <jettyVersion>9.0.2.v20130417</jettyVersion>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-server</artifactId>
            <version>${jettyVersion}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>1.1</version>
                <executions>
                    <execution><goals><goal>java</goal></goals></execution>
                </executions>
                <configuration>
                    <mainClass>org.example.HelloWorld</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Building and Running Embedded HelloWorld

You can now compile and execute the HelloWorld class by using these commands:

```
> mvn clean compile exec:java
```

You can point your browser to `http://localhost:8080` to see the hello world page. You can observe what Maven is doing for you behind the scenes by using the `mvn dependency:tree` command, which reveals the transitive dependency resolved and downloaded as:

```
> mvn dependency:tree
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] -----
[INFO] Building Jetty Hello World
[INFO]   task-segment: [dependency:tree]
[INFO] -----
[INFO] [dependency:tree {execution: default-cli}]
[INFO] org.example:hello-world:jar:0.1-SNAPSHOT
[INFO] \- org.eclipse.jetty:jetty-server:jar:9.0.0:compile
[INFO]     +- org.eclipse.jetty:javax.servlet:jar:3.0.0.v201112011016:compile
[INFO]     +- org.eclipse.jetty:jetty-continuation:jar:9.0.0:compile
[INFO]     \- org.eclipse.jetty:jetty-http:jar:9.0.0:compile
[INFO]         \- org.eclipse.jetty:jetty-io:jar:9.0.0:compile
[INFO]             \- org.eclipse.jetty:jetty-util:jar:9.0.0:compile
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 4 seconds
[INFO] Finished at: Thu Jan 24 16:19:08 EST 2013
[INFO] Final Memory: 11M/68M
[INFO] -----
```

Developing a Standard WebApp with Jetty and Maven

The previous section demonstrated how to use Maven with an application that embeds Jetty. Now we will examine instead how to develop a standard webapp with Maven and Jetty. First create the Maven structure (you can use the [maven webapp archetype](#) instead if you prefer):

```
> mkdir JettyMavenHelloWarApp
> cd JettyMavenHelloWebApp
> mkdir -p src/main/java/org/example
> mkdir -p src/main/webapp/WEB-INF
```

Creating a Servlet

Use an editor to create the file `src/main/java/org/example/HelloServlet.java` with the following contents:

```
package org.example;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
    {
        response.setContentType("text/html");
        response.setStatus(HttpServletResponse.SC_OK);
        response.getWriter().println("<h1>Hello Servlet</h1>");
        response.getWriter().println("session=" + request.getSession(true).getId());
    }
}
```

You need to declare this servlet in the deployment descriptor, so edit the file `src/main/webapp/WEB-INF/web.xml` and add the following contents:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app
    xmlns="http://java.sun.com/xml/ns/javaee"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-app_3_0.xsd"
metadata-complete="false"
version="3.0">

<servlet>
  <servlet-name>Hello</servlet-name>
  <servlet-class>org.example.HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Hello</servlet-name>
  <url-pattern>/hello/*</url-pattern>
</servlet-mapping>

</web-app>
```

Creating the POM Descriptor

The pom.xml file declares the project name and its dependencies. Use an editor to create the file pom.xml with the following contents, noting particularly the declaration of the [jetty-maven-plugin](#):

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>org.example</groupId>
  <artifactId>hello-world</artifactId>
  <version>0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <name>Jetty HelloWorld WebApp</name>

  <properties>
    <jettyVersion>9.0.2.v20130417</jettyVersion> /// Adapt this to a version found on
    http://repo.maven.apache.org/maven2/org/eclipse/jetty/jetty-maven-plugin/
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.eclipse.jetty.orbit</groupId>
      <artifactId>javax.servlet</artifactId>
      <version>3.0.0.v201112011016</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>${jettyVersion}</version>
      </plugin>
    </plugins>
  </build>
</project>
```

Building and Running the Web Application

Now you can both build and run the web application without needing to assemble it into a war by using the [jetty-maven-plugin](#) via the command:

```
> mvn jetty:run
```

You can see the static and dynamic content at <http://localhost:8080/hello>

There are a great deal of configuration options available for the jetty-maven-plugin to help you build and run your webapp. The full reference is at [Configuring the Jetty Maven Plugin](#).

Building a WAR file

You can create a Web Application Archive (WAR) file from the project with the command:

```
> mvn package
```

The resulting war file is in the `target` directory and may be deployed on any standard servlet server, including [Jetty](#).

Configuring the Jetty Maven Plugin

The Jetty Maven plugin is useful for rapid development and testing. You can add it to any webapp project that is structured according to the usual Maven defaults. The plugin can then periodically scan your project for changes and automatically redeploy the webapp if any are found. This makes the development cycle more productive by eliminating the build and deploy steps: you use your IDE to make changes to the project, and the running web container automatically picks them up, allowing you to test them straight away.



Important

You need to use Maven 3 and Java 1.7 for this plugin.

Quick Start: Get Up and Running

First, add `jetty-maven-plugin` to your `pom.xml` definition:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>9.2.1.v20140609</version>
</plugin>
```

Then, from the same directory as your root `pom.xml`, type:

```
mvn jetty:run
```

This starts Jetty and serves up your project on `http://localhost:8080/`.

Jetty continues to run until you stop it. While it runs, it periodically scans for changes to your project files, so if you save changes and recompile your class files, Jetty redeploys your webapp, and you can instantly test the changes you just made.

You can terminate the plugin with a `ctrl-c` in the terminal window where it is running.

Supported Goals

The Jetty Maven plugin has a number of distinct Maven goals. Arguably the most useful is the `run` goal that we saw in the Quick Start section which runs Jetty on your unassembled webapp. There are other goals which help you accomplish different tasks. For example, you might need to run your webapp in a forked instance of Jetty, rather than within the process running Maven; or you may need finer grained control over the maven lifecycle stage in which you wish to deploy your webapp. There are different goals to accomplish these tasks, as well as several others.

To see a list of all goals supported by the Jetty Maven plugin, do:

```
mvn jetty:help
```

To see the detailed list of parameters that can be configured for a particular goal, in addition to its description, do:

```
mvn jetty:help -Ddetail=true -Dgoal= goal-name
```

Configuring the Jetty Container

These configuration elements set up the Jetty environment in which your webapp executes. They are common to most goals:

httpConnector

Optional. If not specified, Jetty will create a [ServerConnector](#) instance listening on port 8080. You can change this default port number by using the system property `jetty.port` on the command line, for example, `mvn -Djetty.port=9999 jetty:run`. Alternatively, you can use this configuration element to set up the information for the ServerConnector. The following are the valid configuration sub-elements:

port

The port number for the connector to listen on. By default it is 8080.

host

The particular interface for the connector to listen on. By default, all interfaces.

name

The name of the connector, which is useful for [configuring contexts to respond only on particular connectors](#).

idleTimeout

Maximum idle time for a connection.

soLinger

The socket linger time.

You could instead configure the connectors in a standard [jetty xml config file](#) and put its location into the `jettyXml` parameter. Note that since jetty-9.0 it is no longer possible to configure a [https connector](#) directly in the pom.xml: you need to [use jetty xml config files to do it](#).

jettyXml

Optional. A comma separated list of locations of `jetty.xml` files to apply in addition to any plugin configuration parameters. You might use it if you have other webapps, handlers, specific types of connectors etc., to deploy, or if you have other Jetty objects that you cannot configure from the plugin.

scanIntervalSeconds

The pause in seconds between sweeps of the webapp to check for changes and automatically hot redeploy if any are detected. **By default this is 0, which disables hot deployment scanning.** A number greater than 0 enables it.

reload

Default value is "automatic", used in conjunction with a non-zero `scanIntervalSeconds` causes automatic hot redeploy when changes are detected. Set to "manual" instead to trigger scanning by typing a linefeed in the console running the plugin. This might be useful when you are doing a series of changes that you want to ignore until you're done. In that use, use the `reload` parameter.

loginServices

Optional. A list of [org.eclipse.jetty.security.LoginService](#) implementations. Note that there is no default realm. If you use a realm in your `web.xml` you can specify a corre-

sponding realm here. You could instead configure the login services in a jetty xml file and add its location to the `jettyXml` parameter.

requestLog

Optional. An implementation of the [org.eclipse.jetty.server.RequestLog](#) request log interface. An implementation that respects the NCSA format is available as `org.eclipse.jetty.server.NCSAResponseLog`. There are three other ways to configure the RequestLog:

- In a jetty xml config file, as specified in the `jettyXml` parameter.
- In a context xml config file, as specified in the `contextXml` parameter.
- In the `webApp` element.

See [Configuring Request Logs](#) for more information.

stopPort

Optional. Port to listen on for stop commands. Useful to use in conjunction with the `stop` or `run-forked` goals.

stopKey

Optional. Used in conjunction with `stopPort` for stopping jetty. Useful when used in conjunction with the `stop` or `run-forked` goals.

systemProperties

Optional. Allows you to configure System properties for the execution of the plugin. For more information, see [Setting System Properties](#).

systemPropertiesFile

Optional. A file containing System properties to set for the execution of the plugin. By default, settings that you make here **do not** override any system properties already set on the command line, by the JVM, or in the POM via `systemProperties`. Read [Setting System Properties](#) for how to force overrides.

skip

Default is false. If true, the execution of the plugin exits. Same as setting the SystemProperty `-Djetty.skip` on the command line. This is most useful when configuring Jetty for execution during integration testing and you want to skip the tests

useProvidedScope

Default value is `false`. If true, the dependencies with `<scope>provided</scope>` are placed onto the *container classpath*. Be aware that this is NOT the webapp classpath, as "provided" indicates that these dependencies would normally be expected to be provided by the container. You should very rarely ever need to use this. Instead, you should copy the provided dependencies as explicit dependencies of the plugin instead.

excludedGoals

Optional. A list of jetty plugin goal names that will cause the plugin to print an informative message and exit. Useful if you want to prevent users from executing goals that you know cannot work with your project.

Configuring a Https Connector

In order to configure a https connector, you need to use jetty xml configuration files. This example uses files copied directly from the jetty distribution etc/ directory, although you can of course make up your own xml file or files. We will use the following files:

jetty.xml

Sets up various characteristics of the [org.eclipse.jetty.server.Server](#) instance for the plugin to use. Importantly, it sets up the

[org.eclipse.jetty.server.HttpConfiguration](#) element that we can refer to in subsequent xml files that configure the connectors. Here's the relevant section:

```
<New id="httpConfig" class="org.eclipse.jetty.server.HttpConfiguration">
  <Set name="secureScheme">https</Set>
  <Set name="securePort"><Property name="jetty.secure.port" default="8443" /></Set>
  <Set name="outputBufferSize">32768</Set>
  <Set name="requestHeaderSize">8192</Set>
  <Set name="responseHeaderSize">8192</Set>
  <Set name="sendServerVersion">true</Set>
  <Set name="sendDateHeader">false</Set>
  <Set name="headerCacheSize">512</Set>

  <!-- Uncomment to enable handling of X-Forwarded- style headers
  <Call name="addCustomizer">
    <Arg><New class="org.eclipse.jetty.server.ForwardedRequestCustomizer"/></Arg>
  </Call>
  -->
</New>
```

jetty-ssl.xml

Set up ssl which will be used by the https connector. Here's the `jetty-ssl.xml` file from the jetty-distribution:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<!-- =====>
<!-- Configure a TLS (SSL) Context Factory          -->
<!-- This configuration must be used in conjunction with jetty.xml -->
<!-- and either jetty-https.xml or jetty-spdylay.xml (but not both) -->
<!-- =====-->
<Configure id="sslContextFactory"
  class="org.eclipse.jetty.util.ssl.SslContextFactory">
  <Set name="KeyStorePath"><Property name="jetty.base" default=". " /></Property>
  <Set name="jetty.keystore" default="etc/keystore"/></Set>
  <Set name="KeyStorePassword"><Property name="jetty.keystore.password" default="OBF:1vny1zl0lx8e1vnwlvn6lx8g1zlulvn4" /></Set>
  <Set name="KeyManagerPassword"><Property name="jetty.keymanager.password" default="OBF:lu2ulwml1z7s1z7alwnll1u2g" /></Set>
  <Set name="TrustStorePath"><Property name="jetty.base" default=". " /></Property>
  <Set name="jetty.truststore" default="etc/keystore"/></Set>
  <Set name="TrustStorePassword"><Property name="jetty.truststore.password" default="OBF:1vny1zl0lx8e1vnwlvn6lx8g1zlulvn4" /></Set>
  <Set name="EndpointIdentificationAlgorithm"></Set>
  <Set name="NeedClientAuth"><Property name="jetty.ssl.needClientAuth" default="false" /></Set>
  <Set name="WantClientAuth"><Property name="jetty.ssl.wantClientAuth" default="false" /></Set>
  <Set name="ExcludeCipherSuites">
    <Array type="String">
      <Item>SSL_RSA_WITH_DES_CBC_SHA</Item>
      <Item>SSL_DHE_RSA_WITH_DES_CBC_SHA</Item>
      <Item>SSL_DHE_DSS_WITH_DES_CBC_SHA</Item>
      <Item>SSL_RSA_EXPORT_WITH_RC4_40_MD5</Item>
      <Item>SSL_RSA_EXPORT_WITH_DES40_CBC_SHA</Item>
      <Item>SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA</Item>
      <Item>SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA</Item>
    </Array>
  </Set>

  <!-- =====-->
  <!-- Create a TLS specific HttpConfiguration based on the          -->
  <!-- common HttpConfiguration defined in jetty.xml                  -->
  <!-- Add a SecureRequestCustomizer to extract certificate and     -->
  <!-- session information                                         -->
  <!-- =====-->
  <New id="sslHttpConfig" class="org.eclipse.jetty.server.HttpConfiguration">
    <Arg><Ref refid="httpConfig"/></Arg>
    <Call name="addCustomizer">
      <Arg><New class="org.eclipse.jetty.server.SecureRequestCustomizer"/></Arg>
    </Call>
  </New>
```

```
</Configure>
```

jetty-https.xml

Set up the https connector using the HttpConfiguration from jetty.xml and the ssl configuration from jetty-ssl.xml:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<!-- =====>
<!-- Configure a HTTPS connector.          -->
<!-- This configuration must be used in conjunction with jetty.xml -->
<!-- and jetty-ssl.xml.                  -->
<!-- =====>
<Configure id="Server" class="org.eclipse.jetty.server.Server">

<!-- =====-->
<!-- Add a HTTPS Connector.              -->
<!-- Configure an o.e.j.server.ServerConnector with connection -->
<!-- factories for TLS (aka SSL) and HTTP to provide HTTPS. -->
<!-- All accepted TLS connections are wired to a HTTP connection.-->
<!--
<!-- Consult the javadoc of o.e.j.server.ServerConnector,      -->
<!-- o.e.j.server.SslConnectionFactory and                   -->
<!-- o.e.j.server.HttpConnectionFactory for all configuration -->
<!-- that may be set here.                         -->
<!-- =====-->
<Call id="httpsConnector" name="addConnector">
    <Arg>
        <New class="org.eclipse.jetty.server.ServerConnector">
            <Arg name="server"><Ref refid="Server" /></Arg>
            <Arg name="factories">
                <Array type="org.eclipse.jetty.server.ConnectionFactory">
                    <Item>
                        <New class="org.eclipse.jetty.server.SslConnectionFactory">
                            <Arg name="next">http/1.1</Arg>
                            <Arg name="sslContextFactory"><Ref refid="sslContextFactory"/></
Arg>
                    </New>
                    <Item>
                        <New class="org.eclipse.jetty.server.HttpConnectionFactory">
                            <Arg name="config"><Ref refid="sslHttpConfig"/></Arg>
                        </New>
                    </Item>
                </Array>
            </Arg>
            <Set name="host"><Property name="jetty.host" /></Set>
            <Set name="port"><Property name="https.port" default="443" /></Set>
            <Set name="idleTimeout"><Property name="https.timeout" default="30000"/><
Set>
            <Set name="soLingerTime"><Property name="https.soLingerTime" default="-1"/></
Set>
        </New>
    </Arg>
</Call>
</Configure>
```

Now you need to let the plugin know to apply the files above:

```
<plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-maven-plugin</artifactId>
    <configuration>
        <jettyXml>jetty.xml,jetty-ssl.xml,jetty-https.xml</jettyXml>
    </configuration>
</plugin>
```

Caution



Just like with an installed distribution of Jetty, the ordering of the xml files is significant.

You can also use jetty xml files to configure a http connector for the plugin to use. Here we use the same `jetty-http.xml` file from the Jetty distribution:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">

<!-- ===== -->
<!-- Configure the Jetty Server instance with an ID "Server"      -->
<!-- by adding a HTTP connector.                                     -->
<!-- This configuration must be used in conjunction with jetty.xml -->
<!-- ===== -->
<Configure id="Server" class="org.eclipse.jetty.server.Server">

<!-- ===== -->
<!-- Add a HTTP Connector.                                         -->
<!-- Configure an o.e.j.server.ServerConnector with a single       -->
<!-- HttpConnectionFactory instance using the common httpConfig   -->
<!-- instance defined in jetty.xml                                -->
<!--
<!-- Consult the javadoc of o.e.j.server.ServerConnector and      -->
<!-- o.e.j.server.HttpConnectionFactory for all configuration    -->
<!-- that may be set here.                                         -->
<!-- ===== -->
<Call name="addConnector">
  <Arg>
    <New class="org.eclipse.jetty.server.ServerConnector">
      <Arg name="server"><Ref refid="Server" /></Arg>
      <Arg name="factories">
        <Array type="org.eclipse.jetty.server.ConnectionFactory">
          <Item>
            <New class="org.eclipse.jetty.server.HttpConnectionFactory">
              <Arg name="config"><Ref refid="httpConfig" /></Arg>
            </New>
          </Item>
        </Array>
      </Arg>
      <Set name="host"><Property name="jetty.host" /></Set>
      <Set name="port"><Property name="jetty.port" default="80" /></Set>
      <Set name="idleTimeout"><Property name="http.timeout" default="30000"/></Set>
      <Set name="soLingerTime"><Property name="http.soLingerTime" default="-1"/></Set>
    </New>
  </Arg>
</Call>
</Configure>
```

Now we add it to the list of configs for the plugin to apply:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <configuration>
    <jettyXml>jetty.xml,jetty-http.xml,jetty-ssl.xml,jetty-https.xml</jettyXml>
  </configuration>
</plugin>
```

Alternatively, you can use the `httpConnector` configuration element inside the pom instead as described above.

Configuring Your WebApp

These configuration parameters apply to your webapp. They are common to almost all goals.

webApp

Represents an extension to the class [org.eclipse.jetty.webapp.WebAppContext](#). You can use any of the setter methods on this object to configure your webapp. Here are a few of the most useful ones:

contextPath

The context path for your webapp. By default, this is set to `/`.

descriptor

The path to the `web.xml` file for your webapp.

defaultsDescriptor

The path to a `webdefault.xml` file that will be applied to your webapp before the `web.xml`. If you don't supply one, Jetty uses a default file baked into the `jetty-webapp.jar`.

overrideDescriptor

The path to a `web.xml` file that Jetty applies after reading your `web.xml`. You can use this to replace or add configuration.

tempDirectory

The path to a dir that Jetty can use to expand or copy jars and jsp compiles when your webapp is running. The default is `${project.build.outputDirectory} /tmp`.

baseResource

The path from which Jetty serves static resources. Defaults to `src/main/webapp`.

resourceBases

Use instead of `baseResource` if you have multiple dirs from which you want to serve static content. This is an array of dir names.

baseAppFirst

Defaults to "true". Controls whether any overlaid wars are added before or after the original base resource(s) of the webapp. See the section on [overlaid wars](#) for more information.

contextXml

The path to a context xml file that is applied to your webapp AFTER the `webApp` element.

jetty:run

The `run` goal runs on a webapp that does not have to be built into a WAR. Instead, Jetty deploys the webapp from its sources. It looks for the constituent parts of a webapp in the Maven default project locations, although you can override these in the plugin configuration. For example, by default it looks for:

- resources in `${project.basedir} /src/main/webapp`
- classes in `${project.build.outputDirectory}`
- `web.xml` in `${project.basedir} /src/main/webapp/WEB-INF/`

The plugin automatically ensures the classes are rebuilt and up-to-date before deployment. If you change the source of a class and your IDE automatically compiles it in the background, the plugin picks up the changed class.

You do not need to assemble the webapp into a WAR, saving time during the development cycle. Once invoked, you can configure the plugin to run continuously, scanning for changes in the project and automatically performing a hot redeploy when necessary. Any changes you make are immediately reflected in the running instance of Jetty, letting you quickly jump from coding to testing, rather than going through the cycle of: code, compile, reassemble, redeploy, test.

Here is a small example, which turns on scanning for changes every ten seconds, and sets the webapp context path to `/test`:

```
<plugin>
```

```
<groupId>org.eclipse.jetty</groupId>
<artifactId>jetty-maven-plugin</artifactId>
<configuration>
  <scanIntervalSeconds>10</scanIntervalSeconds>
  <webApp>
    <contextPath>/test</contextPath>
  </webApp>
</configuration>
</plugin>
```

Configuration

In addition to the `webApp` element that is common to most goals, the `jetty:run` goal supports:

`classesDirectory`

Location of your compiled classes for the webapp. You should rarely need to set this parameter. Instead, you should set `build outputDirectory` in your `pom.xml`.

`testClassesDirectory`

Location of the compiled test classes for your webapp. By default this is `${project.build.testOutputDirectory}`.

`useTestScope`

If true, the classes from `testClassesDirectory` and dependencies of scope "test" are placed first on the classpath. By default this is false.

`webAppSourceDirectory`

By default, this is set to `${project.basedir}/src/main/webapp`. If your static sources are in a different location, set this parameter accordingly.

`jettyEnvXml`

Optional. Location of a `jetty-env.xml` file, which allows you to make JNDI bindings that satisfy `env-entry`, `resource-env-ref`, and `resource-ref` linkages in the `web.xml` that are scoped only to the webapp and not shared with other webapps that you might be deploying at the same time (for example, by using a `jettyConfig` file).

`scanTargets`

Optional. A list of files and directories to periodically scan in addition to those the plugin automatically scans.

`scanTargetPatterns`

Optional. If you have a long list of extra files you want scanned, it is more convenient to use pattern matching expressions to specify them instead of enumerating them with the `scanTargetsList` of `scanTargetPatterns`, each consisting of a directory, and including and/or excluding parameters to specify the file matching patterns.

Here's an example:

```
<project>
...
<plugins>
...
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <configuration>
    <webAppSourceDirectory>${project.basedir}/src/staticfiles</webAppSourceDirectory>
    <webApp>
      <contextPath>/</contextPath>
      <descriptor>${project.basedir}/src/over/here/web.xml</descriptor>
      <jettyEnvXml>${project.basedir}/src/over/here/jetty-env.xml</jettyEnvXml>
    </webApp>
  </configuration>
</plugin>

```

```
</webApp>
<classesDirectory>${project.basedir}/somewhere/else</classesDirectory>
<scanTargets>
    <scanTarget>src/mydir</scanTarget>
    <scanTarget>src/myfile.txt</scanTarget>
</scanTargets>
<scanTargetPatterns>
    <scanTargetPattern>
        <directory>src/other-resources</directory>
        <includes>
            <include>**/*.xml</include>
            <include>**/*.properties</include>
        </includes>
        <excludes>
            <exclude>**/myspecial.xml</exclude>
            <exclude>**/myspecial.properties</exclude>
        </excludes>
    </scanTargetPattern>
</scanTargetPatterns>
</configuration>
</plugin>
</plugins>
</project>
```

If, for whatever reason, you cannot run on an unassembled webapp, the goals `run-war` and `run-exploded` work on unassembled webapps.

jetty:run-war

This goal first packages your webapp as a WAR file and then deploys it to Jetty. If you set a non-zero `scanInterval`, Jetty watches your `pom.xml` and the WAR file; if either changes, it redeploys the war.

Configuration

war

The location of the built WAR file. This defaults to `${project.build.directory} / ${project.build.finalName}.war`. If this is not sufficient, set it to your custom location.

Here's how to set it:

```
<project>
...
<plugins>
...
<plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-maven-plugin</artifactId>
    <configuration>
        <war>${project.basedir}/target/mycustom.war</war>
    </configuration>
</plugin>
</plugins>
</project>
```

jetty:run-exploded

The run-exploded goal first assembles your webapp into an exploded WAR file and then deploys it to Jetty. If you set a non-zero `scanInterval`, Jetty watches your `pom.xml`, `WEB-INF/lib`, `WEB-INF/classes` and `WEB-INF/web.xml` for changes and redeploys when necessary.

Configuration

war

The location of the exploded WAR. This defaults to `${project.build.directory} / ${project.build.finalName}`, but you can override the default by setting this parameter.

Here's how to set it:

```
<project>
...
<plugins>
...
<plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>maven-jetty-plugin</artifactId>
    <configuration>
        <war>${project.basedir}/target/myfunkywebapp</war>
    </configuration>
</plugin>
</plugins>
</project>
```

jetty:deploy-war

This is basically the same as `jetty:run-war`, but without assembling the WAR of the current module. Unlike `run-war`, the phase in which this plugin executes is not bound to the "package" phase. For example, you might want to start Jetty on the "test-compile" phase and stop Jetty on the "test-phase".

Configuration

war

The location of the WAR file. This defaults to `${project.build.directory} / ${project.build.finalName}`, but you can override the default by setting this parameter.

Here's the configuration:

```
<project>
...
<plugins>
...
<plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-maven-plugin</artifactId>
    <configuration>
        <war>${project.basedir}/target/mycustom.war</war>
    </configuration>
    <executions>
        <execution>
            <id>start-jetty</id>
            <phase>test-compile</phase>
            <goals>
                <goal>deploy-war</goal>
            </goals>
            <configuration>
                <daemon>true</daemon>
                <reload>manual</reload>
            </configuration>
        </execution>
        <execution>
            <id>stop-jetty</id>
            <phase>test</phase>
            <goals>
```

```
<goal>stop</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</project>
```

jetty:run-forked

This goal allows you to start the webapp in a new JVM, optionally passing arguments to that new JVM.

Configuration

NOTE: unfortunately, unlike most of the other goals, this one does NOT support a **webApp** parameter to configure the webapp. Therefore, if your webapp requires a lot of configuration, it will be difficult to switch between eg `jetty:run` and `jetty:run-forked` executions.

The available configuration parameters are:

jettyXml

The locations of jetty xml configuration files used to configure the container in the new JVM.

contextXml

Optional. The location of a context xml file to configure the webapp in the new JVM.

contextPath

Optional. The context path for the webapp in the new JVM. Defaults to `/${project.artifactId}`. Overrides a setting inside a `contextXml` file.

webAppSourceDirectory

Optional. The location of the static resources for your webapp. Defaults to `src/main/webapp`. Overrides a `Set name="baseResource"` setting inside a `contextXml` file.

resourceBases

Optional. An array of directories containing static content that form the resource base for your webapp, in conjunction with the `webAppSourceDirectory`. See also `baseAppFirst`.

baseAppFirst

Defaults to "true". Controls whether the `webAppSourceDirectory` or `resourceBases` are first on the list of resources that form the base resource for the webapp.

webXml

The location of the `web.xml` file. Defaults to `src/main/webapp/WEB-INF/web.xml`. Overrides a `Set name="descriptor"` inside a `contextXml` file.

tmpDirectory

Temporary directory to use for the webapp. Defaults to `${project.build.directory} / tmp`.

classesDirectory

The location of the compiled classes for the webapp. Defaults to `${project.build.outputDirectory}`.

testClassesDirectory

The location of the compiled test classes for the webapp. Defaults to `${project.build.testOutputDirectory}`.

useTestScope

–Defaults to "false". If true, the test classes and dependencies of `<scope>test</scope>` are placed on the webapp's classpath.

useProvidedScope

Defaults to "false". If true, the dependencies of scope "provided" are placed on the jetty container's classpath.

stopPort

Mandatory. A port number for jetty to listen on to receive a stop command to cause it to shutdown. If configured, the stopKey is used to authenticate an incoming stop command.

stopKey

Mandatory. A string value that has to be sent to the `stopPort` to authenticate the stop command.

skip

Optional. Defaults to false. If true, the execution of this plugin is skipped.

jvmArgs

Optional. A string representing arbitrary arguments to pass to the forked JVM.

To deploy your unassembled web app to Jetty running in a new JVM:

```
mvn jetty:run-forked
```

Jetty continues to execute until you either:

- Press `cntrl-c` in the terminal window to stop the plugin, which also stops the forked JVM.
- Use `jetty:stop` to stop the forked JVM, which also stops the plugin.

**Note**

If you want to set a custom port, you need to specify it in a `jetty.xml` file rather than setting the connector and port tags. You can specify the location of the `jetty.xml` using the `jettyXml` parameter.

jetty:start

This goal is for use with an execution binding in your `pom.xml`. It is similar to the `jetty:run` goal, however it does NOT first execute the build up until the "test-compile" phase to ensure that all necessary classes and files of the webapp have been generated. This is most useful when you want to control the start and stop of Jetty via execution bindings in your `pom.xml`.

For example, you can configure the plugin to start your webapp at the beginning of your unit tests and stop at the end. To do this, you need to set up a couple of execution scenarios for the Jetty plugin and use the `<daemon>true</daemon>` configuration option to force Jetty to execute only while Maven is running, instead of running indefinitely. You use the `pre-integration-test` and `post-integration-test` Maven build phases to trigger the execution and termination of Jetty:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <configuration>
    <scanIntervalSeconds>10</scanIntervalSeconds>
    <stopKey>foo</stopKey>
    <stopPort>9999</stopPort>
  </configuration>
  <executions>
    <execution>
      <id>start-jetty</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>start</goal>
      </goals>
    
```

```
<configuration>
  <scanIntervalSeconds>0</scanIntervalSeconds>
  <daemon>true</daemon>
</configuration>
</execution>
<execution>
  <id>stop-jetty</id>
  <phase>post-integration-test</phase>
  <goals>
    <goal>stop</goal>
  </goals>
</execution>
</executions>
</plugin>
```

Of course, you can use this goal from the command line (`mvn jetty:start`), however you need to be sure that all generated classes and files for your webapp are already present first.

jetty:stop

The stop goal stops a running instance of jetty. To use it, you need to configure the plugin with a special port number and key. That same port number and key will also be used by the `start` goal.

stopPort

A port number for jetty to listen on to receive a stop command to cause it to shutdown.

stopKey

A string value sent to the `stopPort` to validate the stop command.

stopWait

The maximum time in seconds that the plugin will wait for confirmation that jetty has stopped. If false or not specified, the plugin does not wait for confirmation but exits after issuing the stop command. **This parameter is available only since jetty-9.5.**

Here's a configuration example:

```
<plugin>
<groupId>org.eclipse.jetty</groupId>
<artifactId>jetty-maven-plugin</artifactId>
<configuration>
  <stopPort>9966</stopPort>
  <stopKey>foo</stopKey>
  <stopWait>10</stopWait>
</configuration>
</plugin>
```

Then, while Jetty is running (in another window), type:

```
mvn jetty:stop
```

The `stopPort` must be free on the machine you are running on. If this is not the case, you get an "Address already in use" error message after the "Started SelectedChannelConnector ..." message.

Using Overlaid wars

If your webapp depends on other war files, the `jetty:run` and `jetty:run-forked` goals are able to merge resources from all of them. It can do so based on the settings of the [maven-war-plugin](#), or if your project does not use the [maven-war-plugin](#) to handle the overlays, it can fall back to a simple algorithm to determine the ordering of resources.

With maven-war-plugin

The maven-war-plugin has a rich set of capabilities for merging resources. The jetty:run and jetty:run-forked goals are able to interpret most of them and apply them during execution of your unassembled webapp. This is probably best seen by looking at a concrete example.

Suppose your webapp depends on the following wars:

```
<dependency>
  <groupId>com.acme</groupId>
  <artifactId>X</artifactId>
  <type>war</type>
</dependency>
<dependency>
  <groupId>com.acme</groupId>
  <artifactId>Y</artifactId>
  <type>war</type>
</dependency>
```

Containing:

```
WebAppX:
/foo.jsp
/bar.jsp
/WEB-INF/web.xml

WebAppY:
/bar.jsp
/baz.jsp
/WEB-INF/web.xml
/WEB-INF/special.xml
```

They are configured for the [maven-war-plugin](#):

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
    <overlays>
      <overlay>
        <groupId>com.acme</groupId>
        <artifactId>X</artifactId>
        <excludes>
          <exclude>bar.jsp</exclude>
        </excludes>
      </overlay>
      <overlay>
        <groupId>com.acme</groupId>
        <artifactId>Y</artifactId>
        <excludes>
          <exclude>baz.jsp</exclude>
        </excludes>
      </overlay>
    </overlays>
  </configuration>
</plugin>
```

Then executing jetty:run would yield the following ordering of resources: com.acme.X.war : com.acme.Y.war: \${project.basedir}/src/main/webapp. Note that the current

project's resources are placed last in the ordering due to the empty `<overlay>` element in the maven-war-plugin. You can either use that, or specify the `<baseAppFirst>false</baseAppFirst>` parameter to the jetty-maven-plugin.

Moreover, due to the exclusions specified above, a request for the resource `bar.jsp` would only be satisfied from `com.acme.Y.war`. Similarly as `baz.jsp` is excluded, a request for it would result in a 404 error.

Without maven-war-plugin

The algorithm is fairly simple, is based on the ordering of declaration of the dependent wars, and does not support exclusions. The configuration parameter `<baseAppFirst>` (see the section on [Configuring Your Webapp](#) for more information) can be used to control whether your webapp's resources are placed first or last on the resource path at runtime.

For example, suppose our webapp depends on these two wars:

```
<dependency>
  <groupId>com.acme</groupId>
  <artifactId>X</artifactId>
  <type>war</type>
</dependency>
<dependency>
  <groupId>com.acme</groupId>
  <artifactId>Y</artifactId>
  <type>war</type>
</dependency>
```

Suppose the webapps contain:

```
WebAppX:
/foos.jsp
/bar.jsp
/WEB-INF/web.xml

WebAppY:
/bar.jsp
/baz.jsp
/WEB-INF/web.xml
/WEB-INF/special.xml
```

Then our webapp has available these additional resources:

```
/foos.jsp (X)
/bar.jsp (X)
/baz.jsp (Y)
/WEB-INF/web.xml (X)
/WEB-INF/special.xml (Y)
```

Configuring Security Settings

You can configure LoginServices in the plugin. Here's an example of setting up the HashLoginService for a webapp:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <configuration>
    <scanIntervalSeconds>10</scanIntervalSeconds>
    <webApp>
```

```
<contextPath>/test</contextPath>
</webApp>
<loginServices>
    <loginService implementation="org.eclipse.jetty.security.HashLoginService">
        <name>Test Realm</name>
        <config>${project.basedir}/src/etc/realm.properties</config>
    </loginService>
</loginServices>
</configuration>
</plugin>
```

Using Multiple Webapp Root Directories

If you have external resources that you want to incorporate in the execution of a webapp, but which are not assembled into WARs, you can't use the overlaid wars method described above, but you can tell Jetty the directories in which these external resources are located. At runtime, when Jetty receives a request for a resource, it searches all the locations to retrieve the resource. It's a lot like the overlaid war situation, but without the war. Here's a configuration example:

```
<configuration>
<webApp>
    <contextPath>/${build.finalName}</contextPath>
    <baseResource implementation="org.eclipse.jetty.util.resource.ResourceCollection">
        <resourcesAsCSV>src/main/webapp,/home/johndoe/path/to/my/other/source,/yet/another/
folder</resourcesAsCSV>
    </baseResource>
</webApp>
</configuration>
```

Running More than One Webapp

You can use either a `jetty.xml` file to configure extra (pre-compiled) webapps that you want to deploy, or you can use the `<contextHandlers>` configuration element to do so. If you want to deploy webapp A, and webapps B and C in the same Jetty instance:

Putting the configuration in webapp A's `pom.xml`:

```
<plugin>
<groupId>org.eclipse.jetty</groupId>
<artifactId>jetty-maven-plugin</artifactId>
<configuration>
    <scanIntervalSeconds>10</scanIntervalSeconds>
    <webApp>
        <contextPath>/test</contextPath>
    </webApp>
    <contextHandlers>
        <contextHandler implementation="org.eclipse.jetty.webapp.WebAppContext">
            <war>${project.basedir}.../B.war</war>
            <contextPath>/B</contextPath>
        </contextHandler>
        <contextHandler implementation="org.eclipse.jetty.webapp.WebAppContext">
            <war>${project.basedir}.../C.war</war>
            <contextPath>/B</contextPath>
        </contextHandler>
    </contextHandlers>
</configuration>
</plugin>
```

Alternatively, add a `jetty.xml` file to webapp A. Copy the `jetty.xml` file from the jetty distribution, and then add WebAppContexts for the other 2 webapps:

```
<Ref refid="Contexts">
```

```
<Call name="addHandler">
  <Arg>
    <New class="org.eclipse.jetty.webapp.WebAppContext">
      <Set name="contextPath"/>/B</Set>
      <Set name="war">../../B.war</Set>
    </New>
  </Arg>
</Call>
<Call>
  <Arg>
    <New class="org.eclipse.jetty.webapp.WebAppContext">
      <Set name="contextPath"/>/C</Set>
      <Set name="war">../../C.war</Set>
    </New>
  </Arg>
</Call>
</Ref>
```

Then configure the location of this `jetty.xml` file into webapp A's jetty plugin:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <configuration>
    <scanIntervalSeconds>10</scanIntervalSeconds>
    <webApp>
      <contextPath>/test</contextPath>
    </webApp>
    <jettyXml>src/main/etc/jetty.xml</jettyXml>
  </configuration>
</plugin>
```

For either of these solutions, the other webapps must already have been built, and they are not automatically monitored for changes. You can refer either to the packed WAR file of the pre-built webapps or to their expanded equivalents.

Setting System Properties

You can specify property name/value pairs that Jetty sets as System properties for the execution of the plugin. This feature is useful to tidy up the command line and save a lot of typing.

However, sometimes it is not possible to use this feature to set System properties - sometimes the software component using the System property is already initialized by the time that maven runs (in which case you will need to provide the System property on the command line), or by the time that jetty runs. In the latter case, you can use the [maven properties plugin](#) to define the system properties instead. Here's an example that configures the logback logging system as the jetty logger:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>properties-maven-plugin</artifactId>
  <version>1.0-alpha-2</version>
  <executions>
    <execution>
      <goals>
        <goal>set-system-properties</goal>
      </goals>
      <configuration>
        <properties>
          <property>
            <name>logback.configurationFile</name>
            <value>${project.baseUri}/resources/logback.xml</value>
          </property>
        </properties>
      </configuration>
    </execution>
  </executions>
</plugin>
```

```
</executions>
</plugin>
```

Note that if a System property is already set (for example, from the command line or by the JVM itself), then by default these configured properties DO NOT override them (see below for use of the <force> parameter).

Specifying System Properties in the POM

Here's an example of how to specify System properties in the POM:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <configuration>
    <systemProperties>
      <systemProperty>
        <name>fooprop</name>
        <value>222</value>
      </systemProperty>
    </systemProperties>
    <webApp>
      <contextPath>/test</contextPath>
    </webApp>
  </configuration>
</plugin>
```

To change the default behaviour so that these system properties override those on the command line, use the <force> parameter:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <configuration>
    <systemProperties>
      <force>true</force>
      <systemProperty>
        <name>fooprop</name>
        <value>222</value>
      </systemProperty>
    </systemProperties>
    <webApp>
      <contextPath>/test</contextPath>
    </webApp>
  </configuration>
</plugin>
```

Specifying System Properties in a File

You can also specify your System properties in a file. System properties you specify in this way DO NOT override System properties that set on the command line, by the JVM, or directly in the POM via `systemProperties`.

Suppose we have a file called `mysys.props` which contains the following:

```
fooprop=222
```

This can be configured on the plugin like so:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <configuration>
    <systemPropertiesFile>${project.basedir}/mysys.props</systemPropertiesFile>
    <webApp>
      <contextPath>/test</contextPath>
    </webApp>
  </configuration>
</plugin>
```

You can instead specify the file by setting the System property (!) jetty.systemPropertiesFile on the command line.

Files Scanned by the Jetty Maven Plugin

If you set a non zero [scanInterval](#) configuration parameter, the jetty maven plugin will scan certain files every [scanInterval](#) seconds for changes, and redeploy the webapp if necessary. The files that are scanned depend on the goal being executed.

Scanner Matrix

Goal	Files
jetty:run	pom.xml, <dependencies>, <classesDirectory>, <testClassesDirectory>, <webXml> or <webAppSourceDirectory>/WEB-INF/web.xml, <jettyEnvXml> or <webAppSourceDirectory>/WEB-INF/jetty-web.xml, <webAppSourceDirectory>/WEB-INF/jetty-web.xml, <scanTargets>, <scanTargetPatterns>, any defaultsDescriptor for the webapp, any overrideDescriptor for the webapp
jetty:run-war	pom.xml, <war>
jetty:run-exploded	pom.xml, <war>/WEB-INF/web.xml, <war>/WEB-INF/jetty-web.xml, <war>/WEB-INF/jetty-env.xml, <war>/WEB-INF/classes, <war>/WEB-INF/lib
jetty:deploy-war	pom.xml, <war>
jetty:run-forked	
jetty:start	pom.xml, <dependencies> from the pom, <classesDirectory>, <testClassesDirectory>, <webXml> or <webAppSourceDirectory>/WEB-INF/web.xml, <jettyEnvXml> or <webAppSourceDirectory>/WEB-INF/jetty-web.xml, <webAppSourceDirectory>/WEB-INF/jetty-web.xml, <scanTargets>, <scanTargetPatterns>, any defaultsDescriptor for the webapp, any overrideDescriptor for the webapp
jetty:stop	

Jetty Jspc Maven Plugin

This plugin will help you pre-compile your jsp's and works in conjunction with the maven war plugin to put them inside an assembled war.

Configuration

Here's the basic setup required to put the jspc plugin into your build:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-jspc-maven-plugin</artifactId>
  <version>9.1.0.RC1</version>
  <executions>
    <execution>
      <id>jspc</id>
      <goals>
        <goal>jspc</goal>
      </goals>
      <configuration>
        </configuration>
      </execution>
    </executions>
  </plugin>
```

The configurable parameters are as follows:

webXmlFragment

Default value: \${project.basedir}/target/webfrag.xml

File into which to generate the servlet declarations. Will be merged with an existing web.xml.

webAppSourceDirectory

Default value: \${project.basedir}/src/main/webapp

Root of resources directory where jsps, tags etc are located.

webXml

Default value: \${project.basedir}/src/main/webapp/WEB-INF/web.xml

The web.xml file to use to merge with the generated fragments.

includes

Default value: ***.jsp, ***.jspx

The comma separated list of patterns for file extensions to be processed.

excludes

Default value: ***.svn**

The comma separated list of patterns for file extensions to be skipped.

classesDirectory

Default value: \${project.build.outputDirectory}

Location of classes for the webapp.

generatedClasses

Default value: \${project.build.outputDirectory}

Location to put the generated classes for the jsps.

insertionMarker

Default value: *none*

A marker string in the src web.xml file which indicates where to merge in the generated web.xml fragment. Note that the marker string will NOT be preserved during the insertion. Can be left blank, in which case the generated fragment is inserted just before the line containing </web-app>.

useProvidedScope

Default value: false

If true, jars of dependencies marked with <scope>provided</scope> will be placed on the compilation classpath.

mergeFragment

Default value: true

Whether or not to merge the generated fragment file with the source web.xml. The merged file will go into the same directory as the webXmlFragment.

keepSources

Default value: false

If true, the generated .java files are not deleted at the end of processing.

tldJarNamePatterns

Default value: .*taglibs[^/]*.jar|.*jstl-impl[^/]*.jar\$

Patterns of jars on the 'system' (ie container) path that contain tlds. Use | to separate each pattern.

jspc

Default value: the org.apache.jasper.JspC instance being configured.

The JspC class actually performs the precompilation. All setters on the JspC class are available. You can [download the javadoc here](#).

Taking all the default settings, here's how to configure the war plugin to use the generated web.xml that includes all of the jsp servlet declarations:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
    <webXml>${project.basedir}/target/web.xml</webXml>
  </configuration>
</plugin>
```

Precompiling only for Production Build

As compiling jsps is usually done during preparation for a production release and not usually done during development, it is more convenient to put the plugin setup inside a <profile> which which can be deliberately invoked during prep for production.

For example, the following profile will only be invoked if the flag -Dprod is present on the run line:

```
<profiles>
  <profile>
    <id>prod</id>
    <activation>
      <property><name>prod</name></property>
    </activation>
    <build>
      <plugins>
        <plugin>
          <groupId>org.eclipse.jetty</groupId>
          <artifactId>jetty-jspc-maven-plugin</artifactId>
          <version>9.1.0.RC1</version>
          <!-- put your configuration in here -->
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
```

```
<artifactId>maven-war-plugin</artifactId>
    <!-- put your configuration in here -->
  </plugin>
</plugins>
</build>
</profile>
</profiles>
```

So, the following invocation would cause your code to be compiled, the jsps to be compiled, the <servlet> and <servlet-mapping>s inserted in the web.xml and your webapp assembled into a war:

```
$ mvn -Dprod package
```

Precompiling Jsp's with Overlaid Wars

Precompiling jsp's with an overlaid war requires a bit more configuration. This is because you need to separate the steps of unpacking the overlaid war and then repacking the final target war so the jetty-jspc-maven-plugin has the opportunity to access the overlaid resources.

In the example we'll show, we will use an overlaid war. The overlaid war will provide the web.xml file but the jsp's will be in src/main/webapp (ie part of the project that uses the overlay). We will unpack the overlaid war file, compile the jsp's and merge their servlet definitions into the extracted web.xml, then war up the lot.

Here's an example configuration of the war plugin that separate those phases into an unpack phase, and then a packing phase:

```
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <executions>
    <execution>
      <id>unpack</id>
      <goals><goal>exploded</goal></goals>
      <phase>generate-resources</phase>
      <configuration>
        <webappDirectory>target/foo</webappDirectory>
        <overlays>
          <overlay />
          <overlay>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>test-jetty-webapp</artifactId>
          </overlay>
        </overlays>
      </configuration>
    </execution>
    <execution>
      <id>pack</id>
      <goals><goal>war</goal></goals>
      <phase>package</phase>
      <configuration>
        <warSourceDirectory>target/foo</warSourceDirectory>
        <webXml>target/web.xml</webXml>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Now you also need to configure the jetty-jspc-maven-plugin so that it can use the web.xml that was extracted by the war unpacking and merge in the generated definitions of the servlets. This is in tar-

get/foo/WEB-INF/web.xml. Using the default settings, the web.xml merged with the jsp servlet definitions will be put into target/web.xml.

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-jspc-maven-plugin</artifactId>
  <version>9.1.0.RC1</version>
  <executions>
    <execution>
      <id>jspc</id>
      <goals>
        <goal>jspc</goal>
      </goals>
      <configuration>
        <webXml>target/foo/WEB-INF/web.xml</webXml>
        <includes>**/*.foo</includes>
        <excludes>**/*.fff</excludes>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Chapter 24. Using Ant with Jetty

Table of Contents

Using the Ant Jetty Plugin 312

This chapter explains how to use Jetty with Ant and the Jetty Ant tasks.

Using the Ant Jetty Plugin

The Ant Jetty plugin is a part of Jetty 9 under the `jetty-ant` module. This plugin makes it possible to start a Jetty web server directly from the Ant build script, and to embed the Jetty web server inside your build process. Its purpose is to provide almost the same functionality as the Jetty plugin for Maven: dynamic application reloading, working directly on web application sources, and tightly integrating with the build system.

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-ant</artifactId>
</dependency>
```

Preparing Your Project

To set up your project for Ant to run Jetty, you need a Jetty distribution and the `jetty-ant` Jar:

1. [Download](#) a Jetty distribution and unpack it in the local filesystem.
2. [Get](#) the `jetty-ant` Jar.
3. Make a directory in your project called `jetty-lib/`.
4. Copy all of the Jars in your Jetty distribution's `lib` directory, and all its subdirectories, into your new `jetty-lib` dir. When copying the Jars, *don't* preserve the Jetty distribution's `lib` dir hierarchy – all the jars should be directly inside your `jetty-lib` dir.
5. Also copy the `jetty-ant` Jar you downloaded earlier into the `jetty-lib` dir.
6. Make a directory in your project called `jetty-temp`.

Now you're ready to edit or create your Ant `build.xml` file.

Preparing the `build.xml` file

Begin with an empty `build.xml`:

```
<project name="Jetty-Ant integration test" basedir=".">
</project>
```

Add a `<taskdef>` that imports all available Jetty tasks:

```
<project name="Jetty-Ant integration test" basedir=".">
  <path id="jetty.plugin.classpath">
    <fileset dir="jetty-lib" includes="*.jar"/>
  </path>

  <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader"
  />

</project>
```

Now you are ready to add a new target for running Jetty:

```
<project name="Jetty-Ant integration test" basedir=".">
  <path id="jetty.plugin.classpath">
    <fileset dir="jetty-lib" includes="*.jar"/>
  </path>

  <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader"
  />

  <target name="jetty.run">
    <jetty.run />
  </target>

</project>
```

This is the minimal configuration you need. You can now start Jetty on the default port of 8080.

Starting Jetty via Ant

At the command line enter:

```
> ant jetty.run
```

Configuring the Jetty Container

A number of configuration options can help you set up the Jetty environment so that your web application has all the resources it needs:

ports and connectors:

To configure the port that Jetty starts on you need to define a connector. First you need to configure a `<typedef>` for the Connector class and then define the connector in the Jetty tags:

```
<project name="Jetty-Ant integration test" basedir=".">
  <path id="jetty.plugin.classpath">
    <fileset dir="jetty-lib" includes="*.jar"/>
  </path>

  <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader"
  />

  <typedef name="connector" classname="org.eclipse.jetty.ant.types.Connector"
    classpathref="jetty.plugin.classpath" loaderref="jetty.loader" />

  <target name="jetty.run">
    <jetty.run>
      <connectors>
        <connector port="8090"/>
      </connectors>
    </jetty.run>
  </target>
```

```
</project>
```



Tip

You can set the port to 0, which starts the Jetty server connector on an arbitrary available port. You can then access these values from system properties `jetty.ant.server.port` and `jetty.ant.server.host`.

login services:

If your web application requires authentication and authorization services, you can configure these on the Jetty container. Here's an example of how to set up an [org.eclipse.jetty.security.HashLoginService](#):

```
<project name="Jetty-Ant integration test" basedir=".">>

    <path id="jetty.plugin.classpath">
        <fileset dir="jetty-lib" includes="*.jar"/>
    </path>

    <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader" />

    <typedef name="hashLoginService" classname="org.eclipse.jetty.security.HashLoginService" classpathref="jetty.plugin.classpath" loaderref="jetty.loader" />

    <target name="jetty.run">
        <jetty.run>
            <loginServices>
                <hashLoginService name="Test Realm" config="${basedir}/realm.properties"/>
            </loginServices>
        </jetty.run>
    </target>

</project>
```

request log:

The `requestLog` option allows you to specify a request logger for the Jetty instance. You can either use the [org.eclipse.jetty.server.NCSARequestLog](#) class, or supply the name of your custom class:

```
<project name="Jetty-Ant integration test" basedir=".">>

    <path id="jetty.plugin.classpath">
        <fileset dir="jetty-lib" includes="*.jar"/>
    </path>

    <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader" />

    <target name="jetty.run">
        <jetty.run requestLog="com.acme.MyFancyRequestLog">
        </jetty.run>
    </target>

</project>
```

temporary directory:

You can configure a directory as a temporary file store for uses such as expanding files and compiling JSPs by supplying the `tempDirectory` option:

```
<project name="Jetty-Ant integration test" basedir=".">
  <path id="jetty.plugin.classpath">
    <fileset dir="jetty-lib" includes="*.jar"/>
  </path>

  <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader" />

  <target name="jetty.run">
    <jetty.run tempDirectory="${basedir}/jetty-temp">
    </jetty.run>
  </target>

</project>
```

other context handlers:

You may need to configure some other context handlers to run at the same time as your web application. You can specify these other context handlers using the `<contextHandlers>` element. You need to supply a `<typedef>` for it before you can use it:

```
<project name="Jetty-Ant integration test" basedir=".">
  <path id="jetty.plugin.classpath">
    <fileset dir="jetty-lib" includes="*.jar"/>
  </path>

  <taskdef classpathref="jetty.plugin.classpath"
           resource="tasks.properties" loaderref="jetty.loader" />

  <typedef name="contextHandlers" classname="org.eclipse.jetty.ant.types.ContextHandlers"
           classpathref="jetty.plugin.classpath" loaderref="jetty.loader" />

  <target name="jetty.run">
    <jetty.run>
      <contextHandlers>
        <contextHandler resourceBase="${basedir}/stuff" contextPath="/stuff"/>
      </contextHandlers>
    </jetty.run>
  </target>

</project>
```

system properties:

As a convenience, you can configure system properties by using the `<systemProperties>` element. Be aware that, depending on the purpose of the system property, setting it from within the Ant execution may mean that it is evaluated too late, as the JVM evaluates some system properties on entry.

```
<project name="Jetty-Ant integration test" basedir=".">
  <path id="jetty.plugin.classpath">
    <fileset dir="jetty-lib" includes="*.jar"/>
  </path>

  <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader" />

  <target name="jetty.run">
    <jetty.run>
      <systemProperties>
        <systemProperty name="foo" value="bar"/>
      </systemProperties>
    </jetty.run>
  </target>

</project>
```

jetty XML file:

If you have a lot of configuration to apply to the Jetty container, it can be more convenient to put it into a standard Jetty XML configuration file and have the Ant plugin apply it before starting Jetty:

```
<project name="Jetty-Ant integration test" basedir="."> <path id="jetty.plugin.classpath"> <fileset dir="jetty-lib" includes="*.jar"/> </path> <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader" /> <target name="jetty.run"> <jetty.run jettyXml="${basedir}/jetty.xml"> </jetty.run> </target> </project>
```

scanning for changes:

The most useful mode in which to run the Ant plugin is for it to continue to execute Jetty and automatically restart your web application if any part of it changes (for example, your IDE recompiles the classes of the web application). The `scanIntervalSeconds` option controls how frequently the `<jetty.run>` task scans your web application/WAR file for changes. The default value of 0 disables scanning. Here's an example where Jetty checks for changes every five seconds:

```
<project name="Jetty-Ant integration test" basedir="."> <path id="jetty.plugin.classpath"> <fileset dir="jetty-lib" includes="*.jar"/> </path> <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader" /> <target name="jetty.run"> <jetty.run scanIntervalSeconds="5"> </jetty.run> </target> </project>
```

stopping:

In normal mode (`daemon="false"`), the `<jetty.run>` task runs until you `ctrl-c` it. It may be useful to script both the stop AND the start of Jetty. For such a case, we provide the `<jetty.stop>` task.

To use it, you need to provide a port and an identifying string to both the `<jetty.run>` and the `<jetty.stop>` tasks, where `<jetty.run>` listens on the given port for a stop message containing the given string, and cleanly stops Jetty when it is received. The `<jetty.stop>` task sends this stop message. You can also optionally provide a `stopWait` value (in seconds), which is the length of time the `<jetty.stop>` task waits for confirmation that the stop succeeded:

```
<project name="Jetty-Ant integration test" basedir="."> <path id="jetty.plugin.classpath"> <fileset dir="jetty-lib" includes="*.jar"/> </path> <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader" />
```

```
<target name="jetty.run">
  <jetty.run stopPort="9999" stopKey="9999">
  </jetty.run>
</target>

<target name="jetty.stop">
  <jetty.stop stopPort="9999" stopKey="9999" stopWait="10"/>
</target>

</project>
```

To stop jetty via Ant, enter:

```
> ant jetty.stop
```

execution without pausing ant:

Usually, the `<jetty.run>` task runs until you `ctrl-c` it, pausing the execution of Ant as it does so. In some cases, it may be useful to let Ant continue executing. For example, to run your unit tests you may need other tasks to execute while Jetty is running. For this case, we provide the `daemon` option. This defaults to `false`. For `true`, Ant continues to execute after starting Jetty. If Ant exits, so does Jetty. Understand that this option does *not* fork a new process for Jetty.

```
<project name="Jetty-Ant integration test" basedir=".">
  <path id="jetty.plugin.classpath">
    <fileset dir="jetty-lib" includes="*.jar"/>
  </path>

  <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader" />

  <target name="jetty.run">
    <jetty.run daemon="true">
    </jetty.run>
  </target>
</project>
```

Deploying a Web Application

Add a `<typedef>` for the `org.eclipse.jetty.ant.AntWebAppContext` class with name `webApp`, then add a `<webApp>` element to `<jetty.run>` to describe your web application. The following example deploys a web application that is expanded in the local directory `foo/` to context path `/`:

```
<project name="Jetty-Ant integration test" basedir=".">
  <path id="jetty.plugin.classpath">
    <fileset dir="jetty-lib" includes="*.jar"/>
  </path>

  <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader" />

  <typedef name="webApp" classname="org.eclipse.jetty.ant.AntWebAppContext"
    classpathref="jetty.plugin.classpath" loaderref="jetty.loader" />

  <target name="jetty.run">
    <jetty.run>
      <webApp war="${basedir}/foo" contextPath="/" />
    </jetty.run>
  </target>
</project>
```

deploying a WAR file:

It is not necessary to expand the web application into a directory. It is fine to deploy it as a WAR file:

```
<project name="Jetty-Ant integration test" basedir=".">>

    <path id="jetty.plugin.classpath">
        <fileset dir="jetty-lib" includes="*.jar"/>
    </path>

    <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader" />

    <typedef name="webApp" classname="org.eclipse.jetty.ant.AntWebAppContext"
             classpathref="jetty.plugin.classpath" loaderref="jetty.loader" />

    <target name="jetty.run">
        <jetty.run>
            <webApp war="${basedir}/foo.war" contextPath="/" />
        </jetty.run>
    </target>

</project>
```

deploying more than one web application:

You can also deploy more than one web application:

```
<project name="Jetty-Ant integration test" basedir=".">>

    <path id="jetty.plugin.classpath">
        <fileset dir="jetty-lib" includes="*.jar"/>
    </path>

    <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader" />

    <typedef name="webApp" classname="org.eclipse.jetty.ant.AntWebAppContext"
             classpathref="jetty.plugin.classpath" loaderref="jetty.loader" />

    <target name="jetty.run">
        <jetty.run>
            <webApp war="${basedir}/foo.war" contextPath="/" />
            <webApp war="${basedir}/other" contextPath="/other" />
            <webApp war="${basedir}/bar.war" contextPath="/bar" />
        </jetty.run>
    </target>

</project>
```

Configuring the Web Application

As the `org.eclipse.jetty.ant.AntWebAppContext` class is an extension of the [org.eclipse.jetty.webapp.WebAppContext](#) class, you can configure it by adding attributes of the same name (without the set or add prefix) as the setter methods.

Here's an example that specifies the location of the `web.xml` file (equivalent to method [`AntWebAppContext.setDescriptor\(\)`](#)) and the web application's temporary directory (equivalent to method [`AntWebAppContext.setTempDirectory\(\)`](#)):

```
<project name="Jetty-Ant integration test" basedir=".">>

    <path id="jetty.plugin.classpath">
        <fileset dir="jetty-lib" includes="*.jar"/>
```

```
</path>

<taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader"
/>

<typedef name="webApp" classname="org.eclipse.jetty.ant.AntWebAppContext"
          classpathref="jetty.plugin.classpath" loaderref="jetty.loader" />

<target name="jetty.run">
    <jetty.run>
        <webApp descriptor="${basedir}/web.xml" tempDirectory="${basedir}/my-
temp" war="${basedir}/foo" contextPath="/" />
    </jetty.run>
</target>

</project>
```

Other extra configuration options for the AntWebAppContext include:

extra classes and Jars:

If your web application's classes and Jars do not reside inside WEB-INF of the resource base directory, you can use the <classes> and <jar> elements to tell Ant where to find them. Here's an example:

```
<project name="Jetty-Ant integration test" basedir=".">
    <path id="jetty.plugin.classpath">
        <fileset dir="jetty-lib" includes="*.jar"/>
    </path>

    <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader"
    />

    <typedef name="webApp" classname="org.eclipse.jetty.ant.AntWebAppContext"
              classpathref="jetty.plugin.classpath" loaderref="jetty.loader" />

    <target name="jetty.run">
        <jetty.run>
            <webApp descriptor="${basedir}/web.xml" tempDirectory="${basedir}/my-
temp" war="${basedir}/foo" contextPath="/" />
            <classes dir="${basedir}/classes">
                <include name="**/*.class"/>
                <include name="**/*.properties"/>
            </classes>
            <lib dir="${basedir}/jars">
                <include name="**/*.jar"/>
                <exclude name="**/*.dll"/>
            </lib>
        </webApp>
    </jetty.run>
</target>

</project>
```

context attributes:

Jetty allows you to set up ServletContext attributes on your web application. You configure them in a context XML file that is applied to your WebAppContext instance prior to starting it. For convenience, the Ant plugin permits you to configure these directly in the build file. Here's an example:

```
<project name="Jetty-Ant integration test" basedir=".">
    <path id="jetty.plugin.classpath">
        <fileset dir="jetty-lib" includes="*.jar"/>
    </path>

    <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader"
    />
```

```
<typedef name="webApp" classname="org.eclipse.jetty.ant.AntWebAppContext"
         classpathref="jetty.plugin.classpath" loaderref="jetty.loader" />

<target name="jetty.run">
  <jetty.run>
    <webApp war="${basedir}/foo" contextPath="/">
      <attributes>
        <attribute name="my.param" value="123"/>
      </attributes>
    </webApp>
  </jetty.run>
</target>

</project>
```

jetty-env.xml file:

If you are using features such as [JNDI](#) with your web application, you may need to configure a [WEB-INF/jetty-env.xml](#) file to define resources. If the structure of your web application project is such that the source of jetty-env.xml file resides somewhere other than WEB-INF, you can use the `jettyEnvXml` attribute to tell Ant where to find it:

```
<project name="Jetty-Ant integration test" basedir=".">
  <path id="jetty.plugin.classpath">
    <fileset dir="jetty-lib" includes="*.jar"/>
  </path>

  <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader" />

  <typedef name="webApp" classname="org.eclipse.jetty.ant.AntWebAppContext"
            classpathref="jetty.plugin.classpath" loaderref="jetty.loader" />

  <target name="jetty.run">
    <jetty.run>
      <webApp war="${basedir}/foo" contextPath="/" jettyEnvXml="${basedir}/jetty-env.xml">
        <attributes>
        </webApp>
      </jetty.run>
    </target>
  </project>
```

context XML file:

You may prefer or even require to do some advanced configuration of your web application outside of the Ant build file. In this case, you can use a standard context XML configuration file which the Ant plugin applies to your web application before it is deployed. Be aware that the settings from the context XML file *override* those of the attributes and nested elements you defined in the build file.

```
project name="Jetty-Ant integration test" basedir=".">
  <path id="jetty.plugin.classpath">
    <fileset dir="jetty-lib" includes="*.jar"/>
  </path>

  <taskdef classpathref="jetty.plugin.classpath" resource="tasks.properties" loaderref="jetty.loader" />

  <typedef name="webApp" classname="org.eclipse.jetty.ant.AntWebAppContext"
            classpathref="jetty.plugin.classpath" loaderref="jetty.loader" />

  <target name="jetty.run">
    <jetty.run>
      <webApp war="${basedir}/foo" contextPath="/" contextXml="${basedir}/jetty-env.xml">
      </webApp>
    </jetty.run>
  </target>
</project>
```

```
<attributes>
  </webApp>
  </jetty.run>
</target>

</project>
```

Chapter 25. Handlers

Table of Contents

Rewrite Handler	322
Writing Custom Handlers	324

Rewrite Handler

The RewriteHandler matches a request against a set of rules, and modifies the request accordingly for any rules that match. The most common use is to rewrite request URIs, but it is capable of much more: rules can also be configured to redirect the response, set a cookie or response code on the response, modify the header, etc.

Quick Start

The standard Jetty distribution bundle contains the `jetty-rewrite` module JAR, at `lib/jetty-rewrite-*.jar`, and a sample configuration file, at `etc/jetty-rewrite.xml`. To enable the rewrite module, using the sample configuration file, start up Jetty with this command:

```
$ java -jar start.jar OPTIONS=default,rewrite etc/jetty.xml etc/jetty-rewrite.xml
```



Note

If you are running the standard Jetty distribution with the sample test webapp, there will be a demo of the rewrite module at <http://localhost:8080/rewrite/>

Configuring Rules

The rules are configured using `jetty.xml` syntax. This example file shows how to add the rewrite handler for the entire server:

```
<Configure id="Server" class="org.eclipse.jetty.server.Server">
    <!-- create and configure the rewrite handler -->
    <New id="Rewrite" class="org.eclipse.jetty.rewrite.handler.RewriteHandler">
        <Set name="rewriteRequestURI">true</Set>
        <Set name="rewritePathInfo">false</Set>
        <Set name="originalPathAttribute">requestedPath</Set>

        <!-- redirect the response. This is a redirect which is visible to the browser.
            After the redirect, the browser address bar will show /redirected -->
        <Call name="addRule">
            <Arg>
                <New class="org.eclipse.jetty.rewrite.handler.RedirectPatternRule">
                    <Set name="pattern">/redirect/*</Set>
                    <Set name="replacement">/redirected</Set>
                </New>
            </Arg>
        </Call>

        <!-- rewrite the request URI. This is an internal rewrite, visible to server,
            but the browser will still show /some/old/context -->
        <Call name="addRule">
            <Arg>
```

```

<New class="org.eclipse.jetty.rewrite.handler.RewritePatternRule">
    <Set name="pattern">/some/old/context</Set>
    <Set name="replacement">/some/new/context</Set>
    </New>
</Arg>
</Call>

<!-- reverse the order of the path sections. Internal rewrite -->
<Call name="addRule">
    <Arg>
        <New class="org.eclipse.jetty.rewrite.handler.RewriteRegexRule">
            <Set name="regex">/reverse/([^\/*])/(.*)</Set>
            <Set name="replacement">/reverse/$2/$1</Set>
        </New>
    </Arg>
    </Call>
</New>

<!-- add the rewrite handler to the server -->
<Set name="handler"><Ref id="Rewrite" /></Set>
</Configure>

```

See `etc/jetty-rewrite.xml` for more configuration examples.

Embedded Example

This is an example for embedded Jetty, which does the same thing as the configuration file example above:

```

Server server = new Server();

RewriteHandler rewrite = new RewriteHandler();
rewrite.setRewriteRequestURI(true);
rewrite.setRewritePathInfo(false);
rewrite.originalPathAttribute("requestedPath");

RedirectPatternRule redirect = new RedirectPatternRule();
redirect.setPattern("/redirect/*");
redirect.setReplacement("/redirected");
rewrite.addRule(redirect);

RewritePatternRule oldToNew = new RewritePatternRule();
oldToNew.setPattern("/some/old/context");
oldToNew.setReplacement("/some/new/context");
rewrite.addRule(oldToNew);

RewriteRegexRule reverse = new RewriteRegexRule();
reverse.setRegex("/reverse/([^\/*])/(.*)");
reverse.setReplacement("/reverse/$2/$1");
rewrite.addRule(reverse);

server.setHandler(rewrite);

```

Rules

There are several types of rules that are written extending useful base rule classes.

PatternRule

Matches against the request URI using the servlet pattern syntax.

[CookiePatternRule](#)

Adds a cookie to the response.

[HeaderPatternRule](#)

Adds/modifies a header in the response.

[RedirectPatternRule](#)

Redirects the response.

[ResponsePatternRule](#)

Sends the response code (status or error).

[RewritePatternRule](#)

Rewrite the URI by replacing the matched request path with a fixed string.

RegexRule

Matches against the request URI using regular expressions.

[RedirectRegexRule](#)

Redirect the response.

[RewriteRegexRule](#)

Rewrite the URI by matching with a regular expression. (The replacement string may use Template:\$n to replace the nth capture group.)

HeaderRule

Match against request headers. Match either on a header name + specific value, or on the presence of a header (with any value).

[ForwardedSchemaHeaderRule](#)

Set the scheme on the request (defaulting to https).

Others

Oddball rules that defy classification.

[MsieSslRule](#)

Disables the keep alive for SSL from IE5 or IE6.

[LegacyRule](#)

Implements the legacy API of RewriteHandler

RuleContainer

Groups rules together. The contained rules will only be processed if the conditions for the RuleContainer evaluate to true.

[VirtualHostRuleContainer](#)

Groups rules that apply only to a specific virtual host or a set of virtual hosts

Writing Custom Handlers

The Handler is the Jetty component that deals with received requests.

Many users of Jetty never need to write a Jetty Handler, but instead use the [Servlet API](#). You can reuse the existing Jetty handlers for context, security, sessions and servlets without the need for extension. However, some users might have special requirements or footprint concerns that prohibit the use of the full servlet API. For them implementing a Jetty handler is a straight forward way to provide dynamic web content with a minimum of fuss.

See the section called “Jetty Architecture” to understand more about Handlers vs. Servlets.

The Handler API

The [Handler](#) interface provides Jetty's core of content generation or manipulation. Classes that implement this interface are used to coordinate requests, filter requests and generate content.

The core API of the Handler interface is:

```
public void handle(String target, Request baseRequest, HttpServletRequest request,
    HttpServletResponse response)
    throws IOException, ServletException
```

An implementation of this method can handle a request, pass the request onto another handler (or servlet), or it can modify and/or wrap the request and then pass it on. This gives three styles of handler:

- Coordinating Handlers - Handlers that route requests to other handlers (HandlerCollection, ContextHandlerCollection)
- Filtering Handlers - Handlers that augment a request and pass it on to other handlers (HandlerWrapper, ContextHandler, SessionHandler)
- Generating Handlers - Handlers that produce content (ResourceHandler and ServletHandler)

The Target

The target of a handler is an identifier for the resource that should handle the passed request. This is normally the URI that is parsed from an HTTP Request. However, in two key circumstances the target may differ from the URI of the passed request:

- If the request has been dispatched to a named resource, such as a named servlet, the target is the name of that resource.
- If the request is being made by a call to [Request Dispatcher](#), the target is the URI of the included resource and is different to the URI of the actual request.

The Request and Response

The request and response objects used in the signature of the handle method are [Servlet Request](#) and [Servlet Response](#). These are the standard APIs and are moderately restricted in what they can do to the request and response. More often than not, access to the Jetty implementations of these classes is required: [Request](#) and [Response](#). However, as the request and response may be wrapped by handlers, filters and servlets, it is not possible to pass the implementation directly. The following mantra retrieves the core implementation objects from under any wrappers:

```
Request base_request = request instanceof Request ? (Request)request :
    HttpConnection.getCurrentConnection().getRequest();
Response base_response = response instanceof Response ? (Response)response :
    HttpConnection.getCurrentConnection().getResponse();
```

Notice that if the handler passes the request on to another handler, it should use the request/response objects passed in, and not the base objects. This is to preserve any wrapping done by up stream handlers.

The Dispatch

The dispatch argument indicates the state of the handling of the call and may be:

- REQUEST == 1 - An original request received from a connector.

- FORWARD == 2 - A request being forwarded by a RequestDispatcher.
- INCLUDE == 4 - A request being included by a RequestDispatcher.
- ERROR == 8 - A request being forwarded to a error handler by the container.

These mostly have significance for servlet and related handlers. For example, the security handler only applies authentication and authorization to REQUEST dispatches.

Handling Requests

A Handler may handle a request by:

- the section called “Generating a Response”
- the section called “Filtering the Request and/or Response”
- the section called “Passing the Request and Response to Another Handler”

Generating a Response

The [OneHandler](#) embedded example shows how a simple handler can generate a response.

You can use the normal servlet response API, which will typically set some status, content headers and then write out the content:

```
response.setContentType("text/html");
response.setStatus(HttpServletResponse.SC_OK);
response.getWriter().println("<h1>Hello OneHandler</h1>");
```

It is also very important that a handler indicate that it has completed handling the request and that the request should not be passed to other handlers:

```
Request base_request = (request instanceof Request) ?
(Request)request:HttpConnection.getCurrentConnection().getRequest();
base_request.setHandled(true);
```

Filtering the Request and/or Response

Once the base request or response object is obtained, you can modify it. Typically you would make modifications to accomplish:

- Breaking the URI into contextPath, servletPath and pathInfo components.
- Associating a resource base with a request for static content.
- Associating a session with a request.
- Associating a security principal with a request.
- Changing the URI and paths during a request dispatch forward to another resource.

You can also update the context of the request:

- Setting the current threads context classloader.
- Setting thread locals to identify the current ServletContext.

Typically Jetty passes a modified request to another handler and undoes modifications in a finally block afterwards:

```
try
```

```
{  
    base_request.setSession(a_session);  
    next_handler.handle(target,request,response,dispatch);  
}  
finally  
{  
    base_request.setSession(old_session);  
}
```

The classes that implement the [HandlerWrapper](#) class are typically handler filters of this style.

Passing the Request and Response to Another Handler

A handler might simply inspect the request and use the target, request URI or other information to select another handler to pass the request to. These handlers typically implement the [HandlerContainer](#) interface.

Examples include:

- [Class Handler Collection](#) - A collection of handlers, where each handler is called regardless of the state of the request. This is typically used to pass a request to a [ContextHandlerCollection](#), and then the [RequestLogHandler](#).
- [HandlerList](#) - A list of handlers that are called in turn until the request state is set as handled.
- [ContextHandlerCollection](#) - A collection of Handlers, of which one is selected by best match for the context path.

More About Handlers

See [Jetty 7 Latest Source XRef](#) and [Jetty 7 Latest JavaDoc](#) for detailed information on each Jetty handler.

Chapter 26. Embedding

Table of Contents

Jetty Embedded HelloWorld	328
Embedding Jetty	329
Embedded Examples	334

Jetty Embedded HelloWorld

This section provides a tutorial that shows how you can quickly develop embedded code against the Jetty API.

Downloading the Jars

Jetty is decomposed into many jars and dependencies to achieve a minimal footprint by selecting the minimal set of jars. Typically it is best to use something like Maven to manage jars, however this tutorial uses an aggregate Jar that contains all of the Jetty classes in one Jar. You can manually download the aggregate jetty-all Jar and the servlet api Jar using [wget](#) or similar command (for example, [curl](#)) or a browser. Use wget as follows:

```
mkdir Demo  
cd Demo  
wget -O jetty-all.jar -U none http://central.maven.org/maven2/org/eclipse/jetty/  
aggregate/jetty-all/9.0.0-RC1/jetty-all-9.0.0-RC1.jar  
wget -O servlet-api.jar -U none http://central.maven.org/maven2/org/eclipse/jetty/orbit/  
javax.servlet/3.0.0.v201112011016/javax.servlet-3.0.0.v201112011016.jar
```

Writing a HelloWorld Example

The [Embedding Jetty](#) section contains many examples of writing against the Jetty API. This tutorial uses a simple HelloWorld handler with a main method to run the server. You can either [download](#) or create in an editor the file `HelloWorld.java` with the following content:

```
import java.io.IOException;  
import javax.servlet.ServletException;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import org.eclipse.jetty.server.Request;  
import org.eclipse.jetty.server.Server;  
import org.eclipse.jetty.server.handler.AbstractHandler;  
  
public class HelloWorld extends AbstractHandler {  
  
    @Override  
    public void handle(String target, Request baseRequest, HttpServletRequest request,  
    HttpServletResponse response) throws IOException, ServletException {  
        response.setContentType("text/html;charset=utf-8");  
        response.setStatus(HttpServletResponse.SC_OK);  
        baseRequest.setHandled(true);  
        response.getWriter().println("<h1>Hello World</h1>");  
    }  
  
    public static void main(String[] args) throws Exception {  
        Server server = new Server(8080);  
        server.setHandler(new HelloWorld());  
        server.start();  
        server.join();  
    }  
}
```

Compiling the HelloWord example

The following command compiles the HelloWorld class:

```
javac -cp servlet-api.jar:jetty-all.jar HelloWorld.java
```

Running the Handler and Server

The following command runs the HelloWorld example:

```
java -cp .:servlet-api.jar:jetty-all.jar HelloWorld
```

You can now point your browser at <http://localhost:8080> to see your hello world page.

Next Steps

To learn more about Jetty, take these next steps:

- Follow the examples in [Embedding Jetty](#) to better understand the jetty APIs.
- Explore the complete [jetty javadoc](#)
- Consider using [Jetty and Maven](#) to manage your Jars and dependencies.

Embedding Jetty

Jetty has a slogan, "*Don't deploy your application in Jetty, deploy Jetty in your application!*" What this means is that as an alternative to bundling your application as a standard WAR to be deployed in Jetty, Jetty is designed to be a software component that can be instantiated and used in a Java program just like any POJO. Put another way, running Jetty in embedded mode means putting an HTTP module into your application, rather than putting your application into an HTTP server.

This tutorial takes you step-by-step from the simplest Jetty server instantiation to running multiple web applications with standards-based deployment descriptors. The source for most of these examples is part of the standard Jetty project.

Overview

To embed a Jetty server the following steps are typical and are illustrated by the examples in this tutorial:

1. Create a [Server](#) instance.
2. Add/Configure [Connectors](#).
3. Add/Configure [Handlers](#) and/or [Contexts](#) and/or [Servlets](#).
4. Start the Server.
5. Wait on the server or do something else with your thread.

Creating the Server

The following code from SimplestServer.java instantiates and runs the simplest possible Jetty server:

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/SimplestServer.java
```

This runs an HTTP server on port 8080. It is not a very useful server as it has no handlers, and thus returns a 404 error for every request.

Using Handlers

To produce a response to a request, Jetty requires that you set a [Handler](#) on the server. A handler may:

- Examine/modify the HTTP request.
- Generate the complete HTTP response.
- Call another Handler (see `HandlerWrapper`).
- Select one or many Handlers to call (see `HandlerCollection`).

HelloWorld Handler

The following code based on `HelloHandler.java` shows a simple hello world handler:

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/HelloHandler.java
```

The parameters passed to the handle method are:

- `target`—the target of the request, which is either a URI or a name from a named dispatcher.
- `baseRequest`—the Jetty mutable request object, which is always unwrapped.
- `request`—the immutable request object, which may have been wrapped by a filter or servlet.
- `response`—the response, which may have been wrapped by a filter or servlet.

The handler sets the response status, content-type, and marks the request as handled before it generates the body of the response using a writer.

Running HelloWorldHandler

To allow a Handler to handle HTTP requests, you must add it to a Server instance. The following code from `OneHandler.java` shows how a Jetty server can use the `HelloWorld` handler:

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/OneHandler.javemain
```

One or more handlers do all request handling in Jetty. Some handlers select other specific handlers (for example, a `ContextHandlerCollection` uses the context path to select a `ContextHandler`); others use application logic to generate a response (for example, the `ServletHandler` passes the request to an application `Servlet`), while others do tasks unrelated to generating the response (for example, `RequestLogHandler` or `StatisticsHandler`).

Later sections describe how you can combine handlers like aspects. You can see some of the handlers available in Jetty in the [org.eclipse.jetty.server.handler](#) package.

Handler Collections and Wrappers

Complex request handling is typically built from multiple Handlers that you can combine in various ways. Jetty has several implementations of the [HandlerContainer](#) interface:

HandlerCollection

Holds a collection of other handlers and calls each handler in order. This is useful for combining statistics and logging handlers with the handler that generates the response.

HandlerList

A Handler Collection that calls each handler in turn until either an exception is thrown, the response is committed or the request.isHandled() returns true. You can use it to combine handlers that conditionally handle a request, such as calling multiple contexts until one matches a virtual host.

HandlerWrapper

A Handler base class that you can use to daisy chain handlers together in the style of aspect-oriented programming. For example, a standard web application is implemented by a chain of a context, session, security and servlet handlers.

ContextHandlerCollection

A specialized HandlerCollection that uses the longest prefix of the request URI (the contextPath) to select a contained ContextHandler to handle the request.

Scoped Handlers

Much of the standard Servlet container in Jetty is implemented with HandlerWrappers that daisy chain handlers together: ContextHandler to SessionHandler to SecurityHandler to ServletHandler. However, because of the nature of the servlet specification, this chaining cannot be a pure nesting of handlers as the outer handlers sometimes need information that the inner handlers process. For example, when a ContextHandler calls some application listeners to inform them of a request entering the context, it must already know which servlet the ServletHandler will dispatch the request to so that the servletPath method returns the correct value.

The HandlerWrapper is specialized to the [ScopedHandler](#) abstract class, which supports a daisy chain of scopes. For example if a ServletHandler is nested within a ContextHandler, the order and nesting of execution of methods is:

```
Server.handle(...)  
  ContextHandler.doScope(...)  
    ServletHandler.doScope(...)  
      ContextHandler.doHandle(...)  
        ServletHandler.doHandle(...)  
          SomeServlet.service(...)
```

Thus when the ContextHandler handles the request, it does so within the scope the ServletHandler has established.

Resource Handler

The [FileServer example](#) shows how you can use a ResourceHandler to serve static content from the current working directory:

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/  
main/java/org/eclipse/jetty/embedded/FileServer.java
```

Notice that a HandlerList is used with the ResourceHandler and a DefaultHandler, so that the DefaultHandler generates a good 404 response for any requests that do not match a static resource.

Embedding Connectors

In the previous examples, the Server instance is passed a port number and it internally creates a default instance of a Connector that listens for requests on that port. However, often when embedding Jetty it is desirable to explicitly instantiate and configure one or more Connectors for a Server instance.

One Connector

The following example, [OneConnector.java](#), instantiates, configures, and adds a single HTTP connector instance to the server:

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/OneConnector.java
```

In this example the connector handles the HTTP protocol, as that is the default for the [ServerConnector](#) class.

Many Connectors

When configuring multiple connectors (for example, HTTP and HTTPS), it may be desirable to share configuration of common parameters for HTTP. To achieve this you need to explicitly configure the [ServerConnector](#) class with [ConnectionFactory](#) instances, and provide them with common HTTP configuration.

The [ManyConnectors example](#), configures a server with two [ServerConnector](#) instances: the http connector has a [HTTPConnectionFactory](#) instance; the https connector has a [SslConnectionFactory](#) chained to a [HttpConnectionFactory](#). Both [HttpConnectionFactory](#)s are configured based on the same [HttpConfiguration](#) instance, however the HTTPS factory uses a wrapped configuration so that a [SecureRequestCustomizer](#) can be added.

SPDY Connectors

The [SPDYConnector example](#) is a similar to the HTTPS connector except that the [SslConnectionFactory](#) is chained to the [NPNConnectionFactory](#) that negotiates whether the next protocol is to be SPDY/2, SPDY/3 or HTTPS.

Embedding Servlets

[Servlets](#) are the standard way to provide application logic that handles HTTP requests. Servlets are similar to a Jetty Handler except that the request object is not mutable and thus cannot be modified. Servlets are handled in Jetty by a [ServletHandler](#). It uses standard path mappings to match a Servlet to a request; sets the requests servletPath and pathInfo; passes the request to the servlet, possibly via Filters to produce a response.

The [MinimalServlets example](#) creates a [ServletHandler](#) instance and configures a single [HelloServlet](#):

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/MinimalServlets.java
```

Embedding Contexts

A [ContextHandler](#) is a [ScopedHandler](#) that responds only to requests that have a URI prefix that matches the configured context path. Requests that match the context path have their path methods updated accordingly and the contexts scope is available, which optionally may include:

- A Classloader that is set as the Thread context classloader while request handling is in scope.
- A set of attributes that is available via the [ServletContext](#) API.
- A set of init parameters that is available via the [ServletContext](#) API.
- A base Resource which is used as the document root for static resource requests via the [ServletContext](#) API.
- A set of virtual host names.

The following [OneContext example](#) shows a context being established that wraps the [HelloHandler](#):

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/OneContext.java
```

When many contexts are present, you can embed a ContextHandlerCollection to efficiently examine a request URI to then select the matching ContextHandler(s) for the request. The [ManyContexts example](#) shows how many such contexts you can configure:

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/ManyContexts.java
```

Embedding ServletContexts

A [ServletContextHandler](#) is a specialization of ContextHandler with support for standard sessions and Servlets. The following [OneServletContext example](#) instantiates a [DefaultServlet](#) to serve static content from /tmp/ and a [DumpServlet](#) that creates a session and dumps basic details about the request:

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/OneServletContext.java
```

Embedding Web Applications

A [WebAppContext](#) is an extension of a ServletContextHandler that uses the [standard layout](#) and web.xml to configure the servlets, filters and other features from a web.xml and/or annotations. The following [OneWebApp example](#) configures the Jetty test webapp. Web applications can use resources the container provides, and in this case a LoginService is needed and also configured:

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/OneWebApp.java
```

Like Jetty XML

The typical way to configure an instance of the Jetty server is via jetty.xml and associated configuration files. However the Jetty XML configuration format is just a simple rendering of what you can do in code; it is very simple to write embedded code that does precisely what the jetty.xml configuration does. The [LikeJettyXml example](#) following renders in code the behaviour obtained from the configuration files:

- [jetty.xml](#)
- [jetty-jmx.xml](#)
- [jetty-http.xml](#)
- [jetty-https.xml](#)
- [jetty-deploy.xml](#)
- [jetty-stats.xml](#)
- [jetty-requestlog.xml](#)
- [jetty-lowresources.xml](#)
- [test-realm.xml](#)

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/LikeJettyXml.java
```

Embedded Examples

Jetty has a rich history of being embedded into a wide variety of applications. In this section we will walk you through a number of our simple examples under our `embedded-jetty-examples` project in our git repository.

Live Files



These files are pulled directly from our git repository when this document is generated. If the line numbers do not line up feel free to fix this documentation in github and give us a pull request, or at least open an issue to notify us of the discrepancy.

Simple File Server

This example shows how to create a simple file server in Jetty. It is perfectly suitable for test cases where you need an actual web server to obtain a file from, it could easily be configured to serve files from a directory under `src/test/resources`. Note that this does not have any logic for caching of files, either within the server or setting the appropriate headers on the response. It is simply a few lines that illustrate how easy it is to serve out some files.

Example 26.1. [FileServer.java](#)

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/FileServer.java
```

Run it!

After you have started things up you should be able to navigate to `http://localhost:8080/index.html` (assuming one is in the resource base directory) and you are good to go.

Maven Coordinates

To use this example in your project you will need the following maven dependencies declared.

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-server</artifactId>
  <version>${project.version}</version>
</dependency>
```

Split File Server

This example builds on the [Simple File Server](#) to show how chaining multiple ResourceHandlers together can let you aggregate multiple directories to serve content on a single path and how you can link these together with ContextHandlers.

Example 26.2. [SplitFileServer.java](#)

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/
embedded/src/main/java/org/eclipse/jetty/embedded/SplitFileServer.java
```

Run it!

After you have started things up you should be able to navigate to `http://localhost:8090/index.html` (assuming one is in the resource base directory) and you are good to go. Any requests for files will be looked for in the first resource handler, then the second, and so on and so forth.

Maven Coordinates

To use this example as is in your project you will need the following maven dependencies declared. We would recommend not using the toolchain dependency in your actual application.

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-server</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>org.eclipse.jetty.toolchains</groupId>
  <artifactId>jetty-test-helper</artifactId>
  <version>2.2</version>
</dependency>
```

Multiple Connectors

This example shows how to configure Jetty to use multiple connectors, specifically so it can process both http and https requests. Since the meat of this example is the server and connector configuration it only uses a simple HelloHandler but this example should be easily merged with other examples like those deploying servlets or webapps.

Example 26.3. [ManyConnectors.java](#)

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/
examples/embedded/src/main/java/org/eclipse/jetty/embedded/ManyConnectors.java
```

Walkthrough

Start things up! By using the `server.join()` the server thread will join with the current thread. See [Thread.join\(\)](#) for more details.

Maven Coordinates

To use this example in your project you will need the following maven dependencies declared.

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-server</artifactId>
  <version>${project.version}</version>
</dependency>
```

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-security</artifactId>
  <version>${project.version}</version>
</dependency>
```

Embedded SPDY Server

This example shows how to create a server with a SPDY connector in Jetty. Clients (mostly browsers) that have a SPDY client implementation will talk SPDY to that connector. All others will transparently fall back to HTTP.

Example 26.4. [SpdyServer.java](#)

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/
examples/embedded/src/main/java/org/eclipse/jetty/embedded/SpdyServer.java
```

Run it

After you have started things up you should be able to navigate to <http://localhost:8080/index.html> (assuming one is in the resource base directory) and you are good to go.

Maven Coordinates

To use this example in your project you will need the following maven dependencies declared.

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-server</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-spdy</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-deploy</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-util</artifactId>
  <version>${project.version}</version>
</dependency>
```

Secured Hello Handler

This example shows how to wrap one handler with another one that handles security. We have a simple Hello Handler that just return a greeting but add on the restriction that to get this greeting you must authenticate. Another thing to remember is that this example uses the ConstraintSecurityHandler which is what supports the security mappings inside of the servlet api, it could be easier to show just the SecurityHandler usage, but the constraint provides more configuration power. If you don't need that you can drop the Constraint bits and use just the SecurityHandler.

Example 26.5. [SecuredHelloHandler.java](#)

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/SecuredHelloHandler.java
```

Run it!

After you have started things up you should be able to navigate to <http://localhost:8080/index.html> (assuming one is in the resource base directory) and you are good to go.

The Realm Properties File

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/test/resources/realm.properties
```

Maven Coordinates

To use this example in your project you will need the following maven dependencies declared.

```
<dependency>
<groupId>org.eclipse.jetty</groupId>
<artifactId>jetty-server</artifactId>
<version>${project.version}</version>
</dependency>
```

Minimal Servlet

This example shows the bare minimum required for deploying a servlet into jetty. Note that this is strictly a servlet, not a servlet in the context of a web application, that example comes later. This is purely just a servlet deployed and mounted on a context and able to process requests. This example is excellent for situations where you have a simple servlet that you need to unit test, just mount it on a context and issue requests using your favorite http client library (like our Jetty client found in Chapter 29, *HTTP Client*).

Example 26.6. [MinimalServlets.java](#)

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/MinimalServlets.java
```

Walkthrough

Start things up! By using the `server.join()` the server thread will join with the current thread. See [Thread.join\(\)](#) for more details.

It is really simple to create useful servlets for testing behaviors, sometimes you just need a http server to run a unit test against that will return test content and wiring up a servlet like this makes it trivial.

After you have started things up you should be able to navigate to <http://localhost:8080/> and you are good to go.

Maven Coordinates

To use this example in your project you will need the following maven dependencies declared.

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-servlet</artifactId>
  <version>${project.version}</version>
</dependency>
```

Web Application

This example shows how to deploy a simple webapp with an embedded instance of jetty. This is useful when you want to manage the lifecycle of a server programmatically, either within a production application or as a simple way to deploying and debugging a full scale application deployment. In many ways it is easier then traditional deployment since you control the classpath yourself, making this easy to wire up in a test case in maven and issue requests using your favorite http client library (like our Jetty client found in Chapter 29, *HTTP Client*).

Example 26.7. [OneWebApp.java](#)

```
http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git/plain/examples/embedded/src/main/java/org/eclipse/jetty/embedded/OneWebApp.java
```

Run it!

After you have started things up you should be able to navigate to <http://localhost:8080/> and you are good to go.

Maven Coordinates

To use this example in your project you will need the following maven dependencies declared.

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-webapp</artifactId>
  <version>${project.version}</version>
</dependency>
```

Adding Examples

If you would like to add an example to this list, fork the documentation project from github (see the blue bar at the bottom of this page) and add the new page. Feel free to add the example contents directly as a programlisting using language= "java" and we will take it from there.

If you feel and example is missing, feel free to open a bug to ask for it. No guarantees, but the more helpful and demonstrative it is the better.

Chapter 27. Debugging

Table of Contents

Options	339
Enable remote debugging	339
Debugging With Eclipse	340
Debugging With IntelliJ	342

Options

Given how flexible Jetty is in how it can be configured and deployed into development and production, there exists a wealth of different options for debugging your application in your favorite environment. In this section we will gather up some of these different options available and explain how you can use them. If you would like to contribute to this section simply fork the repository and contribute the information, or open a github or bugzilla issue with the information and we'll bring it over.

Enable remote debugging

Remote Debugging

If you have a web application deployed into Jetty you can interact with it remotely from a debugging perspective easily. The basics are that you must start up the remote JVM with additional parameters and then start up a remote debugging session in Eclipse for the webapp in question. This is easily accomplished.



Note

This example assumes you are deploying your web application into the jetty-distribution.

Starting Jetty

Assuming you have your webapp deployed into jetty, there are two different ways to approach this:

Via command line

Add the required parameters on the commandline like so.

```
$ java -Xdebug -agentlib:jdwp=transport=dt_socket,address=9999,server=y,suspend=n -jar start.jar
```

Via start.ini

This approach is best used if you want to debug a particular jetty-distribution and not have to remember the commandline incantations.

1. Edit the `start.ini` and uncomment the `--exec` line, this is required if you are adding jvm options to the `start.ini` file as jetty-start must generate the classpath required and fork a new jvm.
2. Add the parameters mentioned above in the Command Line option so your `start.ini` looks like this:

```

=====
# Configure JVM arguments.
# If JVM args are include in an ini file then --exec is needed
# to start a new JVM from start.jar with the extra args.
# If you wish to avoid an extra JVM running, place JVM args
# on the normal command line and do not use --exec
#-----
--exec
-Xdebug
-agentlib:jdwp=transport=dt_socket,address=9999,server=y,suspend=n
# -Xmx2000m
# -Xmn512m
# -XX:+UseConcMarkSweepGC
# -XX:ParallelCMSThreads=2
# -XX:+CMSClassUnloadingEnabled
# -XX:+UseCMSCompactAtFullCollection
# -XX:CMSInitiatingOccupancyFraction=80
# -verbose:gc
# -XX:+PrintGCDetails
# -XX:+PrintGCDateStamps
# -XX:+PrintGCTimeStamps
# -XX:+PrintGCDetails
# -XX:+PrintTenuringDistribution
# -XX:+PrintCommandLineFlags
# -XX:+DisableExplicitGC

```

Uncomment any other jvm environmental options you so desire for your debugging session.

3. Regardless of the option chosen, you should see the following lines at the top of your jetty-distribution startup.

```
Listening for transport dt_socket at address: 9999
```

Linking with your IDE

Refer to the documentation for your ide:

- the section called “Debugging With Eclipse”
- the section called “Debugging With IntelliJ”

Debugging With Eclipse

There are a number of options available to debug your application in Eclipse.

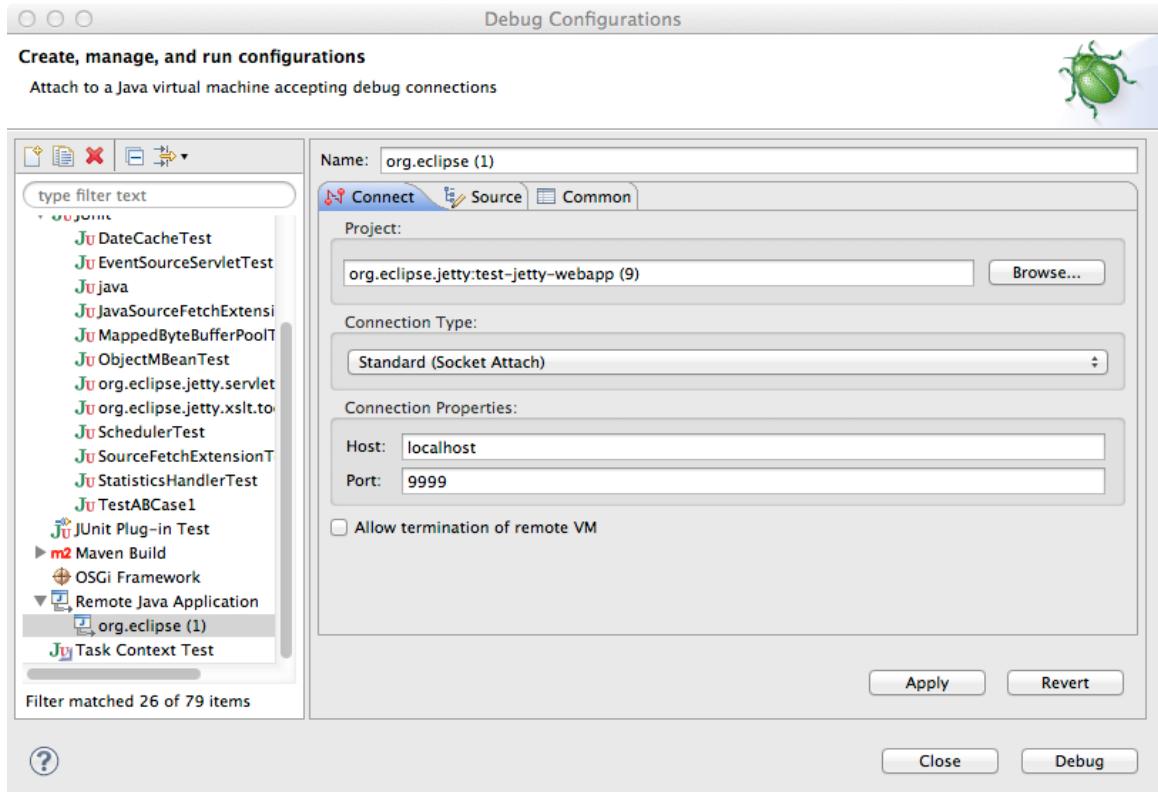
If not done already prepare your application for remote debugging as described here: the section called “Enable remote debugging”

Linking with Eclipse

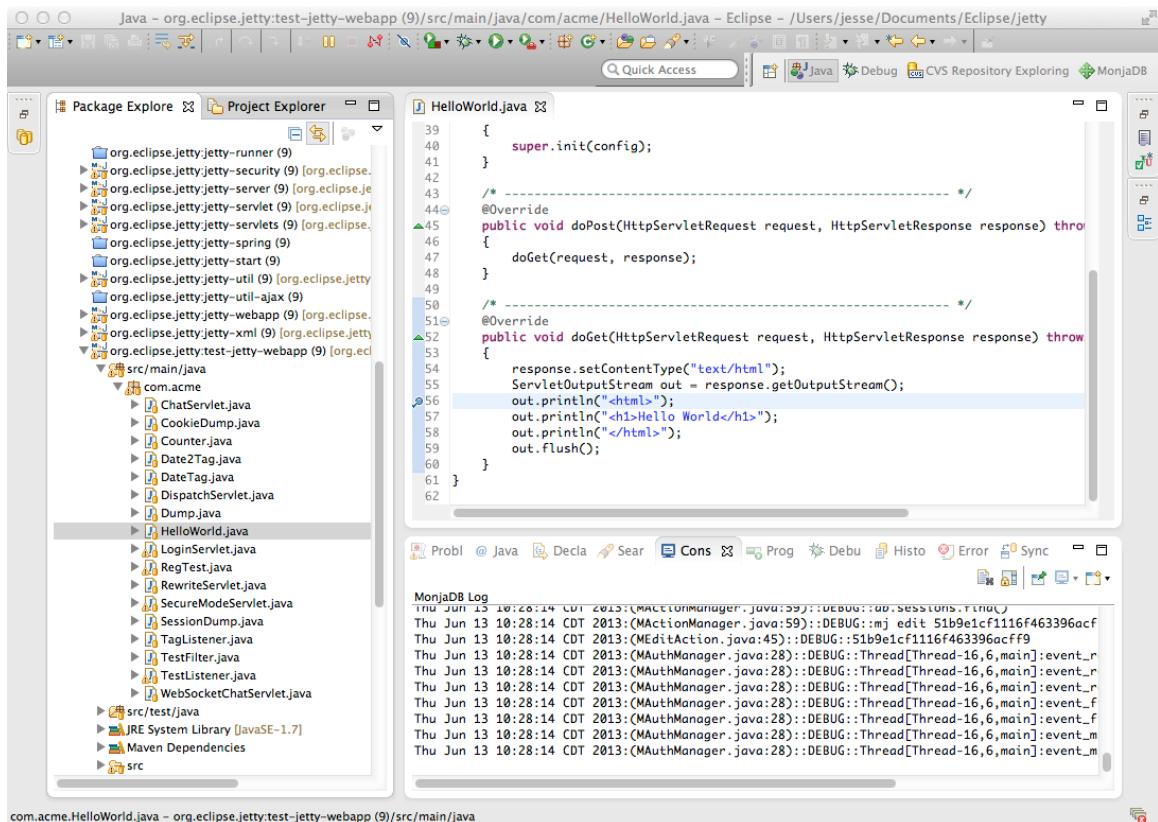
Next we need to link the Eclipse project with the deployed webapp.

1. Within Eclipse, right-click on the project containing the webapp deployed into jetty and select **Debug -> Debug Configurations** and create a new configuration of**Remote Java Application**. Make sure the port you choose is the same as the one you added in the section called “Enable remote debugging”.

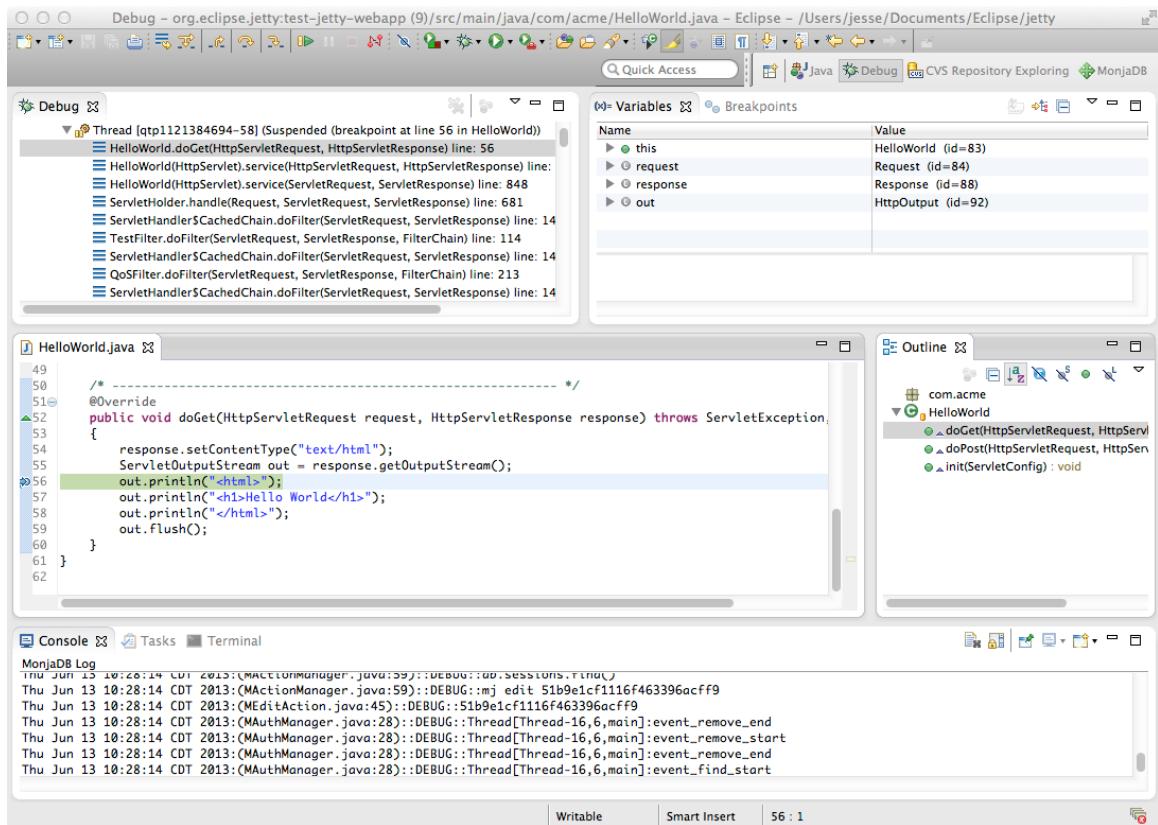
Debugging



2. Next in your webapp you can set a breakpoint within a servlet which when it is tripped will halt the remote jvm's processing thread to await for debugging commands from your Eclipse instance.



3. Accessing that servlet within your browser, pointed at your remote debug configurated jetty-distribution, should transition your Eclipse instance to the standard Debug view.



Within Eclipse

Since Jetty can be incredibly simple to embed, many people choose to create a small main method which they can launch directly within Eclipse in order to more easily debug their entire application. The best place to get started on this approach is to look through the section called “Embedding Jetty” and the the section called “Embedded Examples” sections.

Once you have a main method defined in order to launch your application, right-click on the source file and select **Debug As -> Java Application**. In your **Console** tab within Eclipse you should see your application startup and once it has completed startup you should be able to configure breakpoints and hit the Jetty instance as normal via your web browser.



Logging

You can easily configure logging through a `jetty-logging.properties` file. If this file is on your classpath then Jetty will use it for configuring logging, we use this approach extensively throughout Jetty development and it makes life ever so much easier. You can see this in action in the the section called “The `jetty-logging.properties` file” section.

Debugging With IntelliJ

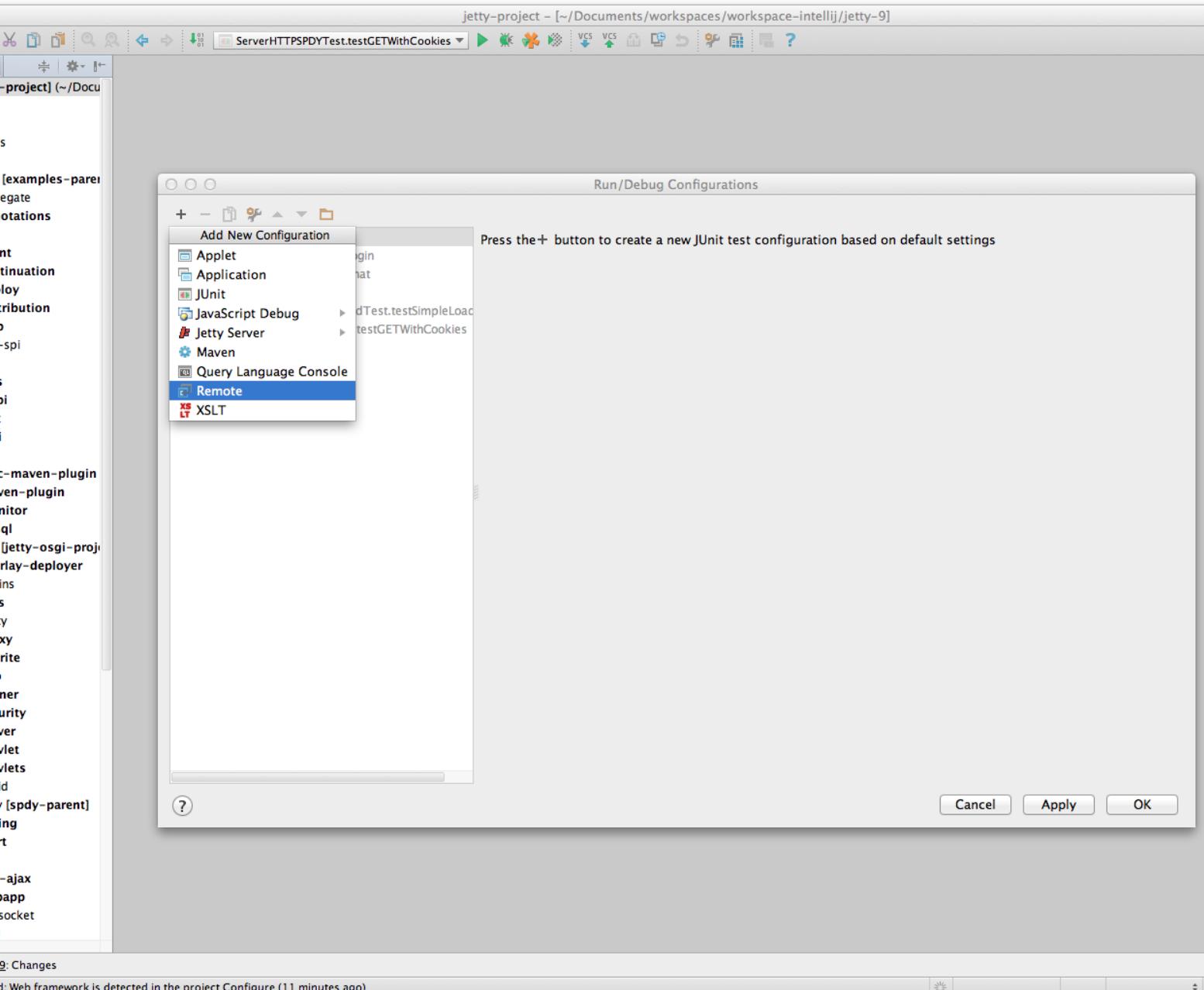
There are a number of options available to debug your application in IntelliJ.

If not done already prepare your application for remote debugging as described here: the section called “Enable remote debugging”

Linking with IntelliJ

Next we need to link the IntelliJ project with the deployed webapp.

1. Within IntelliJ, open the project containing the webapp deployed into jetty that you want to debug. Select **Run -> Edit Configurations**. Add a new configuration by clicking the "+" icon. Choose **Remote**. Make sure the port you choose is the same as the one you added in the section called "Enable remote debugging".



2. Next in your webapp you can set a breakpoint within a servlet which when it is tripped will halt the remote jvm's processing thread to await for debugging commands from your IntelliJ instance. To set a breakpoint, simply open the servlet or any other class you want to debug and click left to the line you want to set the breakpoint at (where the red dot is on the next screenshot). The red dot and red background on the line mark the breakpoint.

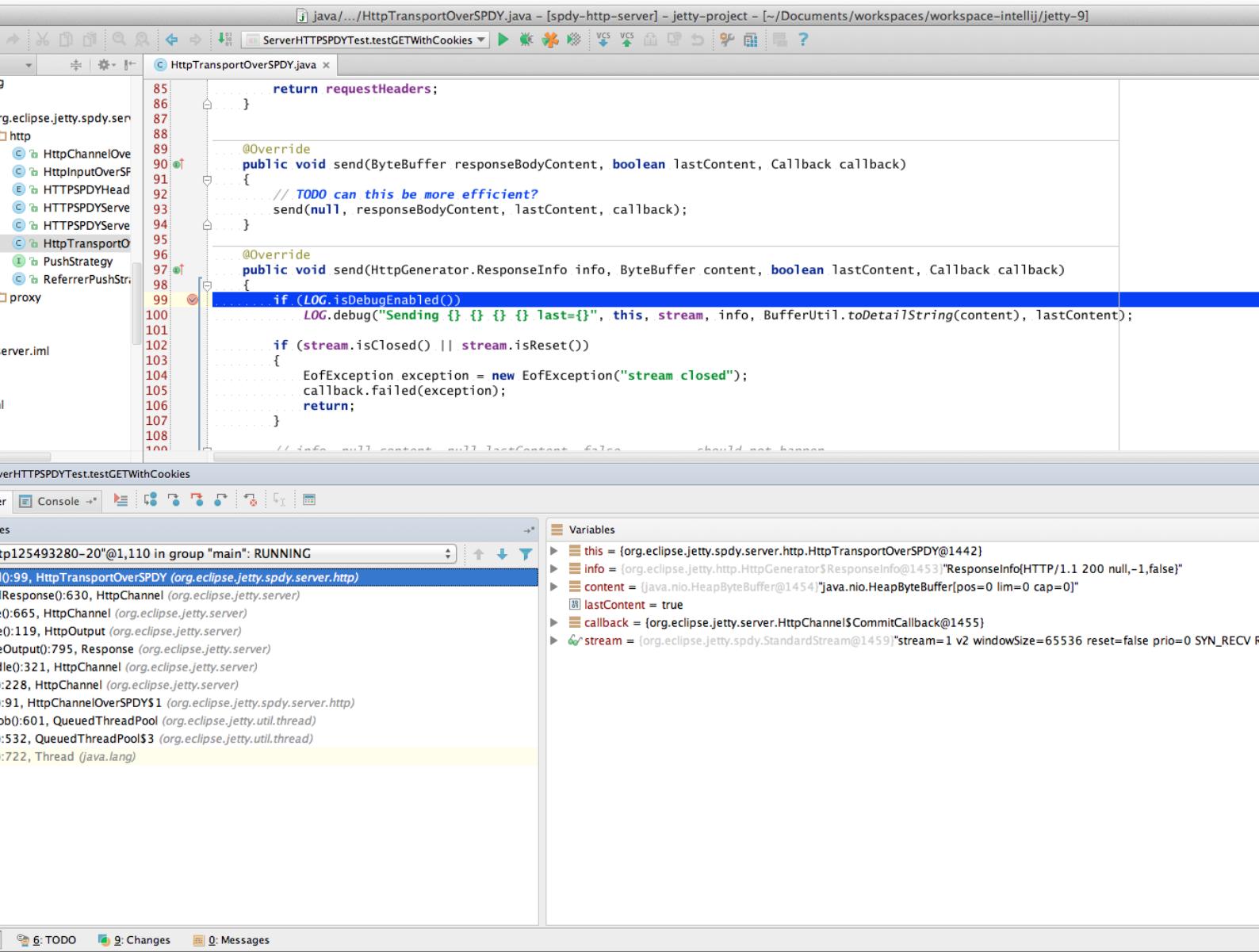
```

91     {
92         // TODO can this be more efficient?
93         send(null, responseBodyContent, lastContent, callback);
94     }
95
96     @Override
97     public void send(HttpGenerator.ResponseInfo info, ByteBuffer content, boolean lastContent, Callback callback)
98     {
99         if (LOG.isDebugEnabled())
100             LOG.debug("Sending {} {} {} {} last={}", this, stream, info, BufferUtil.toDetailString(content), lastContent);
101
102         if (stream.isClosed() || stream.isReset())
103         {
104             EofException exception = new EofException("stream closed");
105             callback.failed(exception);
106             return;
107         }
108
109         // info==null content==null lastContent==false      should not happen
110         // info==null content==null lastContent==true       signals no more content - complete
111         // info==null content!=null lastContent==false      send data on committed response
112         // info==null content!=null lastContent==true       send last data on committed response - complete
113         // info!=null content==null lastContent==false      reply, commit
114         // info!=null content==null lastContent==true       reply, commit and complete
115         // info!=null content!=null lastContent==false      reply, commit with content
116         // info!=null content!=null lastContent==true       reply, commit with content and complete
117
118         boolean hasContent = BufferUtil.hasContent(content);
119
120         if (info != null)
121         {
122             if (!committed.compareAndSet(false, true))
123             {
124                 StreamException exception = new StreamException(stream.getId(), StreamStatus.PROTOCOL_ERROR,
125                                                               "Stream already committed!");
126                 callback.failed(exception);
127                 LOG.warn("Committed response twice.", exception);
128                 return;
129             }
130             short version = stream.getSession().getVersion();
131             Fields headers = new Fields();
132
133             HttpVersion httpVersion = HttpVersion.HTTP_1_1;
134             headers.put(HTTPSPDYHeader.VERSION.name(version), httpVersion.toString());
135
136             int status = info.getStatus();
137             StringBuilder httpStatus = new StringBuilder().append(status);
138             String reason = info.getReason();
139             if (reason == null)
140                 reason = HttpStatus.getMessage(status);
141             if (reason != null)
142                 httpStatus.append(" ").append(reason);
143             headers.put(HTTPSPDYHeader.STATUS.name(version), httpStatus.toString());
144             LOG.debug("HTTP <{} {}>", httpVersion, httpStatus);
145         }
146     }

```

9: Changes
detected: Web framework is detected in the project Configure (18 minutes ago)
152:21 LF UTF-8 Git: master

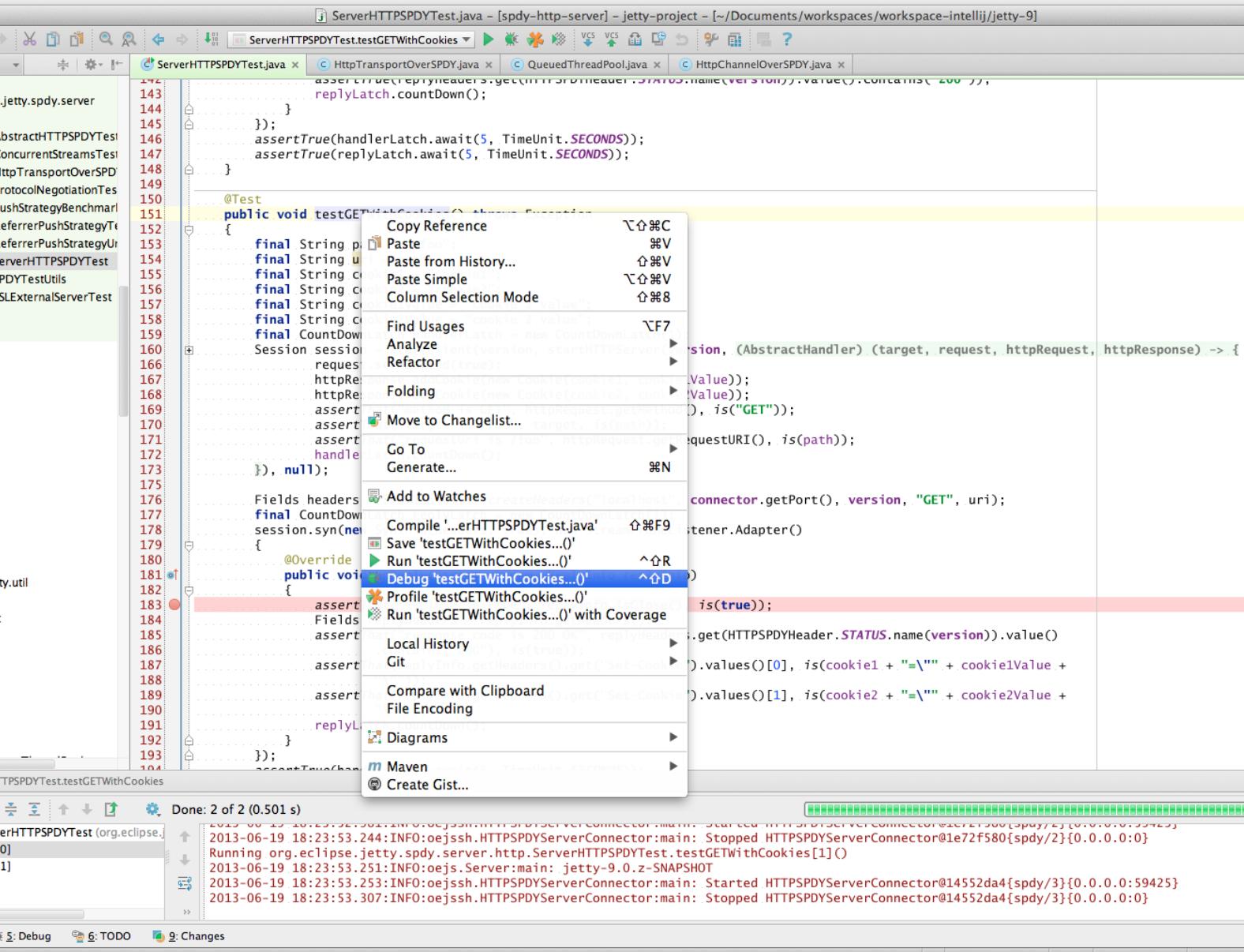
3. Accessing that servlet within your browser, pointed at your remote debug configurated jetty-distribution, should transition your IntelliJ instance to the standard debugger view.



Within IntelliJ

Since Jetty can be incredibly simple to embed, many people choose to create a small `main` method which they can launch directly within IntelliJ in order to more easily debug their entire application. The best place to get started on this approach is to look through the section called “Embedding Jetty” and the the section called “Embedded Examples” sections.

Once you have a `main` method defined in order to launch your application, open the source file and right-click the `main` method. Select **Debug** or simply hit **CTRL+SHIFT+D**. In your **Console** tab within IntelliJ you should see your application startup and once it has completed startup you should be able to configure breakpoints and hit the Jetty instance as normal via your web browser. The same thing works for unit tests. Instead of the `main` method run debug on the test method you want to debug.



Debugging in IntelliJ is extremely powerful. For example it's possible to have conditional breakpoints that only trigger a break if the configured conditions are met. Have a look at the various tutorials in the internet or the [IntelliJ documentation](#) for further details.



Logging

You can easily configure logging through a `jetty-logging.properties` file. If this file is on your classpath then Jetty will use it for configuring logging, we use this approach extensively throughout Jetty development and it makes life ever so much easier. You can see this in action in the the section called “The `jetty-logging.properties` file” section.

Chapter 28. Frameworks

Table of Contents

Spring Setup	347
OSGI	348
Weld	362
Metro	364

Spring Setup

You can assemble and configure Jetty in code or with almost any IoC style framework including Spring. If all you want to do is setup a Jetty server in your stock Spring usage, simply look at the xml snippet below as an example. If you want to replace the jetty-xml being used to start the normal Jetty distribution with spring, you may do so however currently it will not leverage the rest of the module system. If you are interested in doing this as anything other then a novelty contact us and let us know and we can work out a suitable jetty.base example.

Downloading the Jetty-Spring Module and Dependencies

The jetty-spring module is available as a Maven artifact. You can obtain it through the following maven coordinates:

```
<groupId>org.eclipse.jetty</groupId>
<artifactId>jetty-spring</artifactId>
<version>${project.version}</version>
```

Using jetty-spring requires a couple of transitive dependencies so if you are not using a build system that handles such things you will need to grab a couple of other artifacts as well.

```
$ wget --user-agent=other http://repo2.maven.org/maven2/org/eclipse/jetty/jetty-spring/
${project.version}/jetty-spring-${project.version}.jar
$ wget --user-agent=other http://repo2.maven.org/maven2/org/springframework/spring/2.5.6/
spring-2.5.6.jar
$ wget --user-agent=other http://repo2.maven.org/maven2/commons-logging/commons-
logging/1.1.1/commons-logging-1.1.1.jar
```

Using Spring to Configure Jetty

Configuring Jetty via Spring is simply a matter of calling the API as Spring beans. The following is an example mimicing the default jetty startup configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
```

```

<!-- ===== -->
<!-- Configure the Jetty Server with Spring -->
<!-- This file is the similar to jetty.xml, but written in spring -->
<!-- XmlBeanFactory format. -->
<!-- ===== -->

<beans>
  <bean id="contexts" class="org.eclipse.jetty.server.handler.ContextHandlerCollection"/>
  <bean id="server" name="Main" class="org.eclipse.jetty.server.Server" init-
method="start" destroy-method="stop">
    <constructor-arg>
      <bean id="threadPool" class="org.eclipse.jetty.util.thread.QueuedThreadPool">
        <property name="minThreads" value="10"/>
        <property name="maxThreads" value="50"/>
      </bean>
    </constructor-arg>
    <property name="connectors">
      <list>
        <bean id="connector" class="org.eclipse.jetty.server.ServerConnector">
          <constructor-arg ref="server"/>
          <property name="port" value="8080"/>
        </bean>
      </list>
    </property>
    <property name="handler">
      <bean id="handlers" class="org.eclipse.jetty.server.handler.HandlerCollection">
        <property name="handlers">
          <list>
            <ref bean="contexts"/>
            <bean id="defaultHandler" class="org.eclipse.jetty.server.handler.DefaultHandler"/>
          </list>
        </property>
      </bean>
    </property>
    <property name="beans">
      <list>
        <bean id="deploymentManager" class="org.eclipse.jetty.deploy.DeploymentManager">
          <property name="contexts" ref="contexts"/>
          <property name="appProviders">
            <list>
              <bean id="webAppProvider" class="org.eclipse.jetty.deploy.providers.WebAppProvider">
                <property name="monitoredDirName" value="webapps"/>
                <property name="scanInterval" value="1"/>
                <property name="extractWars" value="true"/>
              </bean>
            </list>
          </property>
        </bean>
      </list>
    </property>
  </bean>
</beans>

```

OSGI

Introduction

The Jetty OSGi infrastructure provides a Jetty container inside an OSGi container. Traditional JavaEE webapps can be deployed, in addition to Jetty ContextHandlers, along with OSGi web bundles. In addition, the infrastructure also supports the OSGi HttpService interface.

General Setup

All of the Jetty jars contain manifest entries appropriate to ensure that they can be deployed into an OSGi container as bundles. You will need to install some jetty jars into your OSGi container. You can

always find the jetty jars either in the maven central repository, or you can download a distribution of jetty. Here's the minimal set:

Table 28.1. Bundle Name Mapping

Jar	Bundle Symbolic Name
jetty-util	org.eclipse.jetty.util
jetty-http	org.eclipse.jetty.http
jetty-io	org.eclipse.jetty.io
jetty-security	org.eclipse.jetty.security
jetty-server	org.eclipse.jetty.server
jetty-servlet	org.eclipse.jetty.servlet
jetty-webapp	org.eclipse.jetty.webapp
jetty-deploy	org.eclipse.jetty.deploy
jetty-xml	org.eclipse.jetty.xml
javax.servlet-api (minimum version 3.1.0)	javax.servlet-api
jetty-schemas (minimum version 3.1.M0)	org.eclipse.jetty.schemas

The Jetty OSGi Container

The jetty-osgi-boot jar

Now that you have the basic set of Jetty jars installed, you can install the [jetty-osgi-boot.jar](#) bundle, downloadable from the maven central repo [here](#).

This bundle will instantiate and make available the Jetty OSGi container.

Customizing the Jetty Container

Before going ahead with the install, you may want to customize the Jetty container. In general this is done by a combination of System properties and the usual jetty xml configuration files. The way you define the System properties will depend on which OSGi container you are using, so ensure that you are familiar with how to set them for your environment. In the following examples, we will assume that the OSGi container allows us to set System properties as simple name=value pairs.

The available System properties are:

jetty.port

If not specified, this defaults to the usual jetty port of 8080.

jetty.home

Either this property *or* the **jetty.home.bundle** must be specified. This property should point to a file system location that has an etc/ directory containing xml files to configure the Jetty container on startup. For example:

```
jetty.home=/opt/custom/jetty
```

Where /opt/custom/jetty contains:

```
etc/jetty.xml
etc/jetty-selector.xml
```

```
etc/jetty-deployer.xml
etc/jetty-special.xml
```

jetty.home.bundle

Either this property or the **jetty.home** property must be specified. This property should specify the symbolic name of a bundle which contains a directory called `jettyhome/`. The `jettyhome/` directory should have a subdirectory called `etc/` that contains the xml files to be applied to Jetty on startup. The `jetty-osgi-boot.jar` contains a `jettyhome/` directory with a default set of xml configuration files. Here's how you would specify it:

```
jetty.home.bundle=org.eclipse.jetty.osgi.boot
```

Here's a partial listing of that jar that shows you the names of the xml files contained within it:

```
META-INF/MANIFEST.MF
jettyhome/etc/jetty.xml
jettyhome/etc/jetty-deployer.xml
jettyhome/etc/webdefault.xml
jettyhome/etc/jetty-selector.xml
```

jetty.etc.config.urls

This specifies the paths of the xml files that are to be used. If not specified, they default to:

```
etc/jetty.xml,etc/jetty-selector.xml,etc/jetty-deployer.xml
```

Note that the paths can either be relative or absolute, or a mixture. If the path is relative, it is resolved against either **jetty.home** or **jetty.home.bundle**, whichever was specified. You can use this ability to mix and match jetty configuration files to add functionality, such as adding in a https connector. Here's an example of adding a https connector, using the relevant files from the `jetty-distribution` in `/opt/jetty`:

```
etc/jetty.xml, etc/jetty-selector.xml, /opt/jetty/etc/jetty-ssl.xml, /opt/jetty/etc/jetty-https.xml, etc/jetty-deployer.xml
```

Note that regardless of whether you set the **jetty.home** or **jetty.home.bundle** property, when Jetty executes the configuration files, it will set an appropriate value for **jetty.home** so that references in xml files to `<property name="jetty.home">` will work. Be careful, however, if you are mixing and matching relative and absolute configuration file paths: the value of **jetty.home** is determined from the resolved location of the *relative* files only.

The Jetty Container as an OSGi Service

You can now go ahead and deploy the `jetty-osgi-boot.jar` into your OSGi container. A Jetty Server instance will be created, the xml config files applied to it, and then published as an OSGi service. Normally, you will not need to interact with this service instance, however you can retrieve a reference to it using the usual OSGi API:

```
org.osgi.framework.BundleContext bc;
org.osgi.framework.ServiceReference ref =
bc.getServiceReference("org.eclipse.jetty.server.Server");
```

The Server service has a couple of properties associated with it that you can retrieve using the `org.osgi.framework.ServiceReference.getProperty(String)` method:

managedServerName

The Jetty Server instance created by the `jetty-osgi-boot.jar` will be called "defaultJettyServer"

jetty.etc.config.urls

The list of xml files resolved from either **jetty.home** or **jetty.home.bundle/jettyhome**

Adding More Jetty Servers

As we have seen in the previous section, the jetty-osgi-boot code will create an `org.eclipse.jetty.server.Server` instance, apply the xml configuration files specified by **jetty.etc.config.urls** System property to it, and then register it as an OSGi Service. The name associated with this default instance is "defaultJettyServer".

You can create other Server instances, register them as OSGi Services, and the jetty-osgi-boot code will notice them, and configure them so that they can deploy ContextHandlers and webapp bundles. When you deploy webapps or ContextHandlers as bundles or Services (see sections below) you can target them to be deployed to a particular server instance via the Server's name.

Here's an example of how to create a new Server instance and register it with OSGi so that the jetty-osgi-boot code will find it and configure it so it can be a deployment target:

```
public class Activator implements BundleActivator
{
    public void start(BundleContext context) throws Exception
    {
        Server server = new Server();
        //do any setup on Server in here
        String serverName = "fooServer";
        Dictionary serverProps = new Hashtable();
        //define the unique name of the server instance
        serverProps.put("managedServerName", serverName);
        serverProps.put("jetty.port", "9999");
        //let Jetty apply some configuration files to the Server instance
        serverProps.put("jetty.etc.config.urls", "file:/opt/jetty/etc/jetty.xml,file:/opt/jetty/etc/jetty-selector.xml,file:/opt/jetty/etc/jetty-deployer.xml");
        //register as an OSGi Service for Jetty to find
        context.registerService(Server.class.getName(), server, serverProps);
    }
}
```

Now that we have created a new Server called "fooServer", we can deploy webapps and ContextHandlers as Bundles or Services to it (see below for more information on this). Here's an example of deploying a webapp as a Service and targeting it to the "fooServer" Server we created above:

```
public class Activator implements BundleActivator
{
    public void start(BundleContext context) throws Exception
    {
        //Create a webapp context as a Service and target it at the "fooServer" Server
        //instance
        WebAppContext webapp = new WebAppContext();
        Dictionary props = new Hashtable();
        props.put("war", ".");
        props.put("contextPath", "/acme");
        props.put("managedServerName", "fooServer");
        context.registerService(ContextHandler.class.getName(), webapp, props);
    }
}
```

Deploying Bundles as Webapps

The Jetty OSGi container listens for the installation of bundles, and will automatically attempt to deploy any that appear to be webapps.

Any of the following criteria are sufficient for Jetty to deploy the bundle as a webapp:

Bundle contains a WEB-INF/web.xml file

If the bundle contains a web descriptor, then it is automatically deployed. This is an easy way to deploy classic JavaEE webapps.

Bundle MANIFEST contains Jetty-WarFolderPath

This is the location within the bundle of the webapp resources. Typically this would be used if the bundle is not a pure webapp, but rather the webapp is a component of the bundle. Here's an example of a bundle where the resources of the webapp are not located at the root of the bundle, but rather inside the subdirectory web/ :

MANIFEST:

```
Bundle-Name: Web
Jetty-WarFolderPath: web
Import-Package: javax.servlet;version="2.6.0",
    javax.servlet.resources;version="2.6.0"
Bundle-SymbolicName: com.acme.sample.web
```

Bundle contents:

```
META-INF/MANIFEST.MF
web/index.html
web/foo.html
web/WEB-INF/web.xml
com/acme/sample/web/MyStuff.class
com/acme/sample/web/MyOtherStuff.class
```

Bundle MANIFEST contains Web-ContextPath

This header can be used in conjunction with either of the two preceding headers to control the context path to which the webapp is deployed, or alone to identify that the bundle's contents should be published as a webapp. This header is part of the RFC-66 specification for using webapps with OSGi. Here's an example based on the previous one where we use the Web-ContextPath header to set its deployment context path to be "/sample" :

MANIFEST:

```
Bundle-Name: Web
Jetty-WarFolderPath: web
Web-ContextPath: /sample
Import-Package: javax.servlet;version="2.6.0",
    javax.servlet.resources;version="2.6.0"
Bundle-SymbolicName: com.acme.sample.web
```

You can also define extra headers in your bundle MANIFEST that help customize the web app to be deployed:

Jetty-defaultWebXmlFilePath

The location of a webdefault.xml file to apply to the webapp. The location can be either absolute (either absolute path or file: url), or relative (in which case it is interpreted as relative to the bundle root). Defaults to the webdefault.xml file built into the Jetty OSGi container.

Jetty-WebXmlFilePath

The location of the web.xml file. The location can be either absolute (either absolute path or file: url), or relative (in which case it is interpreted as relative to the bundle root). Defaults to WEB-INF/web.xml

Jetty-extraClassPath

A classpath of additional items to add to the webapp's classloader.

Jetty-bundleInstall

The path to the base folder that overrides the computed bundle installation - mostly useful for those OSGi frameworks that unpack bundles by default.

Require-TldBundle

A comma separated list of bundle symbolic names of bundles containing TLDs that this webapp depends upon.

managedServerName

The name of the Server instance to which to deploy this webapp bundle. If not specified, defaults to the default Server instance called "defaultJettyServer".

Determining the Context Path for a Webapp Bundle

As we have seen in the previous section, if the bundle MANIFEST contains the RFC-66 header **Web-ContextPath**, Jetty will use that as the context path. If the MANIFEST does not contain that header, then Jetty will concoct a context path based on the last element of the bundle's location (by calling `Bundle.getLocation()` after stripping off any file extensions).

For example, suppose we have a bundle whose location is:

```
file:///some/where/over/the/rainbow/oz.war
```

The corresponding synthesized context path would be:

```
/oz
```

Extra Properties Available for Webapp Bundles

You can further customize your webapp by including a jetty context xml file that is applied to the webapp. This xml file must be placed in META-INF of the bundle, and must be called `jetty-webapp-context.xml`.

Here's an example of a webapp bundle listing containing such a file:

```
META-INF/MANIFEST.MF
META-INF/jetty-webapp-context.xml
web/index.html
web/foo.html
web/WEB-INF/web.xml
com/acme/sample/web/MyStuff.class
com/acme/sample/web/MyOtherStuff.class
```

Here's an example of the contents of a META-INF/jetty-webapp-context.xml file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
```

```
<Set name="defaultsDescriptor"><Property name="bundle.root"/>META-INF/webdefault.xml</Set>
</Configure>
```

As you can see, it is a normal context xml file used to set up a webapp. There are, however, some additional useful properties that can be referenced

Server

This is a reference to the Jetty org.eclipse.jetty.server.Server instance to which the webapp being configured in the context xml file will be deployed.

bundle.root

This is a reference to the org.eclipse.jetty.util.resource.Resource that represents the location of the Bundle. Note that this could be either a directory in the file system if the OSGi container automatically unpacks bundles, or it may be a jar:file: url if the bundle remains packed.

Deploying Bundles as Jetty ContextHandlers

In addition to deploying webapps, the Jetty OSGi container listens for the installation of bundles that are not heavyweight webapps, but rather use the flexible Jetty-specific concept of ContextHandlers.

The following is the criteria used to decide if a bundle can be deployed as a ContextHandler:

Bundle MANIFEST contains Jetty-ContextFilePath

A comma separated list of names of context files - each one of which represents a ContextHandler that should be deployed by Jetty. The context files can be inside the bundle, external to the bundle somewhere on the file system, or external to the bundle in the **jetty.home** directory.

A context file that is inside the bundle:

```
Jetty-ContextFilePath: ./a/b/c/d/foo.xml
```

A context file that is on the file system:

```
Jetty-ContextFilePath: /opt/appcontexts/foo.xml
```

A context file that is relative to jetty.home:

```
Jetty-ContextFilePath: contexts/foo.xml
```

A number of different context files:

```
Jetty-ContextFilePath: ./a/b/c/d/foo.xml,/opt/appcontexts/foo.xml,contexts/foo.xml
```

Other MANIFEST properties that can be used to configure the deployment of the ContextHandler:

managedServerName

The name of the Server instance to which to deploy this webapp bundle. If not specified, defaults to the default Server instance called "defaultJettyServer".

Determining the Context Path for a ContextHandler Bundle

Usually, the context path for the ContextHandler will be set by the context xml file. However, you can override any path set in the context xml file by using the **Web-ContextPath** header in the MANIFEST.

Extra Properties Available for Context Xml Files

Before the Jetty OSGi container applies a context xml file found in a Jetty-ContextFilePath MANIFEST header, it sets a few useful properties that can be referred to within the xml file:

Server

This is a reference to the Jetty org.eclipse.jetty.server.Server instance to which the ContextHandler being configured in the context xml file will be deployed.

bundle.root

This is a reference to the org.eclipse.jetty.util.resource.Resource that represents the location of the Bundle (obtained by calling Bundle.getLocation()). Note that this could be either a directory in the file system if the OSGi container automatically unpacks bundles, or it may be a jar:file: url if the bundle remains packed.

Here's an example of a context xml file that makes use of these properties:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/configure.dtd">

<Configure class="org.eclipse.jetty.server.handler.ContextHandler">

    <!-- Get root for static content, could be on file system or this bundle -->
    <Call id="res" class="org.eclipse.jetty.util.resource.Resource" name="newResource">
        <Arg><Property name="bundle.root"/></Arg>
    </Call>

    <Ref id="res">
        <Call id="base" name="addPath">
            <Arg>/static/</Arg>
        </Call>
    </Ref>

    <Set name="contextPath">/unset</Set>

    <!-- Set up the base resource for static files relative to inside bundle -->
    <Set name="baseResource">
        <Ref id="base"/>
    </Set>

    <Set name="handler">
        <New class="org.eclipse.jetty.server.handler.ResourceHandler">
            <Set name="welcomeFiles">
                <Array type="String">
                    <Item>index.html</Item>
                </Array>
            </Set>
            <Set name="cacheControl">max-age=3600,public</Set>
        </New>
    </Set>
</Configure>
```

Deploying Services as Webapps

In addition to listening for bundles whose format or MANIFEST entries define a webapp or ContextHandler for to be deployed, the Jetty OSGi container also listens for the registration of OSGi services that are instances of org.eclipse.jetty.webapp.WebAppContext. So you may programmatically create a WebAppContext, register it as a service, and have Jetty pick it up and deploy it.

Here's an example of doing that with a simple bundle that serves static content, and an org.osgi.framework.BundleActivator that instantiates the WebAppContext:

The bundle contents:

```
META-INF/MANIFEST.MF
index.html
com/acme/osgi/Activator.class
```

The MANIFEST.MF:

```
Bundle-Classpath: .
Bundle-Name: Jetty OSGi Test WebApp
DynamicImport-Package: org.eclipse.jetty.*;version="[9.0,10.0)"
Bundle-Activator: com.acme.osgi.Activator
Import-Package: org.eclipse.jetty.server.handler;version="[9.0,10)",
    org.eclipse.jetty.webapp;version="[9.0,10)",
    org.osgi.framework;version= "[1.5,2)",
    org.osgi.service.cm;version="1.2.0",
    org.osgi.service.packag eadmin;version="[1.2,2)",
    org.osgi.service.startlevel;version="1.0.0",
    org.osgi.service.url;version="1.0.0",
    org.osgi.util.tracker;version= "1.3.0",
    org.xml.sax,org.xml.sax.helpers
Bundle-SymbolicName: com.acme.testwebapp
```

The Activator code:

```
public void start(BundleContext context) throws Exception
{
    WebAppContext webapp = new WebAppContext();
    Dictionary props = new Hashtable();
    props.put("Jetty-WarFolderPath", ".");
    props.put("contextPath", "/acme");
    context.registerService(ContextHandler.class.getName(), webapp, props);
}
```

The above setup is sufficient for Jetty to recognize and deploy the WebAppContext at /acme.

As the example shows, you can use OSGi Service properties in order to communicate extra configuration information to Jetty:

Jetty-WarFolderPath

The location within the bundle of the root of the static resources for the webapp

Web-ContextPath

The context path at which to deploy the webapp.

Jetty-defaultWebXmlFilePath

The location within the bundle of a webdefault.xml file to apply to the webapp. Defaults to that of the Jetty OSGi container.

Jetty-WebXmlFilePath

The location within the bundle of the web.xml file. Defaults to WEB-INF/web.xml

Jetty-extraClassPath

A classpath of additional items to add to the webapp's classloader.

Jetty-bundleInstall

The path to the base folder that overrides the computed bundle installation - mostly useful for those OSGi frameworks that unpack bundles by default.

Require-TldBundle

A comma separated list of bundle symbolic names of bundles containing TLDs that this webapp depends upon.

managedServerName

The name of the Server instance to which to deploy this webapp. If not specified, defaults to the default Server instance called "defaultJettyServer".

Deploying Services as ContextHandlers

Similarly to WebAppContexts, the Jetty OSGi container can detect the registration of an OSGi Service that represents a ContextHandler and ensure that it is deployed. The ContextHandler can either be fully configured before it is registered as an OSGi service - in which case the Jetty OSGi container will merely deploy it - or the ContextHandler can be partially configured, with the Jetty OSGi container completing the configuration via a context xml file and properties associated with the Service.

Here's an example of doing that with a simple bundle that serves static content with an org.osgi.framework.BundleActivator that instantiates a ContextHandler and registers it as an OSGi Service, passing in properties that define a context xml file and context path for Jetty to apply upon deployment:

The bundle contents:

```
META-INF/MANIFEST.MF
static/index.html
acme.xml
com/acme/osgi/Activator.class
com/acme/osgi/Activator$1.class
```

The MANIFEST:

```
Bundle-Classpath: .
Bundle-Name: Jetty OSGi Test Context
DynamicImport-Package: org.eclipse.jetty.*;version="[9.0,10.0)"
Bundle-Activator: com.acme.osgi.Activator
Import-Package: javax.servlet;version="2.6.0",
javax.servlet.resources;version="2.6.0",
org.eclipse.jetty.server.handler;version="[9.0,10)",
org.osgi.framework;version="1.5.2)",
org.osgi.service.cm;version="1.2.0",
org.osgi.service.packageadmin;version="[1.2,2)",
org.osgi.service.startlevel;version="1.0.0.o",
org.osgi.service.url;version="1.0.0",
org.osgi.util.tracker;version="1.3.0",
org.xml.sax,org.xml.sax.helpers
Bundle-SymbolicName: com.acme.testcontext
```

The Activator code:

```
public void start(final BundleContext context) throws Exception
{
    ContextHandler ch = new ContextHandler();
    ch.addEventListerner(new ServletContextListener () {

        @Override
        public void contextInitialized(ServletContextEvent sce)
        {
            System.err.println("Context is initialized");
        }

        @Override
        public void contextDestroyed(ServletContextEvent sce)
        {
```

```

        System.err.println("Context is destroyed!");
    }

});

Dictionary props = new Hashtable();
props.put("Web-ContextPath", "/acme");
props.put("Jetty-ContextFilePath", "acme.xml");
context.registerService(ContextHandler.class.getName(), ch, props);
}

```

The contents of the acme.xml context file:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure.dtd">

<Configure class="org.eclipse.jetty.server.handler.ContextHandler">

    <!-- Get root for static content, could be on file system or this bundle -->
    <Call id="res" class="org.eclipse.jetty.util.resource.Resource" name="newResource">
        <Arg><Property name="bundle.root"/></Arg>
    </Call>

    <Ref id="res">
        <Call id="base" name="addPath">
            <Arg>/static/</Arg>
        </Call>
    </Ref>

    <Set name="contextPath">/unset</Set>

    <!-- Set up the base resource for static files relative to inside bundle -->
    <Set name="baseResource">
        <Ref id="base"/>
    </Set>

    <Set name="handler">
        <New class="org.eclipse.jetty.server.handler.ResourceHandler">
            <Set name="welcomeFiles">
                <Array type="String">
                    <Item>index.html</Item>
                </Array>
            </Set>
            <Set name="cacheControl">max-age=3600,public</Set>
        </New>
    </Set>
</Configure>

```

You may also use the following OSGi Service properties:

managedServerName

The name of the Server instance to which to deploy this webapp. If not specified, defaults to the default Server instance called "defaultJettyServer".

Extra Properties Available for Context Xml Files

Before the Jetty OSGi container applies a context xml file found in a Jetty-ContextFilePath property, it sets a few useful properties that can be referred to within the xml file:

Server

This is a reference to the Jetty org.eclipse.jetty.server.Server instance to which the ContextHandler being configured in the context xml file will be deployed.

bundle.root

This is a reference to the org.eclipse.jetty.util.resource.Resource that represents the location of the Bundle publishing the ContextHandler as a Service(obtained by calling Bundle.getLocation()).

Note that this could be either a directory in the file system if the OSGi container automatically unpacks bundles, or it may be a jar:file: url if the bundle remains packed.

In the example above, you can see both of these properties being used in the context xml file.

Support for the OSGi Service Platform Enterprise Specification

The Jetty OSGi container implements several aspects of the Enterprise Specification v4.2 for the WebAppContexts and ContextHandlers that it deploys from either bundles or OSGi services as outlined in foregoing sections.

Context Attributes

For each WebAppContext or ContextHandler, the following context attribute is set, as required by section *128.6.1 Bundle Context* pg 427:

osgi-bundleContext

The value of this attribute is the BundleContext representing the Bundle associated with the WebAppContext or ContextHandler.

Service Attributes

As required by the specification section *128.3.4 Publishing the Servlet Context* pg 421, each WebAppContext and ContextHandler deployed by the Jetty OSGi container is also published as an OSGi service (unless it has been already - see sections 1.6 and 1.7). The following properties are associated with these services:

osgi.web.symbolicname

The symbolic name of the Bundle associated with the WebAppContext or ContextHandler

osgi.web.version

The Bundle-Version header from the Bundle associated with the WebAppContext or ContextHandler

osgi.web.contextpath

The context path of the WebAppContext or ContextHandler

OSGi Events

As required by the specification section *128.5 Events* pg 426, the following OSGi Event Admin events will be posted:

org/osgi/service/web/DEPLOYING

The Jetty OSGi container is about to deploy a WebAppContext or ContextHandler

org/osgi/service/web/DEPLOYED

The Jetty OSGi container has finished deploying a WebAppContext or ContextHandler and it is in service

org/osgi/service/web/UNDEPLOYING

The Jetty OSGi container is about to undeploy a WebAppContext or ContextHandler

org/osgi/service/web/UNDEPLOYED

The Jetty OSGi container has finished undeploying a WebAppContext or ContextHandler and it is no longer in service

org/osgi/service/web/FAILED

The Jetty OSGi container failed to deploy a WebAppContext or ContextHandler

Using JSPs

Setup

In order to use JSPs with your webapps and bundles you will need to install the JSP related jars into your OSGi container. You can use the jars from the [jetty distribution](#) in the `${jetty.home}/lib/jsp` directory, but some you will need to download them from [maven central](#). Here is the list of recommended jars (NOTE the version numbers may change in future):

Alternatively, the following maven meta artifact will be the most current list of JSP artifacts for the version of Jetty you are working with.

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-jsp</artifactId>
  <version>${project.version}</version>
  <type>pom</type>
</dependency>
```

Table 28.2. Jars Required for JSP

Jar	Bundle Symbolic Name
javax.el-3.0.0.jar	javax.el
javax.servlet.jsp-api-2.3.1.jar	javax.servlet.jsp
javax.servlet.jsp.jstl-1.2.0.v201105211821.jar	javax.servlet.jsp.jstl
jetty-jsp-fragment-2.3.3.jar	org.eclipse.jetty.jsp.fragment
org.apache.taglibs.standard.glassfish-1.2.0.v201110081803.jar	org.apache.taglibs.standard.glassfish
org.eclipse.jdt.core-3.8.2.v20130121.jar	org.eclipse.jdt.core.compiler.batch

The jetty-osgi-boot-jsp jar

To be able to use JSPs you will need to also install the [jetty-osgi-boot-jsp.jar](#) into your OSGi container. This jar can be obtained from maven central [here](#).

This bundle acts as a fragment extension to the jetty-osgi-boot.jar and adds in support for using JSP.

The jetty-jsp-fragment jar

The standard JSP implementation jar from Glassfish does not contain some classes that are essential in an OSGi environment. To overcome this, we provide the jetty-jsp-fragment jar, which provides all necessary JSP implementation classes, a proper OSGi manifest, and also attaches itself as a fragment extension to the jetty-osgi-boot.jar. You can [download](#) it from the maven central repo [here](#).

If you do not use taglibs, or you only use the JSTL taglibs then you do not need to do any further configuration.

Using TagLibs

The Jetty JSP OSGi container will make available the JSTL tag library to all webapps. If you only use this tag library, then your webapp will work without any further modification.

However, if you make use of other taglibs, you will need to ensure that they are installed into the OSGi container, and also define some System properties and/or MANIFEST headers in your webapp. This is necessary because the classloading regime used by the OSGi container is very different than that used by JSP containers, and the MANIFEST of a normal webapp does not contain enough information for the OSGi environment to allow a JSP container to find and resolve TLDs referenced in the webapp's .jsp files.

Firstly, lets look at an example of a web bundle's modified MANIFEST file so you get an idea of what is required. This example is a web bundle that uses the Spring servlet framework:

```
Bundle-SymbolicName: com.acme.sample
Bundle-Name: WebSample
Web-ContextPath: taglibs
Import-Bundle: org.springframework.web.servlet
Require-TldBundle: org.springframework.web.servlet
Bundle-Version: 1.0.0
Import-Package: org.eclipse.virgo.web.dm;version="[3.0.0,4.0.0)",org.springframework.context.config;version="[2.5.6,4.0.0)",org.springframework.stereotype;version="[2.5.6,4.0.0)",org.springframework.web.bind.annotation;version="[2.5.6,4.0.0)",org.springframework.web.context;version="[2.5.6,4.0.0)",org.springframework.web.servlet;version="[2.5.6,4.0.0)",org.springframework.web.servlet.view;version="[2.5.6,4.0.0)"
```

The **Require-TldBundle** header tells the Jetty OSGi container that this bundle contains TLDs that need to be passed over to the JSP container for processing. The **Import-Bundle** header ensures that the implementation classes for these TLDs will be available to the webapp on the OSGi classpath.

The format of the **Require-TldBundle** header is a comma separated list of one or more symbolic names of bundles containing TLDs.

Container Path Taglibs

Some TLD jars are required to be found on the Jetty OSGi container's classpath, rather than considered part of the web bundle's classpath. For example, this is true of JSTL and Java Server Faces. The Jetty OSGi container takes care of JSTL for you, but you can control which other jars are considered as part of the container's classpath by using the System property **org.eclipse.jetty.osgi.tlbdbundles**:

org.eclipse.jetty.osgi.tlbdbundles

System property defined on the OSGi environment that is a comma separated list of symbolic names of bundles containing taglibs that will be treated as if they are on the container's classpath for web bundles. For example:

```
org.eclipse.jetty.osgi.tlbdbundles=com.acme.special.tags,com.foo.web,org.bar.web.framework
```

You will still need to define the **Import-Bundle** header in the MANIFEST file for the web bundle to ensure that the TLD bundles are on the OSGi classpath.

Alternatively or additionally, you can define a pattern as a context attribute that will match symbolic bundle names in the OSGi environment containing TLDs that should be considered as discovered from the container's classpath.

org.eclipse.jetty.server.webapp.containerIncludeBundlePattern

This pattern must be specified as a context attribute of the WebAppContext representing the web bundle. Unless you are deploying your own WebAppContext (see [Deploying Services as Webapps](#)), you won't have a reference to the WebAppContext to do this. In that case, it can be specified on the org.eclipse.jetty.deploy.DeploymentManager, where it will be applied to *every* webapp deployed by the Jetty OSGi container. The jetty-osgi-boot.jar contains the default jettyhome/etc/jetty-deploy.xml file where the DeploymentManager is defined. To set the pattern, you will need to provide your own etc files - see the section on [customizing the jetty container](#) for how to do this. Here's how the jetty-deploy.xml file would look if we defined a pattern that matched all bundle symbolic names ending in "tag" and "web":

```
<?xml version="1.0"?>
```

```
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure_9_0.dtd">
<Configure id="Server" class="org.eclipse.jetty.server.Server">
    <Call name="addBean">
        <Arg>

        <New id="DeploymentManager" class="org.eclipse.jetty.deploy.DeploymentManager">
            <Set name="contexts">
                <Ref refid="Contexts" />
            </Set>
            <Call name="setContextAttribute">
                <Arg>org.eclipse.jetty.server.webapp.ContainerIncludeBundlePattern</Arg>
                <Arg>.*\tag$|.*\web$</Arg>
            </Call>
        </New>
        </Arg>
    </Call>
</Configure>
```

Again, you will still need to define suitable **Import-Bundle** headers in your web bundle MANIFEST to ensure that bundles matching the pattern are available on the OSGi class path.

Using Annotations/ServletContainerInitializers

Annotations are very much part of the Servlet 3.0 and 3.1 specifications. In order to use them with Jetty, you will need to deploy some extra jars into your OSGi container:

Table 28.3. Jars Required for Annotations

Jar	Bundle Symbolic Name
org.ow2.asm.asm-4.1.jar	org.objectweb.asm
org.ow2.asm.asm-commons-4.1.jar	org.objectweb.asm.commons
org.ow2.asm.asm-tree-4.1.jar	org.objectweb.asm.tre
org.apache.aries.org.apache.aries.util-1.0.0.jar	org.apache.aries.util
org.apache.aries.spifly.org.apache.aries.spifly.dny	org.apache.aries.spifly.dynamic.bundle
javax.annotation javax.annotation-api-1.2.jar	javax.annotation-api
jta api version 1.1.1	
javax mail api version 1.4.1	
jetty-jndi	org.eclipse.jetty.jndi
jetty-plus	org.eclipse.jetty.plus
jetty-annotations	org.eclipse.jetty.annotations

Even if your webapp itself does not use annotations, you may need to deploy these jars because your webapp depends on a Jetty module or a 3rd party library that uses a [javax.servlet.ServletContainerInitializer](#). This interface requires annotation support. It is implemented by providers of code that extend the capabilities of the container. An example of this is the Jetty JSR356 Websocket implementation, although it is being used increasingly commonly in popular libraries like [Spring](#), [Jersey](#) and JSP containers.

To find `ServletContainerInitializers` on the classpath, Jetty uses the Java [ServiceLoader](#) mechanism. For this to function in OSGi, you will need an OSGi R5 compatible container, and have support for the [Service Loader Mediator](#). Jetty has been tested with the [Aries SpyFly](#) module, which is the reference implementation of the Service Loader Mediator, and is listed in the jars above.

Weld

[Weld](#) can be used to add support for CDI (Contexts and Dependency Injection) to Servlets, Listeners and Filters. It is easily configured with Jetty 9.

Weld Setup

The easiest way to configure weld is within the jetty distribution itself.

Filters and Servlets Only



At the moment, only filters and servlets are supported for CDI with Weld. `ServletContextListeners` can not be configured with CDI (until we sort out the proper solution to the issue).

1. Create the directory `$jetty.home/lib/weld`.
2. Download the [weld-servlet artifact](#) and place it into the newly created directory.

```
<dependency>
  <groupId>org.jboss.weld.servlet</groupId>
  <artifactId>weld-servlet</artifactId>
  <version>2.0.0.FINAL</version>
</dependency>
```

3. Edit the `start.ini` file and add an `OPTION` line for weld near the end.

```
OPTIONS=weld
```

4. Continue editing the `start.ini` file and uncomment the lines for `jndi`, `annotations` and `plus`.
5. Ensure your `WEB-INF/web.xml` contains the following listeners.

```
<listener>
  <listener-class>org.jboss.weld.environment.servlet.BeanManagerResourceBindingListener</listener-class>
</listener>
<listener>
  <listener-class>org.jboss.weld.environment.servlet.Listener</listener-class>
</listener>
```

That should be it so when you start up your jetty distribution with the webapp you should see the following output (providing your logging is the default configuration).

```
013-05-30 15:49:01.511:INFO:oejs.Server:main: jetty-9.0.3.v20130506
2013-05-30 15:49:01.564:INFO:oejdp.ScanningAppProvider:main: Deployment monitor [file:/Users/jesse/Desktop/jetty-distribution-9.0.3.v20130506/webapps/] at interval 1
2013-05-30 15:49:01.764:INFO:oejpw.PlusConfiguration:main: No Transaction manager found - if your webapp requires one, please configure one.
May 30, 2013 3:49:01 PM org.jboss.weld.bootstrap.WeldBootstrap <clinit>
INFO: WELD-000900 2.0.0 (Final)
May 30, 2013 3:49:02 PM org.jboss.weld.bootstrap.WeldBootstrap startContainer
INFO: WELD-000101 Transactional services not available. Injection of @Inject UserTransaction not available. Transactional observers will be invoked synchronously.
```

```

May 30, 2013 3:49:02 PM org.jboss.weld.bootstrap.WeldBootstrap startContainer
WARNING: Legacy deployment metadata provided by the integrator. Certain functionality
will not be available.
May 30, 2013 3:49:02 PM org.jboss.weld.environment.jetty.JettyPost72Container initialize
INFO: Jetty7 detected, JSR-299 injection will be available in Listeners, Servlets and
Filters.
May 30, 2013 3:49:02 PM org.jboss.weld.interceptor.util.InterceptionTypeRegistry <clinit>
WARNING: Class 'javax.ejb.PostActivate' not found, interception based on it is not
enabled
May 30, 2013 3:49:02 PM org.jboss.weld.interceptor.util.InterceptionTypeRegistry <clinit>
WARNING: Class 'javax.ejb.PrePassivate' not found, interception based on it is not
enabled
2013-05-30 15:49:02.919:INFO:oejsh.ContextHandler:main: Started
o.e.j.w.WebAppContext@40a49a3{/simple-web,file:/private/var/folders/br/
kbs2g3753c54wmv4j31pnw5r0000gn/T/jetty-0.0.0-8080-simple-web.war-_simple-web-any-
/webapp/,AVAILABLE}{/simple-web.war}
2013-05-30 15:49:02.940:INFO:oejs.ServerConnector:main: Started
ServerConnector@430e0623{HTTP/1.1}{0.0.0.0:8080}

```

Metro

[Metro](#) is the reference implementation for [web services](#). You can easily use Metro with Jetty to integrate web services with your web applications.

Metro Setup

1. [Download](#) the Metro distribution and unpack it to your disk. We'll refer to the unpacked location as `$metro.home`.
2. Create the directory `$jetty.home/lib/metro`
3. Copy the jars from `$metro.home/lib` to `$jetty.home/lib/metro`
4. Edit the `start.ini` file and add an `OPTION` line for metro near the end.

```
OPTIONS=metro
```

That's all the setup you need to do to integrate Jetty and Metro.

Now read the [Metro documentation](#) on [how to create web services](#). The Metro distribution you downloaded should also contain several example web applications in the `$metro.home/samples` directory that you can build and deploy to Jetty (simply by copying the war file produced by the build).

Here's an example of the log output from Jetty when one of the sample Metro wars (from `$metro.home/samples/async`) is deployed to Jetty:

```

[2093] java -jar start.jar

2013-07-26 15:47:53.480:INFO:oejs.Server:main: jetty-9.0.4.v20130625
2013-07-26 15:47:53.549:INFO:oejdp.ScanningAppProvider:main: Deployment monitor [file:/
home/janb/usr_local_java/jetty/jetty-distribution-9.0.4.v20130625/webapps/] at interval 1
Jul 26, 2013 3:47:53 PM com.sun.xml.ws.transport.http.servlet.WSServletContextListener
contextInitialized
INFO: WSSEVLET12: JAX-WS context listener initializing
Jul 26, 2013 3:47:56 PM com.sun.xml.ws.server.MonitorBase createRoot
INFO: Metro monitoring rootname successfully set to:
com.sun.metro:pp=/,type=WSEndpoint,name=/metro-async-AddNumbersService-
AddNumbersImplPort
Jul 26, 2013 3:47:56 PM com.sun.xml.ws.transport.http.servlet.WSServletDelegate <init>
INFO: WSSEVLET14: JAX-WS servlet initializing
2013-07-26 15:47:56.800:INFO:oejsh.ContextHandler:main: Started
o.e.j.w.WebAppContext@75707c77{/metro-async,file:/tmp/jetty-0.0.0-8080-metro-
async.war-_metro-async-any-/webapp/,AVAILABLE}{/metro-async.war}

```

```
2013-07-26 15:47:56.853:INFO:oejs.ServerConnector:main: Started  
ServerConnector@47dce809{HTTP/1.1}{0.0.0.0:8080}
```

Chapter 29. HTTP Client

Table of Contents

Introduction	366
API Usage	367
Other Features	373

Introduction

The Jetty HTTP client module provides easy-to-use APIs, utility classes and a high performance, asynchronous implementation to perform HTTP and HTTPS requests.

The Jetty HTTP client module requires Java version 1.7 or superior and it is Java 1.8 lambda compliant, that is, Java 1.8 applications can use lambda expressions in many of the HTTP client APIs.

Jetty HTTP client is implemented and offers an asynchronous API, that is a programming interface that never blocks for I/O events, thus making it very efficient in thread utilization and well suited for load testing and parallel computation.

However, sometimes all you need to do is to perform a GET request to a resource, and Jetty HTTP client offers also a synchronous API, that is a programming interface where the thread that issued the request blocks until the request/response conversation is complete.

Out of the box features that you get with the Jetty HTTP client are:

- Redirect support; redirect codes such as 302 or 303 are automatically followed
- Cookies support; cookies sent by servers are stored and sent back to servers in matching requests
- Authentication support; HTTP "Basic" and "Digest" authentications are supported, others are pluggable
- Forward proxy support

Initialization

The main class is named, as in Jetty 7 and Jetty 8, `org.eclipse.jetty.client.HttpClient` (although it is not backward compatible with the same class in Jetty 7 and Jetty 8).

You can think of an `HttpClient` instance as a browser instance. Like a browser, it can make requests to different domains, it manages redirects, cookies and authentication, you can configure it with a proxy, and it provides you with the responses to the requests you make.

In order to use `HttpClient`, you must instantiate it, configure it, and then start it:

```
// Instantiate HttpClient
HttpClient httpClient = new HttpClient();

// Configure HttpClient, for example:
httpClient.setFollowRedirects(false);

// Start HttpClient
httpClient.start();
```

You can create multiple instances of `HttpClient`; the reason to do this is that you want to specify different configuration parameters (for example, one instance is configured with a forward proxy while another is not), or because you want the two instances to behave like two different browsers and hence have different cookies, different authentication credentials and so on.

When you create a `HttpClient` instance using the parameterless constructor, you will only be able to perform plain HTTP requests, and you will not be able to perform HTTPS requests.

In order to perform HTTPS requests, you should create first a [SslContextFactory](#), configure it, and pass it to `HttpClient`'s constructor. When created with a `SslContextFactory`, the `HttpClient` will be able to perform both HTTP and HTTPS requests to any domain.

```
// Instantiate and configure the SslContextFactory
SslContextFactory sslContextFactory = new SslContextFactory();

// Instantiate HttpClient with the SslContextFactory
HttpClient httpClient = new HttpClient(sslContextFactory);

// Configure HttpClient, for example:
httpClient.setFollowRedirects(false);

// Start HttpClient
httpClient.start();
```

API Usage

Blocking APIs

The simpler way to perform a HTTP request is the following:

```
ContentResponse response = httpClient.GET("http://domain.com/path?query");
```

Method `HttpClient.GET(...)` performs a HTTP GET request to the given URI and returns a `ContentResponse` when the request/response conversation completes successfully.

The `ContentResponse` object contains the HTTP response information: status code, headers and possibly a content. The content length is limited by default to 2 MiB; for larger content see the section called “Response Content Handling”.

If you want to customize the request, for example by issuing a HEAD request instead of a GET, and simulating a browser user agent, you can do it in this way:

```
ContentResponse response = httpClient.newRequest("http://domain.com/path?query")
    .method(HttpMethod.HEAD)
    .agent("Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:17.0) Gecko/20100101
Firefox/17.0")
    .send();
```

This is a shorthand for:

```
Request request = httpClient.newRequest("http://domain.com/path?query");
request.method(HttpMethod.HEAD);
request.agent("Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:17.0) Gecko/20100101
Firefox/17.0");
ContentResponse response = request.send();
```

You first create a request object using `httpClient.newRequest(...)`, and then you customize it using the fluent API style (that is, chained invocation of methods on the request object). When the request object is customized, you call `Request.send()` that produces the `ContentResponse` when the request/response conversation is complete.

Simple POST requests also have a shortcut method:

```
ContentResponse response = httpClient.POST("http://domain.com/entity/1")
    .param("p", "value")
    .send();
```

The POST parameter values are automatically URL-encoded.

Jetty HTTP client automatically follows redirects, so automatically handles the typical web pattern [POST/Redirect/GET](#), and the response object contains the content of the response of the GET request. Following redirects is a feature that you can enable/disable on a per-request basis or globally.

File uploads also require one line, and make use of JDK 7's `java.nio.file` classes:

```
ContentResponse response = httpClient.newRequest("http://domain.com/upload")
    .file(Paths.get("file_to_upload.txt"), "text/plain")
    .send();
```

It is possible to impose a total timeout for the request/response conversation using the `Request.timeout(...)` method, in this way:

```
ContentResponse response = httpClient.newRequest("http://domain.com/path?query")
    .timeout(5, TimeUnit.SECONDS)
    .send();
```

In the example above, when the 5 seconds expire, the request is aborted and a `java.util.concurrent.TimeoutException` is thrown.

Asynchronous APIs

So far we have shown how to use Jetty HTTP client in a blocking style, that is the thread that issues the request blocks until the request/response conversation is complete. In this section we will look at Jetty HTTP client asynchronous, non-blocking, APIs that are perfectly suited for large content downloads, for parallel processing of requests/responses and in all those cases where performance and efficient thread and resource utilization is a key factor.

The asynchronous APIs rely heavily on listeners that are invoked at various stages of request and response processing. These listeners are implemented by applications and may perform any kind of

logic. The implementation invokes these listeners in the same thread that is used to process the request or response. Therefore, if the application code in these listeners takes a long time to execute, the request or response processing is delayed until the listener returns.

If you need to execute application code that takes long time inside a listener, you must spawn your own thread, and remember to deep copy any data provided by the listener that you will need in your code, because when the listener returns the data it provides may be recycled/cleared/destroyed.

Request and response processing are executed by two different threads and therefore may happen concurrently. A typical example of this concurrent processing is an echo server, where a large upload may be concurrent with the large download echoed back. As a side note, remember that responses may be processed and completed *before* requests; a typical example is a large upload that triggers a quick response - for example an error - by the server: the response may arrive and be completed while the request content is still being uploaded.

The application thread that calls `Request.send(CompleteListener)` performs the processing of the request until either the request is fully processed or until it would block on I/O, then it returns (and therefore never blocks). If it would block on I/O, the thread asks the I/O system to emit an event when the I/O will be ready to continue, then returns. When such an event is fired, a thread taken from the `HttpClient` thread pool will resume the processing of the request.

Responses are processed either from the I/O system thread that fires the event that bytes are ready to be read or by a thread taken from the `HttpClient` thread pool (this is controlled by the `HttpClient.isDispatchIO()` property). Response processing continues until either the response is fully processed or until it would block for I/O. If it would block for I/O, the thread asks the I/O system to emit an event when the I/O will be ready to continue, then returns. When such an event is fired, a thread taken from the `HttpClient` thread pool will resume the processing of the response.

When the request and the response are both fully processed, the thread that finished the last processing (usually the thread that processes the response, but may also be the thread that processes the request - if the request takes more time than the response to be processed) is used to dequeue the next request for the same destination and processes it.

A simple asynchronous GET request that discards the response content can be written in this way:

```
httpClient.newRequest("http://domain.com/path")
    .send(new Response.CompleteListener()
    {
        @Override
        public void onComplete(Result result)
        {
            // Your logic here
        }
    });
}
```

Method `Request.send(Response.CompleteListener)` returns void and does not block; the `Response.CompleteListener` provided as a parameter is notified when the request/response conversation is complete, and the `Result` parameter allows you to access the response object.

You can write the same code using JDK 8's lambda expressions:

```
httpClient.newRequest("http://domain.com/path")
    .send((result) -> { /* Your logic here */ });
}
```

You can impose a total timeout for the request/response conversation in the same way used by the synchronous API:

```

Request request = httpClient.newRequest("http://domain.com/path")
    .timeout(3, TimeUnit.SECONDS)
    .send(new Response.CompleteListener()
    {
        @Override
        public void onComplete(Result result)
        {
            // Your logic here
        }
    });

```

The example above will impose a total timeout of 3 seconds on the request/response conversation.

The HTTP client APIs use listeners extensively to provide hooks for all possible request and response events, and with JDK 8's lambda expressions they're even more fun to use:

```

httpClient.newRequest("http://domain.com/path")
    // Add request hooks
    .onRequestQueued((request) -> { ... })
    .onRequestBegin((request) -> { ... })
    ... // More request hooks available

    // Add response hooks
    .onResponseBegin((response) -> { ... })
    .onResponseHeaders((response) -> { ... })
    .onResponseContent((response, buffer) -> { ... })
    ... // More response hooks available

    .send((result) -> { ... });

```

This makes Jetty HTTP client suitable for HTTP load testing because, for example, you can accurately time every step of the request/response conversation (thus knowing where the request/response time is really spent).

Have a look at the [Request.Listener](#) class to know about request events, and to the [Response.Listener](#) class to know about response events.

Content Handling

Request Content Handling

Jetty HTTP client provides a number of utility classes off the shelf to handle request content.

You can provide request content as String, byte[], ByteBuffer, java.nio.file.Path, InputStream, and provide your own implementation of org.eclipse.jetty.client.api.ContentProvider. Here's an example that provides the request content using java.nio.file.Paths:

```

ContentResponse response = httpClient.newRequest("http://domain.com/upload")
    .file(Paths.get("file_to_upload.txt"), "text/plain")
    .send();

```

This is equivalent to using the PathContentProvider utility class:

```
ContentResponse response = httpClient.newRequest("http://domain.com/upload")
    .content(new PathContentProvider(Paths.get("file_to_upload.txt")), "text/plain")
    .send();
```

Alternatively, you can use `FileInputStream` via the `InputStreamContentProvider` utility class:

```
ContentResponse response = httpClient.newRequest("http://domain.com/upload")
    .content(new InputStreamContentProvider(new
        FileInputStream("file_to_upload.txt")), "text/plain")
    .send();
```

Since `InputStream` is blocking, then also the send of the request will block if the input stream blocks, even in case of usage of the asynchronous `HttpClient` APIs.

If you have already read the content in memory, you can pass it as a `byte[]` using the `BytesContentProvider` utility class:

```
byte[] bytes = ...
ContentResponse response = httpClient.newRequest("http://domain.com/upload")
    .content(new BytesContentProvider(bytes), "text/plain")
    .send();
```

If the request content is not immediately available, but your application will be notified of the content to send, you can use `DeferredContentProvider` in this way:

```
DeferredContentProvider content = new DeferredContentProvider();
httpClient.newRequest("http://domain.com/upload")
    .content(content)
    .send(new Response.CompleteListener()
    {
        @Override
        public void onComplete(Result result)
        {
            // Your logic here
        }
    });
// Content not available yet here
...
// An event happens, now content is available
byte[] bytes = ...;
content.offer(ByteBuffer.wrap(bytes));
...
// All content has arrived
content.close();
```

While the request content is awaited and consequently uploaded by the client application, the server may be able to respond (at least with the response headers) completely asynchronously. In this case, `Response.Listener` callbacks will be invoked before the request is fully sent. This allows fine-

grained control of the request/response conversation: for example the server may reject contents that are too big, send a response to the client, which in turn may stop the content upload.

Another way to provide request content is by using an `OutputStreamContentProvider`, which allows applications to write request content when it is available to the `OutputStream` provided by `OutputStreamContentProvider`:

```
OutputStreamContentProvider content = new OutputStreamContentProvider();

// Use try-with-resources to close the OutputStream when all content is written
try (OutputStream output = content.getOutputStream())
{
    client.newRequest("localhost", 8080)
        .content(content)
        .send(new Response.CompleteListener()
    {
        @Override
        public void onComplete(Result result)
        {
            // Your logic here
        }
    });
}

...
// Write content
writeContent(output);
}
// End of try-with-resource, output.close() called automatically to signal end of content
```

Response Content Handling

Jetty HTTP client allows applications to handle response content in different ways.

The first way is to buffer the response content in memory; this is done when using the blocking APIs (see the section called “Blocking APIs”) and the content is buffered within a `ContentResponse` up to 2 MiB.

If you want to control the length of the response content (for example limiting to values smaller than the default of 2 MiB), then you can use a `org.eclipse.jetty.client.util.FutureResponseListener` in this way:

```
Request request = httpClient.newRequest("http://domain.com/path");

// Limit response content buffer to 512 KiB
FutureResponseListener listener = new FutureResponseListener(request, 512 * 1024);

request.send(listener);

ContentResponse response = listener.get(5, TimeUnit.SECONDS);
```

If the response content length is exceeded, the response will be aborted, and an exception will be thrown by method `get()`.

If you are using the asynchronous APIs (see the section called “Asynchronous APIs”), you can use the `BufferingResponseListener` utility class:

```
httpClient.newRequest("http://domain.com/path")
```

```
// Buffer response content up to 8 MiB
.send(new BufferingResponseListener(8 * 1024 * 1024)
{
    @Override
    public void onComplete(Result result)
    {
        if (!result.isFailed())
        {
            byte[] responseContent = getContent();
            // Your logic here
        }
    }
});
```

The second way is the most efficient (because it avoids content copies) and allows you to specify a `Response.ContentListener`, or a subclass, to handle the content as soon as it arrives:

```
ContentResponse response = httpClient
    .newRequest("http://domain.com/path")
    .send(new Response.Listener.Empty()
    {
        @Override
        public void onContent(Response response, ByteBuffer buffer)
        {
            // Your logic here
        }
    });
});
```

The third way allows you to wait for the response and then stream the content using the `InputStreamResponseListener` utility class:

```
InputStreamResponseListener listener = new InputStreamResponseListener();
httpClient.newRequest("http://domain.com/path")
    .send(listener);

// Wait for the response headers to arrive
Response response = listener.get(5, TimeUnit.SECONDS);

// Look at the response
if (response.getStatus() == 200)
{
    // Use try-with-resources to close input stream.
    try (InputStream responseContent = listener.getInputStream())
    {
        // Your logic here
    }
}
```

Other Features

Cookies Support

Jetty HTTP client supports cookies out of the box. The `HttpClient` instance receives cookies from HTTP responses and stores them in a `java.net.CookieStore`, a class that is part of the JDK. When new requests are made, the cookie store is consulted and if there are matching cookies (that is, cookies that are not expired and that match domain and path of the request) then they are added to the requests.

Applications can programmatically access the cookie store to find the cookies that have been set:

```
CookieStore cookieStore = httpClient.getCookieStore();
List<HttpCookie> cookies = cookieStore.get(URI.create("http://domain.com/path"));
```

Applications can also programmatically set cookies as if they were returned from a HTTP response:

```
CookieStore cookieStore = httpClient.getCookieStore();
HttpCookie cookie = new HttpCookie("foo", "bar");
cookie.setDomain("domain.com");
cookie.setPath("/");
cookie.setMaxAge(TimeUnit.DAYS.toSeconds(1));
cookieStore.add(URI.create("http://domain.com"), cookie);
```

You can remove cookies that you do not want to be sent in future HTTP requests:

```
CookieStore cookieStore = httpClient.getCookieStore();
URI uri = URI.create("http://domain.com");
List<HttpCookie> cookies = cookieStore.get(uri);
for (HttpCookie cookie : cookies)
    cookieStore.remove(uri, cookie);
```

If you want to totally disable cookie handling, you can install a `HttpCookieStore.Empty` instance in this way:

```
httpClient.setCookieStore(new HttpCookieStore.Empty());
```

You can enable cookie filtering by installing a cookie store that performs the filtering logic in this way:

```
httpClient.setCookieStore(new GoogleOnlyCookieStore());

public class GoogleOnlyCookieStore extends HttpCookieStore
{
    @Override
    public void add(URI uri, HttpCookie cookie)
    {
        if (uri.getHost().endsWith("google.com"))
            super.add(uri, cookie);
    }
}
```

The example above will retain only cookies that come from the `google.com` domain or sub-domains.

Authentication Support

Jetty HTTP client supports the "Basic" and "Digest" authentication mechanisms defined by [RFC 2617](#).

You can configure authentication credentials in the HTTP client instance as follows:

```
URI uri = new URI("http://domain.com/secure");
String realm = "MyRealm";
String user = "username";
String pass = "password";

// Add authentication credentials
AuthenticationStore auth = httpClient.getAuthenticationStore();
auth.addAuthentication(new BasicAuthentication(uri, realm, user, pass));

ContentResponse response = httpClient
    .newRequest(uri)
    .send()
    .get(5, TimeUnit.SECONDS);
```

Jetty HTTP client tests authentication credentials against the challenge(s) the server issues, and if they match it automatically sends the right authentication headers to the server for authentication. If the authentication is successful, it caches the result and reuses it for subsequent requests for the same domain and matching URIs.

Successful authentications are cached, but it is possible to clear them in order to force authentication again:

```
httpClient.getAuthenticationStore().clearAuthenticationResults();
```

Proxy Support

Jetty's HTTP client can be configured to use proxies to connect to destinations.

Two types of proxies are available out of the box: a HTTP proxy (provided by class `org.eclipse.jetty.client.HttpProxy`) and a SOCKS 4 proxy (provided by class `org.eclipse.jetty.client.Socks4Proxy`). Other implementations may be written by subclassing `ProxyConfiguration.Proxy`.

A typical configuration is the following:

```
ProxyConfiguration proxyConfig = httpClient.getProxyConfiguration();
HttpProxy proxy = new HttpProxy("proxyHost", proxyPort);
// Do not proxy requests for localhost:8080
proxy.getExcludedAddresses().add("localhost:8080");

httpClient.setProxyConfiguration(proxyConfig);

ContentResponse response = httpClient.GET(uri);
```

You specify the proxy host and port, and optionally also the addresses that you do not want to be proxied, and then set the proxy configuration on the `HttpClient` instance.

Configured in this way, `HttpClient` makes requests to the HTTP proxy (for plain-text HTTP requests) or establishes a tunnel via HTTP CONNECT (for encrypted HTTPS requests).

Chapter 30. WebSocket Introduction

Table of Contents

What Jetty provides	376
WebSocket APIs	377

WebSocket is a new protocol for bidirectional communications over HTTP.

It is based on a low level framing protocol that delivers messages in either UTF-8 TEXT or BINARY format.

A single message in WebSocket can be of any size (the underlying framing however does have a single frame limit of [63-bits](#))

There can be an unlimited number of messages sent.

Messages are sent sequentially, the base protocol does not support interleaved messages.

A WebSocket connection goes through some basic state changes:

Table 30.1. WebSocket connection states

State	Description
CONNECTING	A HTTP Upgrade to WebSocket is in progress
OPEN	The HTTP Upgrade succeeded and the socket is now open and ready to read / write
CLOSING	A WebSocket Close Handshake has been started
CLOSED	WebSocket is now closed, no more read/write possible

When a WebSocket is closed, a [status code](#) and short reason string is provided.

What Jetty provides

Jetty provides an implementation of the following standards and specs.

[RFC-6455](#)

The WebSocket Protocol

We support the version 13 of the released and final spec.

Jetty tests its WebSocket protocol implementation using the [autobahn testsuite](#).

Important



The early drafts of WebSocket were supported in Jetty 7 and Jetty 8, but this support has been removed in Jetty 9.

This means that Jetty 9 will not support the old browsers that implemented the early drafts of WebSocket. (such as Safari 5.0 or Opera 12)

Tip



Want to know if the browser you are targeting supports WebSocket?

Use caniuse.com/websockets to find out.

[JSR-356](#)

The Java WebSocket API (`javax.websocket`)

This is the official Java API for working with WebSockets.

Unstable standards and specs:

[perframe-compression](#)

Per Frame Compression Extension.

An early extension draft from the Google/Chromium team that would provide WebSocket frame compression.

perframe-compression using deflate algorithm is present on many versions of Chrome/Chromium.

Jetty's support for perframe-compression is based on the draft-04 spec.

This standard is being replaced with permessage-compression.

[permessage-compression](#)

Per Frame Compression Extension.

This is the replacement for perframe-compression, switching the compression to being based on the entire message, not the individual frames.

WebSocket APIs

APIs and libraries to implement your WebSockets using Jetty.

[Jetty WebSocket API](#)

The basic common API for creating and working with WebSockets using Jetty.

[Jetty WebSocket Server API](#)

Write WebSocket Server Endpoints for Jetty.

[Jetty WebSocket Client API](#)

Connect to WebSocket servers with Jetty.

[Java WebSocket Client API](#)

The new standard Java WebSocket Client API (`javax.websocket`) [JSR-356]

[Java WebSocket Server API](#)

The new standard Java WebSocket Server API (`javax.websocket.server`) [JSR-356]

Chapter 31. Jetty Websocket API

Table of Contents

Jetty WebSocket API Usage	378
WebSocket Events	378
WebSocket Session	379
Send Messages to Remote Endpoint	379
Using WebSocket Annotations	384
Using WebSocketListener	385
Using the WebSocketAdapter	386
Jetty WebSocket Server API	387
Jetty WebSocket Client API	389

These pages are works in progress that have not been moved to their respective sections yet.

Jetty WebSocket API Usage

Jetty provides its own more powerful WebSocket API, with a common core API for both server and client use of WebSockets.

It is an event driven API based on WebSocket Messages.

WebSocket Events

Every WebSocket can receive various events:

On Connect Event

An indication to the WebSocket that the Upgrade has succeeded and the WebSocket is now open.

You will receive a `org.eclipse.jetty.websocket.api.Session` object that references the specific session for this Open Event.

For normal WebSockets, it is important to hold onto this Session and use it for communicating with the Remote Endpoint.

For Stateless WebSockets, the Session will be passed into each event as it occurs, allowing you to only have 1 instance of a WebSocket serving many Remote Endpoints.

On Close Event

An indication that the WebSocket is now closed.

Every Close Event will have a `Status Code` (and an optional Closure Reason Message)

A normal WebSocket closure will go through a Close Handshake where both the Local Endpoint and the Remote Endpoint both send a Close frame to indicate that the connection is closed.

It is possible for the Local WebSocket to indicate its desire to Close by issuing a Close frame to the Remote Endpoint, but the Remote Endpoint can continue to send messages until it sends a Close Frame. This is known as a Half-Open connection, and it is important to note that once the Local Endpoint has send the Close Frame it cannot write anymore WebSocket traffic.

On an abnormal closure, such as a connection disconnect or a connection timeout, the low level connection will be terminated without going through a Close Handshake, this will still result in an On Close Event (and likely a corresponding On Error Event).

On Error Event

If an error occurred, during the implementation, the WebSocket will be notified via this event handler.

On Message Event

An indication that a complete message has been received and is ready for handling by your WebSocket.

This can be a (UTF8) TEXT message or a raw BINARY message.

WebSocket Session

The [Session](#) object can be used to:

Access State of WebSocket

The Connection State (is it open or not)

```
if(session.isOpen()) {  
    // send message  
}
```

Is the Connection Secure.

```
if(session.isSecure()) {  
    // connection is using 'wss://'  
}
```

What was in the Upgrade Request and Response.

```
UpgradeRequest req = session.getUpgradeRequest();  
String channelName = req.getParameterMap().get("channelName");  
  
UpgradeResponse resp = session.getUpgradeResponse();  
String subprotocol = resp.getAcceptedSubProtocol();
```

What is the Local and Remote Address.

```
InetSocketAddress remoteAddr = session.getRemoteAddress();
```

Configure Policy

Get and Set the Idle Timeout

```
session.setIdleTimeout(2000); // 2 second timeout
```

Get and Set the Maximum Message Size

```
session.setMaximumMessageSize(64*1024); // accept messages up to 64k, fail if larger
```

Send Messages to Remote Endpoint

The most important feature of the Session is access to the [org.eclipse.jetty.websocket.api.RemoteEndpoint](#) needed to send messages.

With RemoteEndpoint you can choose to send TEXT or BINARY WebSocket messages, or the WebSocket PING and PONG control frames.

Blocking Send Message

Most calls are blocking in nature, and will not return until the send has completed (or has thrown an exception).

Example 31.1. Send Binary Message (Blocking)

```
RemoteEndpoint remote = session.getRemote();

// Blocking Send of a BINARY message to remote endpoint
ByteBuffer buf = ByteBuffer.wrap(new byte[] { 0x11, 0x22, 0x33, 0x44 });
try
{
    remote.sendBytes(buf);
}
catch (IOException e)
{
    e.printStackTrace(System.err);
}
```

How to send a simple Binary message using the RemoteEndpoint. This will block until the message is sent, possibly throwing an IOException if unable to send the message.

Example 31.2. Send Text Message (Blocking)

```
RemoteEndpoint remote = session.getRemote();

// Blocking Send of a TEXT message to remote endpoint
try
{
    remote.sendString("Hello World");
}
catch (IOException e)
{
    e.printStackTrace(System.err);
}
```

How to send a simple Text message using the RemoteEndpoint. This will block until the message is sent, possibly throwing an IOException if unable to send the message.

Send Partial Message

If you have a large message to send, and want to send it in pieces and parts, you can utilize the partial message sending methods of RemoteEndpoint. Just be sure you finish sending your message (`isLast == true`)

Example 31.3. Send Partial Binary Message (Blocking)

```
RemoteEndpoint remote = session.getRemote();

// Blocking Send of a BINARY message to remote endpoint
// Part 1
ByteBuffer buf1 = ByteBuffer.wrap(new byte[] { 0x11, 0x22 });
// Part 2 (last part)
ByteBuffer buf2 = ByteBuffer.wrap(new byte[] { 0x33, 0x44 });
try
{
    remote.sendPartialBytes(buf1, false);
    remote.sendPartialBytes(buf2, true); // isLast is true
}
catch (IOException e)
{
    e.printStackTrace(System.err);
}
```

How to send a Binary message in 2 parts, using the partial message support in RemoteEndpoint. This will block until each part of the message is sent, possibly throwing an IOException if unable to send the partial message.

Example 31.4. Send Partial Text Message (Blocking)

```
RemoteEndpoint remote = session.getRemote();

// Blocking Send of a TEXT message to remote endpoint
String part1 = "Hello";
String part2 = " World";
try
{
    remote.sendPartialString(part1, false);
    remote.sendPartialString(part2, true); // last part
}
catch (IOException e)
{
    e.printStackTrace(System.err);
}
```

How to send a Text message in 2 parts, using the partial message support in RemoteEndpoint. This will block until each part of the message is sent, possibly throwing an IOException if unable to send the partial message.

Send Ping / Pong Control Frame

You can also send Ping and Pong control frames using the RemoteEndpoint.

Example 31.5. Send Ping Control Frame (Blocking)

```
RemoteEndpoint remote = session.getRemote();

// Blocking Send of a PING to remote endpoint
String data = "You There?";
ByteBuffer payload = ByteBuffer.wrap(data.getBytes());
try
{
    remote.sendPing(payload);
}
catch (IOException e)
{
    e.printStackTrace(System.err);
}
```

How to send a Ping control frame, with a payload of "You There?" (arriving at Remote Endpoint as a byte array payload). This will block until the message is sent, possibly throwing an IOException if unable to send the ping frame.

Example 31.6. Send Pong Control Frame (Blocking)

```
RemoteEndpoint remote = session.getRemote();

// Blocking Send of a PONG to remote endpoint
String data = "Yup, I'm here";
ByteBuffer payload = ByteBuffer.wrap(data.getBytes());
try
{
    remote.sendPong(payload);
}
catch (IOException e)
{
    e.printStackTrace(System.err);
}
```

How to send a Pong control frame, with a payload of "Yup I'm here" (arriving at Remote Endpoint as a byte array payload). This will block until the message is sent, possibly throwing an IOException if unable to send the pong frame.

To be correct in your usage of Pong frames, you should return the same byte array data that you received in the Ping frame

Async Send Message

However there are also 2 Async send message methods available:

- [`RemoteEndpoint.sendBytesByFuture\(ByteBuffer message\)`](#)
- [`RemoteEndpoint.sendStringByFuture\(String message\)`](#)

Both return a `Future<Void>` that can be used to test for success and failure of the message send using standard [`java.util.concurrent.Future`](#) behavior.

Example 31.7. Send Binary Message (Async Simple)

```
RemoteEndpoint remote = session.getRemote();

// Async Send of a BINARY message to remote endpoint
ByteBuffer buf = ByteBuffer.wrap(new byte[] { 0x11, 0x22, 0x33, 0x44 });
remote.sendBytesByFuture(buf);
```

How to send a simple Binary message using the RemoteEndpoint. The message will be enqueued for outgoing write, but you will not know if it succeeded or failed.

Example 31.8. Send Binary Message (Async, Wait Till Success)

```
RemoteEndpoint remote = session.getRemote();

// Async Send of a BINARY message to remote endpoint
ByteBuffer buf = ByteBuffer.wrap(new byte[] { 0x11, 0x22, 0x33, 0x44 });
try
{
    Future<Void> fut = remote.sendBytesByFuture(buf);
    // wait for completion (forever)
    fut.get();
}
catch (ExecutionException | InterruptedException e)
{
    // Send failed
    e.printStackTrace();
}
```

How to send a simple Binary message using the RemoteEndpoint, tracking the `Future<Void>` to know if the send succeeded or failed.

Example 31.9. Send Binary Message (Async, timeout of send)

```
RemoteEndpoint remote = session.getRemote();

// Async Send of a BINARY message to remote endpoint
ByteBuffer buf = ByteBuffer.wrap(new byte[] { 0x11, 0x22, 0x33, 0x44 });
Future<Void> fut = null;
try
{
    fut = remote.sendBytesByFuture(buf);
    // wait for completion (timeout)
    fut.get(2, TimeUnit.SECONDS);
}
catch (ExecutionException | InterruptedException e)
{
    // Send failed
    e.printStackTrace();
}
catch (TimeoutException e)
{
    // timeout
```

```
e.printStackTrace();
if (fut != null)
{
    // cancel the message
    fut.cancel(true);
}
```

How to send a simple Binary message using the RemoteEndpoint, tracking the Future<Void> and waiting only prescribed amount of time for the send to complete, cancelling the message if the timeout occurs.

Example 31.10. Send Text Message (Async Simple)

```
RemoteEndpoint remote = session.getRemote();

// Async Send of a TEXT message to remote endpoint
remote.sendStringByFuture("Hello World");
```

How to send a simple Text message using the RemoteEndpoint. The message will be enqueued for outgoing write, but you will not know if it succeeded or failed.

Example 31.11. Send Text Message (Async, Wait Till Success)

```
RemoteEndpoint remote = session.getRemote();

// Async Send of a TEXT message to remote endpoint
try
{
    Future<Void> fut = remote.sendStringByFuture("Hello World");
    // wait for completion (forever)
    fut.get();
}
catch (ExecutionException | InterruptedException e)
{
    // Send failed
    e.printStackTrace();
}
```

How to send a simple Binary message using the RemoteEndpoint, tracking the Future<Void> to know if the send succeeded or failed.

Example 31.12. Send Text Message (Async, timeout of send)

```
RemoteEndpoint remote = session.getRemote();

// Async Send of a TEXT message to remote endpoint
Future<Void> fut = null;
try
{
    fut = remote.sendStringByFuture("Hello World");
    // wait for completion (timeout)
    fut.get(2, TimeUnit.SECONDS);
}
catch (ExecutionException | InterruptedException e)
{
    // Send failed
    e.printStackTrace();
}
catch (TimeoutException e)
{
    // timeout
    e.printStackTrace();
    if (fut != null)
    {
        // cancel the message
        fut.cancel(true);
    }
}
```

How to send a simple Binary message using the RemoteEndpoint, tracking the Future<Void> and waiting only prescribed amount of time for the send to complete, cancelling the message if the timeout occurs.

Using WebSocket Annotations

The most basic form of WebSocket is a marked up POJO with annotations provided by the Jetty WebSocket API.

Example 31.13. AnnotatedEchoSocket.java

```
package examples.echo;

import org.eclipse.jetty.websocket.api.Session;
import org.eclipse.jetty.websocket.api.annotations.OnWebSocketMessage;
import org.eclipse.jetty.websocket.api.annotations.WebSocket;

/**
 * Example EchoSocket using Annotations.
 */
@WebSocket(maxTextMessageSize = 64 * 1024)
public class AnnotatedEchoSocket {

    @OnWebSocketMessage
    public void onText(Session session, String message) {
        if (session.isOpen()) {
            System.out.printf("Echoing back message [%s]\n", message);
            session.getRemote().sendString(message, null);
        }
    }
}
```

The above example is a simple WebSocket echo endpoint that will echo back any TEXT messages it receives.

This implementation is using a stateless approach to a Echo socket, as the Session is being passed into the Message event as the event occurs. This would allow you to reuse the single instance of the AnnotatedEchoSocket for working with multiple endpoints.

The annotations you have available:

[@WebSocket](#)

A required class level annotation.

Flags this POJO as being a WebSocket.

The class must be not abstract and public.

[@OnWebSocketConnect](#)

An optional method level annotation.

Flags one method in the class as receiving the On Connect event.

Method must be public, not abstract, return void, and have a single [Session](#) parameter.

[@OnWebSocketClose](#)

An optional method level annotation.

Flags one method in the class as receiving the On Close event.

Method signature must be public, not abstract, and return void.

The method parameters:

1. [Session](#) (optional)

2. int closeCode (required)
3. String closeReason (required)

@OnWebSocketMessage

An optional method level annotation.

Flags up to 2 methods in the class as receiving On Message events.

You can have 1 method for TEXT messages, and 1 method for BINARY messages.

Method signature must be public, not abstract, and return void.

The method parameters for Text messages:

1. [Session](#) (optional)
2. String text (required)

The method parameters for Binary messages:

1. [Session](#) (optional)
2. byte buf[] (required)
3. int offset (required)
4. int length (required)

@OnWebSocketError

An optional method level annotation.

Flags one method in the class as receiving Error events from the WebSocket implementation.

Method signatures must be public, not abstract, and return void.

The method parameters:

1. [Session](#) (optional)
2. Throwable cause (required)

@OnWebSocketFrame

An optional method level annotation.

Flags one method in the class as receiving Frame events from the WebSocket implementation after they have been processed by any extensions declared during the Upgrade handshake.

Method signatures must be public, not abstract, and return void.

The method parameters:

1. [Session](#) (optional)
2. [Frame](#) (required)

The Frame received will be notified on this method, then be processed by Jetty, possibly resulting in another event, such as On Close, or On Message. Changes to the Frame will not be seen by Jetty.

Using WebSocketListener

The basic form of a WebSocket using the [org.eclipse.jetty.websocket.api.WebSocketListener](#) for incoming events.

Example 31.14. ListenerEchoSocket.java

```
package examples.echo;

import org.eclipse.jetty.websocket.api.Session;
import org.eclipse.jetty.websocket.api.WebSocketListener;

/**
 * Example EchoSocket using Listener.
 */
public class ListenerEchoSocket implements WebSocketListener {

    private Session outbound;

    @Override
    public void onWebSocketBinary(byte[] payload, int offset, int len) {
    }

    @Override
    public void onWebSocketClose(int statusCode, String reason) {
        this.outbound = null;
    }

    @Override
    public void onWebSocketConnect(Session session) {
        this.outbound = session;
    }

    @Override
    public void onWebSocketError(Throwable cause) {
        cause.printStackTrace(System.err);
    }

    @Override
    public void onWebSocketText(String message) {
        if ((outbound != null) && (outbound.isOpen())) {
            System.out.printf("Echoing back message [%s]\n", message);
            outbound.getRemote().sendString(message, null);
        }
    }
}
```

This is by far the most basic and best performing (speed and memory wise) WebSocket implementation you can create. If the listener is too much work for you, you can instead opt for the `WebSocketAdapter`

Using the `WebSocketAdapter`

A basic adapter for managing the `Session` object on the `WebSocketListener`.

Example 31.15. AdapterEchoSocket.java

```
package examples.echo;

import java.io.IOException;
import org.eclipse.jetty.websocket.api.WebSocketAdapter;

/**
 * Example EchoSocket using Adapter.
 */
public class AdapterEchoSocket extends WebSocketAdapter {

    @Override
    public void onWebSocketText(String message) {
        if (isConnected()) {
            try {
                System.out.printf("Echoing back message [%s]\n", message);
                getRemote().sendString(message);
            } catch (IOException e) {
                e.printStackTrace(System.err);
            }
        }
    }
}
```

```
}
```

This is a convenience class to make using the WebSocketListener easier, and provides some useful methods to check the state of the Session.

Jetty WebSocket Server API

Jetty provides the ability to wire up WebSocket endpoints to Servlet Path Specs via the use of a WebSocketServlet bridge servlet.

Internally, Jetty manages the HTTP Upgrade to WebSocket and migration from a HTTP Connection to a WebSocket Connection.

This will only work when running within the Jetty Container. (unlike past Jetty technologies, you cannot get Jetty WebSocket server functionality running Jetty within other containers like JBoss, Tomcat, or WebLogic)

The Jetty WebSocketServlet

To wire up your WebSocket to a specific path via the WebSocketServlet, you will need to extend org.eclipse.jetty.websocket.servlet.WebSocketServlet and specify what WebSocket object should be created with incoming Upgrade requests.

Example 31.16. MyEchoServlet.java

```
package examples;

import javax.servlet.annotation.WebServlet;
import org.eclipse.jetty.websocket.servlet.WebSocketServlet;
import org.eclipse.jetty.websocket.servlet.WebSocketServletFactory;

@SuppressWarnings("serial")
@WebServlet(name = "MyEcho WebSocket Servlet", urlPatterns = { "/echo" })
public class MyEchoServlet extends WebSocketServlet {

    @Override
    public void configure(WebSocketServletFactory factory) {
        factory.getPolicy(). setIdleTimeout(10000);
        factory.register(MyEchoSocket.class);
    }
}
```

This example will create a Servlet mapped via the [@WebServlet](#) annotation to the Servlet path spec of "/echo" (or you can do this manually in the WEB-INF/web.xml of your web application) which will create MyEchoSocket instances when encountering HTTP Upgrade requests.

The WebSocketServlet.configure(WebSocketServletFactory factory) is where you put your specific configuration for your WebSocket. In the example we specify a 10 second idle timeout and register MyEchoSocket with the default WebSocketCreator the WebSocket class we want to be created on Upgrade.

Using the WebSocketCreator

All WebSocket's are created via whatever [WebSocketCreator](#) you have registered with the [WebSocketServletFactory](#).

By default, the WebSocketServletFactory is a simple WebSocketCreator capable of creating a single WebSocket object. Use [WebSocketCreator.register\(Class<?> websocket\)](#) to tell the WebSocketServletFactory which class it should instantiate (make sure it has a default constructor).

If you have a more complicated creation scenario, you might want to provide your own WebSocketCreator that bases the WebSocket it creates off of information present in the UpgradeRequest object.

Example 31.17. MyAdvancedEchoCreator.java

```
package examples;

import org.eclipse.jetty.websocket.servlet.ServletUpgradeRequest;
import org.eclipse.jetty.websocket.servlet.ServletUpgradeResponse;
import org.eclipse.jetty.websocket.servlet.WebSocketCreator;

public class MyAdvancedEchoCreator implements WebSocketCreator {

    private MyBinaryEchoSocket binaryEcho;

    private MyEchoSocket textEcho;

    public MyAdvancedEchoCreator() {
        this.binaryEcho = new MyBinaryEchoSocket();
        this.textEcho = new MyEchoSocket();
    }

    @Override
    public Object createWebSocket(ServletUpgradeRequest req, ServletUpgradeResponse resp)
    {
        for (String subprotocol : req.getSubProtocols()) {
            if ("binary".equals(subprotocol)) {
                resp.setAcceptedSubProtocol(subprotocol);
                return binaryEcho;
            }
            if ("text".equals(subprotocol)) {
                resp.setAcceptedSubProtocol(subprotocol);
                return textEcho;
            }
        }
        return null;
    }
}
```

Here we show a WebSocketCreator that will utilize the [WebSocket subprotocol](#) information from request to determine what WebSocket type should be created.

Example 31.18. MyAdvancedEchoServlet.java

```
package examples;

import javax.servlet.annotation.WebServlet;
import org.eclipse.jetty.websocket.servlet.WebSocketServlet;
import org.eclipse.websocket.servlet.WebSocketServletFactory;

@SuppressWarnings("serial")
@WebServlet(name = "MyAdvanced Echo WebSocket Servlet", urlPatterns = { "/advecho" })
public class MyAdvancedEchoServlet extends WebSocketServlet {

    @Override
    public void configure(WebSocketServletFactory factory) {
        factory.getPolicy().setIdleTimeout(10000);
        factory.setCreator(new MyAdvancedEchoCreator());
    }
}
```

When you want a custom WebSocketCreator, use [WebSocketServletFactory.setCreator\(WebSocketCreator creator\)](#) and the WebSocketServletFactory will use your creator for all incoming Upgrade requests on this servlet.

Other uses for a WebSocketCreator:

- Controlling the selection of WebSocket subprotocol
- Performing any WebSocket origin you deem important.

- Obtaining the HTTP headers from incoming request
- Obtaining the Servlet HttpSession object (if it exists)
- Specifying a response status code and reason

If you don't want to accept the upgrade, simply return null from the [`WebSocketCreator.createWebSocket\(UpgradeRequest req, UpgradeResponse resp\)`](#) method.

Jetty WebSocket Client API

Jetty also provides a Jetty WebSocket Client Library to write make talking to WebSocket servers easier.

To use the Jetty WebSocket Client on your own Java project you will need the following maven artifacts.

```
<dependency>
  <groupId>org.eclipse.jetty.websocket</groupId>
  <artifactId>websocket-client</artifactId>
  <version>${project.version}</version>
</dependency>
```

The WebSocketClient

To use the WebSocketClient you will need to hook up a WebSocket object instance to a specific destination WebSocket URI.

Example 31.19. SimpleEchoClient.java

```
package examples;

import java.net.URI;
import java.util.concurrent.TimeUnit;
import org.eclipse.jetty.websocket.client.ClientUpgradeRequest;
import org.eclipse.jetty.websocket.client.WebSocketClient;

/**
 * Example of a simple Echo Client.
 */
public class SimpleEchoClient {

    public static void main(String[] args) {
        String destUri = "ws://echo.websocket.org";
        if (args.length > 0) {
            destUri = args[0];
        }
        WebSocketClient client = new WebSocketClient();
        SimpleEchoSocket socket = new SimpleEchoSocket();
        try {
            client.start();
            URI echoUri = new URI(destUri);
            ClientUpgradeRequest request = new ClientUpgradeRequest();
            client.connect(socket, echoUri, request);
            System.out.printf("Connecting to : %s%n", echoUri);
            socket.awaitClose(5, TimeUnit.SECONDS);
        } catch (Throwable t) {
            t.printStackTrace();
        } finally {
            try {
                client.stop();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
    }
}
```

The above example connects to a remote WebSocket server and hands off a SimpleEchoSocket to perform the logic on the websocket once connected, waiting for the socket to register that it has closed.

Example 31.20. SimpleEchoSocket.java

```
package examples;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import org.eclipse.jetty.websocket.api.Session;
import org.eclipse.jetty.websocket.api.StatusCode;
import org.eclipse.jetty.websocket.api.annotations.OnWebSocketClose;
import org.eclipse.jetty.websocket.api.annotations.OnWebSocketConnect;
import org.eclipse.jetty.websocket.api.annotations.OnWebSocketMessage;
import org.eclipse.jetty.websocket.api.annotations.WebSocket;

/**
 * Basic Echo Client Socket
 */
@WebSocket(maxTextMessageSize = 64 * 1024)
public class SimpleEchoSocket {

    private final CountDownLatch closeLatch;

    @SuppressWarnings("unused")
    private Session session;

    public SimpleEchoSocket() {
        this.closeLatch = new CountDownLatch(1);
    }

    public boolean awaitClose(int duration, TimeUnit unit) throws InterruptedException {
        return this.closeLatch.await(duration, unit);
    }

    @OnWebSocketClose
    public void onClose(int statusCode, String reason) {
        System.out.printf("Connection closed: %d - %s%n", statusCode, reason);
        this.session = null;
        this.closeLatch.countDown();
    }

    @OnWebSocketConnect
    public void onConnect(Session session) {
        System.out.printf("Got connect: %s%n", session);
        this.session = session;
        try {
            Future<Void> fut;
            fut = session.getRemote().sendStringByFuture("Hello");
            fut.get(2, TimeUnit.SECONDS);
            fut = session.getRemote().sendStringByFuture("Thanks for the conversation.");
            fut.get(2, TimeUnit.SECONDS);
            session.close(StatusCode.NORMAL, "I'm done");
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }

    @OnWebSocketMessage
    public void onMessage(String msg) {
        System.out.printf("Got msg: %s%n", msg);
    }
}
```

When the SimpleEchoSocket connects, it sends 2 Text messages and then closes the socket.

The onMessage(String msg) receives the responses from the remote server WebSocket and outputs them to the console.

Chapter 32. Java Websocket API

Table of Contents

Java WebSocket Client API Usage	391
Java WebSocket Server API	391

JSR-356 These pages are works in progress that have not been moved to their respective sections yet.

Java WebSocket Client API Usage

Java WebSocket Client API

The simpler way to perform a websocket request is the following:

Java WebSocket Server API

Java WebSocket Server API

The simpler way to perform a websocket request is the following:

Part V. Reference Guide

Table of Contents

33. Platforms, Stacks and Alternative Distributions	394
Many many options...	394
Jelastic	394
CloudFoundry	394
Amazon Elastic Beanstalk	396
Fedora	398
Ubuntu	398
34. Architecture	399
Jetty Architecture	399
Jetty Classloading	405
Managing 1xx Responses	408
Creating a Custom Protocol	408
35. Contributing to Jetty	411
Community	411
Documentation	412
Source Control and Building	415
Coding Standards	417
Issues, Features, and Bugs	418
Contributing Patches	418
Releasing Jetty	422
Testing a Jetty Release	425
36. Reference Section	430
Jetty XML Syntax	430
Jetty XML Usage	442
jetty.xml	443
jetty-web.xml	444
jetty-env.xml	445
webdefault.xml	446
Jetty override-web.xml	448
37. Troubleshooting	450
Troubleshooting Zip Exceptions	450
Troubleshooting Locked Files on Windows	450
Preventing Memory Leaks	452
Jetty Security Reports	455

Chapter 33. Platforms, Stacks and Alternative Distributions

Table of Contents

Many many options...	394
Jelastic	394
CloudFoundry	394
Amazon Elastic Beanstalk	396
Fedora	398
Ubuntu	398

Many many options...

In addition to using Jetty in its distribution form and its multiple embedded forms, there are a number of alternative ways to use Jetty. Many products and open source projects out there distribute Jetty themselves, in both distribution and embedded forms, not to mention different operating systems bundling Jetty in other installable forms.

If your platform supports Jetty from a distribution or deployment perspective and want to be included on this list just fork the documentation and submit a pull request, or contact us. Check out our list of [Powered By](#) page for software that makes use of Jetty, often in novel and exciting ways.

Jelastic

Jelastic is a wonderful place to host your applications with solid support for Jetty. As a cloud hosting platform they take the majority of configuration and installation details out of the picture and focus on letting you focus on your web application.

- [Why Jelastic?](#)
- [Jetty Hosting](#)

CloudFoundry

Overview

[CloudFoundry](#) is an open platform intended as a place to deploy end user applications in a manner which is both simple and eminently scalable to fit the needs of the application. With the release of their V2 framework the Jetty project has created a buildpack which allows you to deploy your Java based web application onto Jetty and still make use of the remainder of the CloudFoundry platform.

This buildpack itself is quite simple to use. A collection of ruby scripting and the buildpack conventions will allow Jetty to be downloaded, configured and customized to your needs and then have your web application deployed onto it. While the default buildpack we have created is useful to deploy a stock configuration of jetty, it is quite likely that you will want to fork the buildpack and tweak it to fit your immediate needs. This process is made trivial since buildpacks install from a GitHub repository. For example, to change the jetty version simply fork it in GitHub and tweak the `JETTY_VERSION` string in the `jetty_web.rb` file.

If you have additional modifications to make to the Jetty server, like perhaps configuring additional static contexts, setting up a proxy servlet, adding jar files to the `jetty.home/lib/ext` directory, etc you

can either adapt the ruby scripting directly or place them under the appropriate location in the `/resources` directory of this buildpack and they will be copied into the correct location.

For the time being I'll leave this buildpack under my personal github account and should there be interest expressed I am more then happy to push it over to <https://github.com/jetty-project> down the road for proper contributions, etc.

Usage

To show how incredibly easy it is to use the Jetty buildpack with cloudfoundry, this is all the more you need to do to deploy your application. Refer to the CloudFoundry [documentation](#) to get started, get the cf utilities installed and an environment configured.

```
$ cf push sniffertest --buildpack=git://github.com/jmcc0nn3ll/jetty-buildpack.git
```



Web Application

In this example the web application is uploaded from the **current** directory so make sure you have changed directory into the root of your web application. The `sniffertest` on the commandline refers to what you are calling the application, not the directory to deploy. Also note that the webapplication is installed into the ROOT context of Jetty as is available at the root context of the server. Any additional web applications will have to be configured within the buildpack as mentioned above.

You will be prompted to answer a series of questions describing the execution environment and any additional services you need enabled (databases, etc).

```
Instances> 1
Custom startup command> none
1: 64M
2: 128M
3: 256M
4: 512M
5: 1G
Memory Limit> 256M
Creating sniffertest... OK
1: sniffertest
2: none
Subdomain> sniffertest
1: al-app.cf-app.com
2: none
Domain> al-app.cf-app.com
Binding sniffertest.al-app.cf-app.com to sniffertest... OK
Create services for application?> n
Save configuration?> n
```

Once answered you will see the installation process of your application.

```
Uploading sniffertest... OK
```

```
Starting sniffertest... OK
-----> Downloaded app package (4.0K)
Initialized empty Git repository in /tmp/buildpacks/jetty-buildpack.git/.git/
Installing jetty-buildpack.git.
Downloading JDK...
Copying openjdk-1.7.0_21.tar.gz from the buildpack cache ...
Unpacking JDK to .jdk
Downloading Jetty: jetty-distribution-9.0.3.v20130506.tar.gz
Downloading jetty-distribution-9.0.3.v20130506.tar.gz from http://repo2.maven.org/maven2/
org/eclipse/jetty/jetty-distribution/9.0.3.v20130506/ ...
Unpacking Jetty to .jetty
-----> Uploading staged droplet (36M)
-----> Uploaded droplet
Checking sniffertest...
Staging in progress...
0/1 instances: 1 starting
0/1 instances: 1 starting
0/1 instances: 1 starting
0/1 instances: 1 starting
1/1 instances: 1 running
OK
```

The application is now available at the configured location! Under the url `http://sniffertest.al-app.cf-app.com/` in this particular example.

Acknowledgements

The Jetty buildpack was forked from the CloudFoundry Java buildpack. The Virgo Buildpack that Glyn worked on was used as a sanity check.

- <http://github.com/cloudfoundry/cloudfoundry-buildpack-java>
- <http://github.com/glyn/virgo-buildpack>

CloudFoundry buildpacks were modelled on Heroku buildpacks.

Amazon Elastic Beanstalk

[Elastic Beanstalk](#) is a component with the [Amazon Web Services](#) offering that allows you to configure an entire virtual machine based on one of several available baseline configurations and then customize it through a powerful configuration system. While the default offerings currently available are based on Tomcat for the java community, we worked out the basics using that configuration system to enable the usage of Jetty instead.

Overview

Elastic beanstalk has a very [powerful configuration mechanism](#) so this integration taps into that to effectively rework the tomcat configuration and replace it with the bits required to make jetty run in its place. Below is a walk through of what the various configuration files are doing and how the general flow of configuration on beanstalk happens.

There is an `.ebextensions` directory in your beanstalk application which contains all of the files required to configure and customize your beanstalk and application combo. Files that end in `.config` in this directory are processed in alphabetical order.

00-java7.config

installs java 7 onto the beanstalk environment and makes it the default

10-tweak.config

not required, but changes the `/opt/elasticbeanstalk` directory to be readable making debugging easier

11-jetty.config

installs jetty9 into `/opt/jetty-9` and removes unneeded distribution files

12-beanstalk.config

handles replacing tomcat with jetty in many configuration files, configures logging and wires up system startup processes. Some files in your `.ebextensions` directory are moved to replace files under `/opt/elasticbeanstalk`.

If you look in the `.ebextensions` directory of your application you should also see other jetty specific xml and ini files. The final config file handles these as they are largely customization for your application.

20-testapp.config

layers application specific configuration files into the jetty installation

The files in our example test webapp here enable various OPTIONS for libraries that need to be loaded, customize the root application being deployed and even deploy additional contexts like we do in our jetty distribution demo. This is also the mechanism that you would use to wire up application specific things, for example if you needed additional software installed, customized directories made, etc.

Maven Bits

Support for this feature leverages Maven to make things easy and is composed of three different modules.

jetty-beanstalk-overlay

This is the collection of scripts that are required to wedge jetty into the normal beanstalk setup.
This module is intended to extract into an webapp to enable it for beanstalk usage with jetty.

jetty-beanstalk-resources

This generates an artifact of files that are downloaded by the configuration process and contains replacements for certain beanstalk files as well as various system level jetty configuration files like an updated `jetty.sh` script for the `/etc/init.d` setup.

jetty-beanstalk-testapp

An example webapp that shows both how to combine the war file from another maven module with the jetty-beanstalk-overlay to produce a beanstalk enabled application bundle. Also included is examples of how to alter the jetty configuration for things like a customized `start.ini` file.



Note

The test webapps needs access to a snapshot version of the test-jetty-webapp so it really serves as more of an example of how to layer your webapp with the bits required to customize your app for beanstalk and jetty.

To actually make use of these artifacts you currently must clone this git repository and build it locally. Once you have the artifacts you simply need to copy the approach in the jetty-beanstalk-testapp to apply the configuration to your webapp.

- <https://github.com/jmcc0nn3ll/jetty-beanstalk>



A Note on Bluepill

Bluepill is used to manage the start and stop process of the app server. This seems to be a problematic bit of software with a colored history and the version in use at the time of this

writing is old. When starting and stopping (or restarting) the appserver you may see error messages show up that the Server timed out getting a response or things like that. These are red herrings and my experience is that jetty has started and stopped just fine, the pid file required shows up in a very timely fashion (under `/var/run/jetty.pid`) so do check that the app server has started, but please be aware there is a strangeness here that hasn't been sorted out yet.

Fedora

As of Fedora 19, Jetty 9 is the version of Jetty available. This distribution of Jetty is not created or maintained by the Jetty project though we have had a fair amount of communication with the folks behind it and we are very pleased with how this Linux distribution has stayed current. Releases are kept largely in sync with our releases as there is a wonderful automatic notification mechanism in place for Fedora that detects our releases and immediately opens an issue for them to update.

- [Jetty on Fedora](#)

Ubuntu

Currently there are no actual .deb files available for installing on Debain based Linux machines but there is a handy blog that as been largely been kept up to date on the steps involved through the comments.

- [Install Jetty9 on Ubuntu](#)

Chapter 34. Architecture

Table of Contents

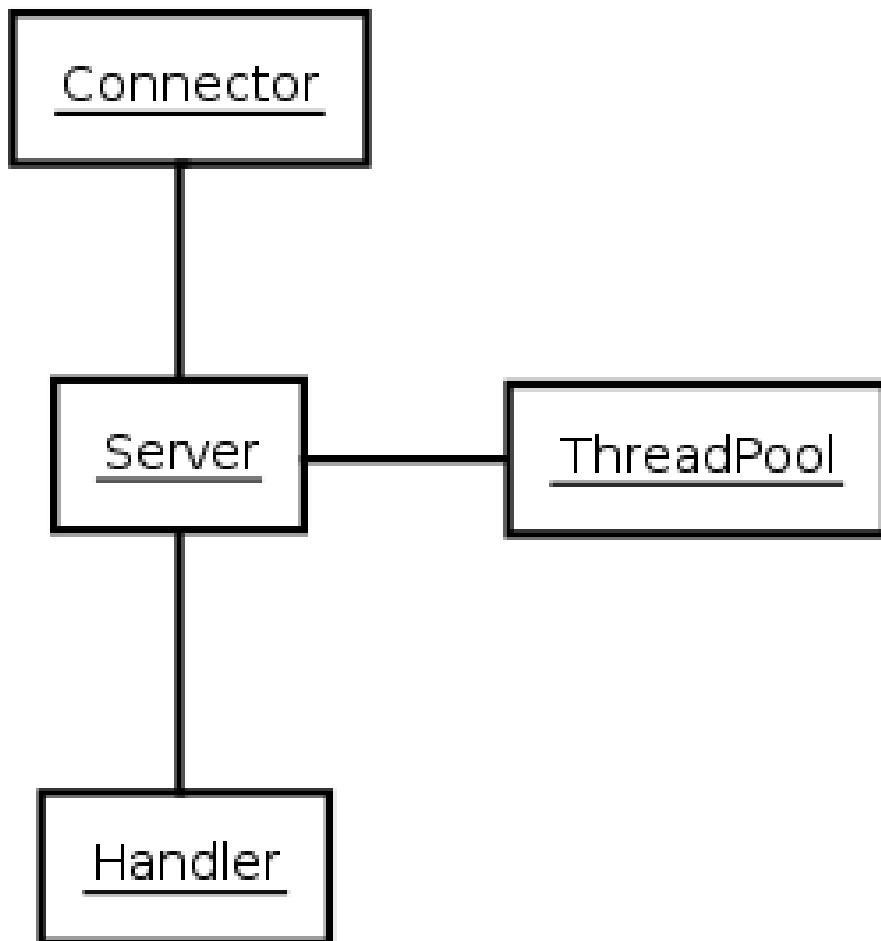
Jetty Architecture	399
Jetty Classloading	405
Managing 1xx Responses	408
Creating a Custom Protocol	408

General items related to the architecture of jetty and how it deals with certain design decisions.

Jetty Architecture

View from 20,000 feet

The Jetty [Server](#) is the plumbing between a collection of Connectors that accept HTTP connections and a collection of Handlers that service requests from the connections and produce responses, with threads from a thread pool doing the work.



While the Jetty request/responses are derived from the Servlet API, the full features of the Servlet API are only available if you configure the appropriate handlers. For example, the session API on the request is inactive unless the request has been passed to a Session Handler. The concept of a servlet itself is implemented by a Servlet Handler. If servlets are not required, there is very little overhead in the use of the servlet request/response APIs. Thus you can build a Jetty server using only connectors and handlers, without using servlets.

The job of configuring Jetty is building a network of connectors and handlers and providing their individual configurations. As Jetty components are simply Plain Old Java Objects (POJOs), you can accomplish this assembly and configuration of components by a variety of techniques:

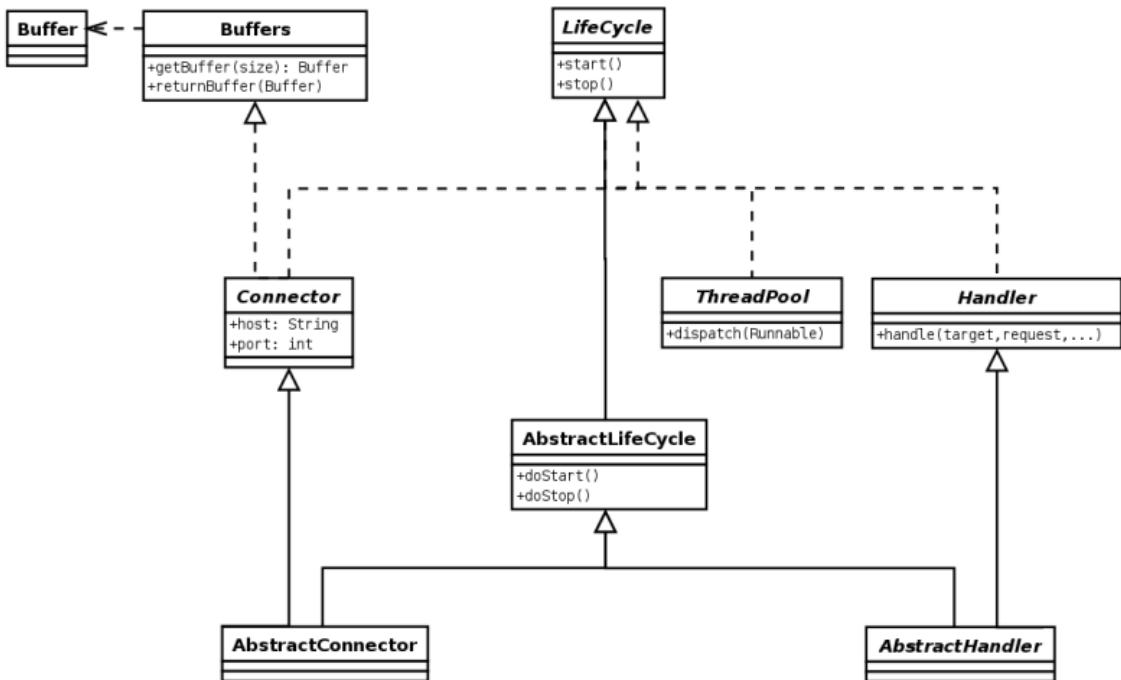
deeper ???

deepest [this is custom](#)

- In code. See the examples in the Jetty 7 Latest Source XRef.
- Using Jetty XML-dependency injection style XML format.
- With your dependency injection framework of choice: Spring or XBean.
- Using Jetty WebApp and Context Deployers.

Patterns

The implementation of Jetty follows some fairly standard patterns. Most abstract concepts such as Connector, Handler and Buffer are captured by interfaces. Generic handling for those interfaces is then provided in an Abstract implementation such as `AbstractConnector`, `AbstractHandler` and `AbstractBuffer`.



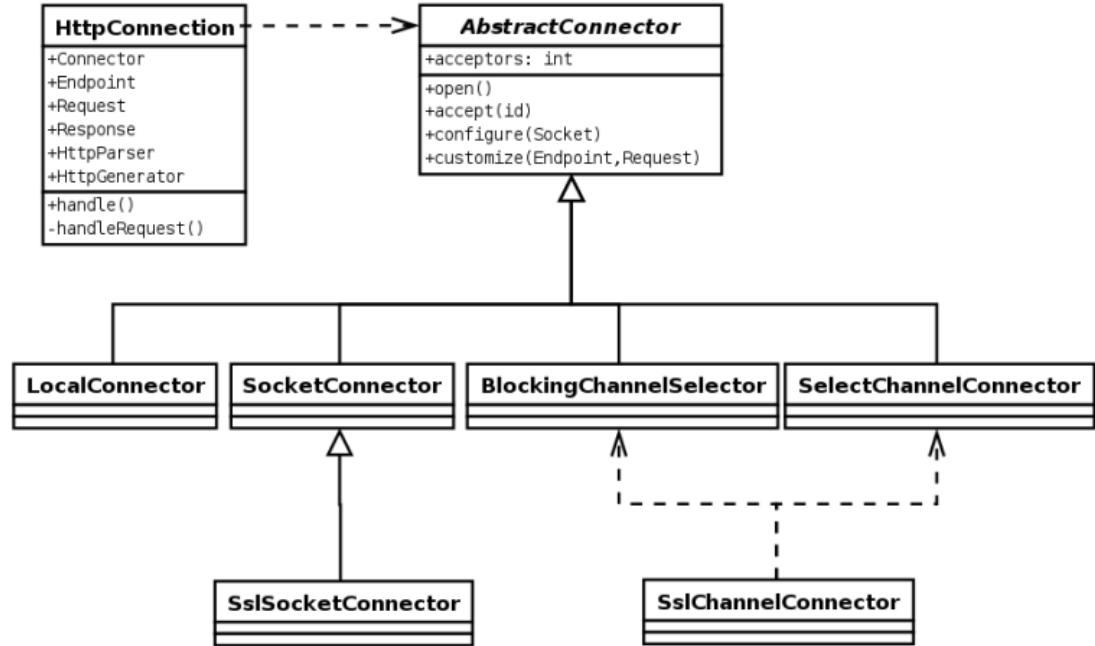
The JSR77 inspired life cycle of most Jetty components is represented by the `LifeCycle` interface and the `AbstractLifeCycle` implementation used as the base of many Jetty components.

Jetty provides its own IO Buffering abstract over String, byte arrays and NIO buffers. This allows for greater portability of Jetty as well as hiding some of the complexity of the NIO layer and its advanced features.

Connectors

This diagram is a little out of date, as a Connection interface has been extracted out of `HttpConnector` to allow support for the AJP protocol.

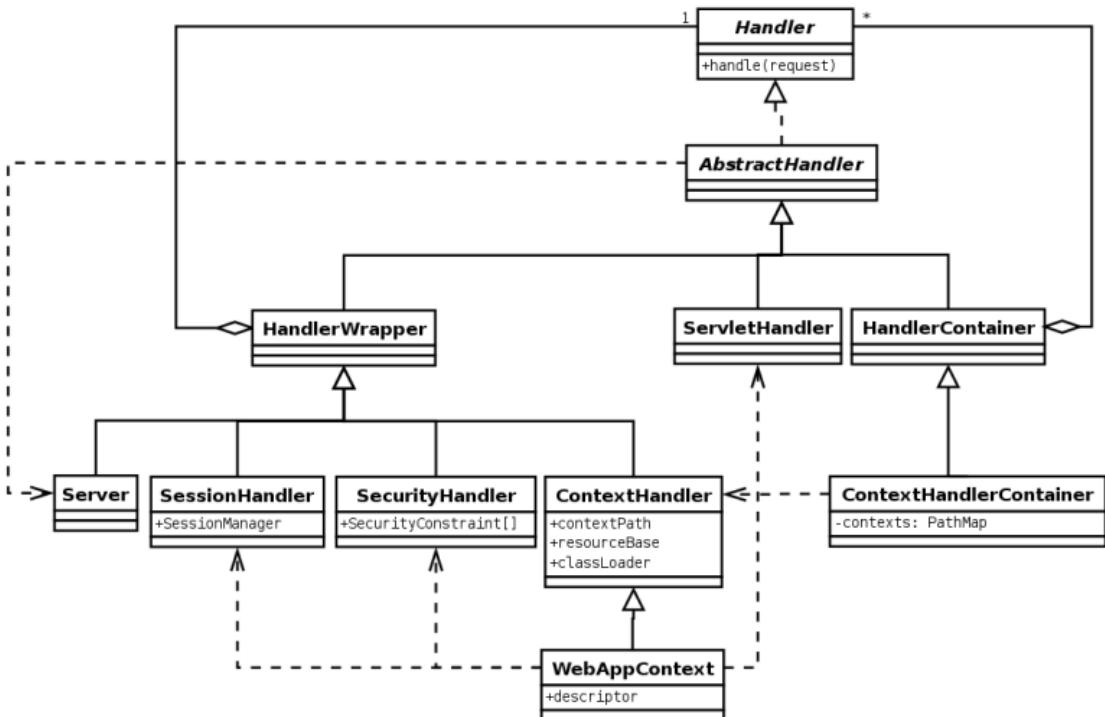
The connectors represent the protocol handlers that accept connections, parse requests and generate responses. The different types of connectors available are based on the protocols, scheduling model, and IO APIs used:



- `SocketConnector` –for few busy connections or when NIO is not available
- `BlockingChannelConnector` –for few busy connections when NIO is available
- `SelectChannelConnector` –for many mostly idle connections or asynchronous handling of Ajax requests
- `SslSocketConnector` –SSL without NIO
- `SslSelectChannelConnector` –SSL with non blocking NIO support
- `AJPConnector` –AJP protocol support for connections from apache mod_jk or mod_proxy_ajp

Handlers

The Handler is the component that deals with received requests. The core API of a handler is the `handle` method:



```

public void handle(String target, Request baseRequest, HttpServletRequest request,
HttpServletResponse response) throws IOException, ServletException
  
```

Parameters:

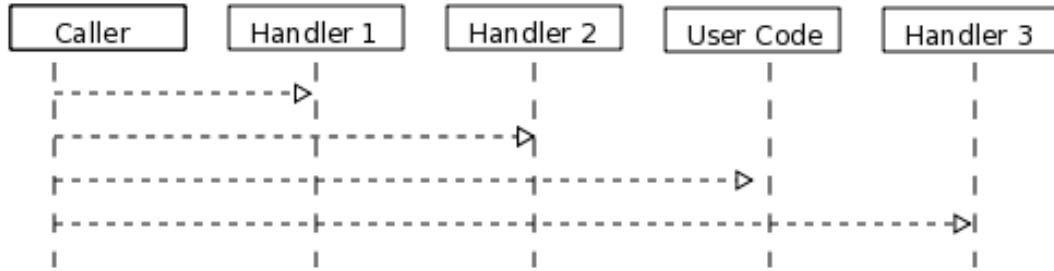
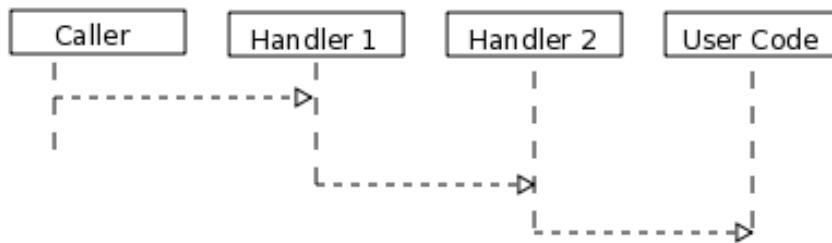
- target—The target of the request, either a URI or a name.
- baseRequest—The original unwrapped request object.
- request—The request either as the Request object or a wrapper of that request. You can use the `HttpConnection.getCurrentConnection()` method to access the Request object if required.
- response—The response as the Response object or a wrapper of that request. You can use the `HttpConnection.getCurrentConnection()` method to access the Response object if required.

An implementation of this method can handle the request, pass the request onto another handler (or servlet) or it might modify and/or wrap the request and then pass it on. This gives three styles of Handler:

- Coordinating Handlers—Handlers that route requests to other handlers (`HandlerCollection`, `ContextHandlerCollection`)
- Filtering Handlers—Handlers that augment a request and pass it on to other handlers (`HandlerWrapper`, `ContextHandler`, `SessionHandler`)
- Generating Handlers—Handlers that produce content (`ResourceHandler` and `ServletHandler`)

Nested Handlers and Handlers Called Sequentially

You can combine handlers to handle different aspects of a request by nesting them, calling them in sequence, or by combining the two models.

Handlers called in sequence:**Nested Handlers**

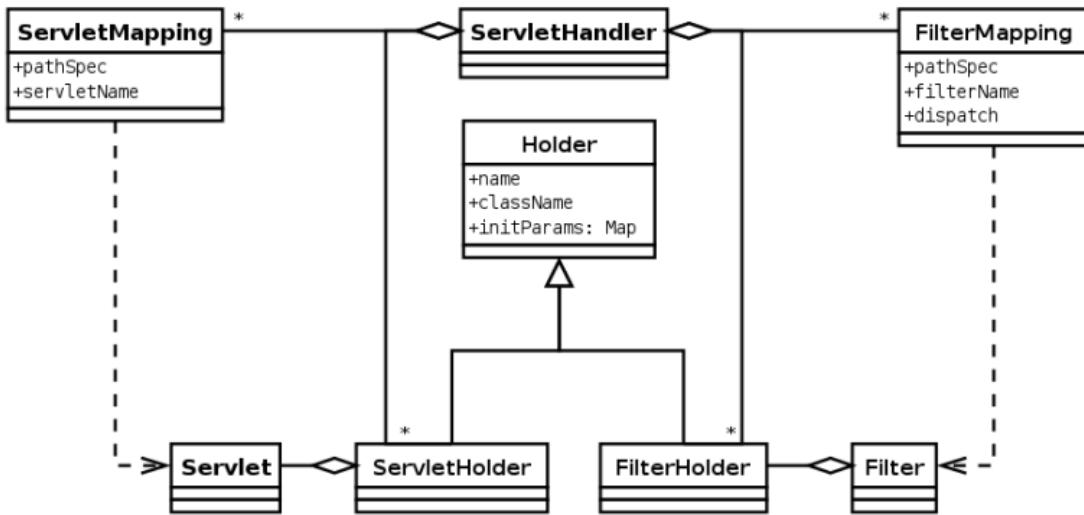
Handlers called in sequence perform actions that do not depend on the next invocation, nor on the handler order. They handle a request and generate the response without interacting with other handlers. The main class for this model is Handler Collection.

Nested handlers are called according to a before/invokeNext/after pattern. The main class for nested handlers is Handler Wrapper. Nested handlers are much more common than those called in sequence.

See also the section called “Writing Custom Handlers”.

Servlet Handler

The `ServletHandler` is a Handler that generates content by passing the request to any configured filters and then to a Servlet mapped by a URI pattern.



A **ServletHandler** is normally deployed within the scope of a **Servlet Context**, which is a **ContextHandler** that provides convenience methods for mapping URIs to servlets.

Filters and Servlets can also use a **RequestDispatcher** to reroute a request to another context or another servlet in the current context.

Contexts

Contexts are handlers that group other handlers below a particular URI context path or a virtual host. Typically a context can have :

- A context path that defines which requests are handled by the context (eg /myapp)
- A resource base for static content (a docroot)
- A class loader to obtain classes specific to the context (typically docroot/WEB-INF/classes)
- Virtual host names

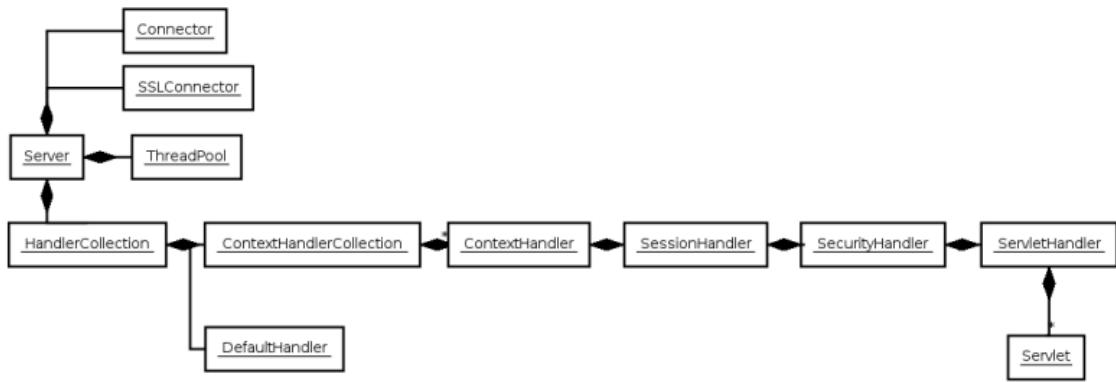
Contexts implementations include:

- **ContextHandler**
- **Servlet Context**
- **Web Application Context**

A web application context combines handlers for security, session and servlets in a single unit that you can configure with a `web.xml` descriptor.

Web Application

A WebApp Context is a derivation of the servlet Context that supports the standardized layout of a web application and configuration of session, security, listeners, filter, servlets, and JSP via a `web.xml` descriptor normally found in the WEB-INF directory of a webapplication.



Essentially the WebAppContext is a convenience class that assists the construction and configuration of other handlers to achieve a standard web application configuration. Configuration is actually done by pluggable implementations of the Configuration class and the prime among these is WebXmlConfiguration.

Jetty Classloading

Class loading in a web container is slightly more complex than a normal Java application. The normal configuration is that each web context (web application or WAR file) has its own classloader, which has the system classloader as its parent. Such a classloader hierarchy is normal in Java, however the servlet specification complicates the hierarchy because it requires the following:

- Classes contained within WEB-INF/lib or WEB-INF/classes have priority over classes on the parent classloader. This is the opposite of the normal behaviour of a Java 2 classloader.
- System classes such as `java.lang.String` are excluded from the webapp priority, and you may not replace them with classes in WEB-INF/lib or WEB-INF/ classes. Unfortunately the specification does not clearly state what classes are *System* classes, and it is unclear if all javax classes should be treated as System classes.
- Server implementation classes like `Server` should be hidden from the web application and should not be available in any classloader. Unfortunately the specification does not state what classes are *Server* classes, and it is unclear if common libraries like the Xerces parser should be treated as Implementation classes.

Configuring Webapp Classloading

Jetty provides configuration options to control the three webapp class loading issues identified above.

You can configure webapp classloading by several methods on the [WebAppContext](#). You can call these methods directly if you are working with the Jetty API, or you can inject methods from a context XML file if you are using the Context Provider (???). You CANNOT set these methods from a `jetty-web.xml` file, as it executes after the classloader configuration is set.

Controlling Webapp Classloader Priority

The method [`org.eclipse.jetty.webapp.WebAppContext.setParentLoaderPriority\(boolean\)`](#) allows control over the priority given to webapp classes over system classes. If you set it to false (the default), Jetty uses standard webapp classloading priority. However, if in this mode some classes that are dependencies of other classes are loaded from the parent classloader (due to settings of system classes below), ambiguities might arise as both the webapp and system classloader versions can end up being loaded.

If set to true, Jetty uses normal JavaSE classloading priority, and gives priority to the parent/system classloader. This avoids the issues of multiple versions of a class within a webapp, but the version the parent/system loader provides must be the right version for all webapps you configure in this way.

Setting System Classes

You can call the methods [`org.eclipse.jetty.webapp.WebAppContext.setSystemClasses\(String Array\)`](#) or [`org.eclipse.jetty.webapp.WebAppContext.addSystemClass\(String\)`](#) to allow fine control over which classes are considered System classes.

- A web application can see a System class.
- A WEB-INF class cannot replace a System class.

The default system classes are:

Table 34.1. Default System Classes

System Classes	
java.	Java SE classes (per servlet spec v2.5 / SRV.9.7.2).
javax.	Java SE classes (per servlet spec v2.5 / SRV.9.7.2).
org.xml	Needed by javax.xml.
org.w3c	Needed by javax.xml.
org.eclipse.jetty.continuation	Webapp continuation not change continuation classes.
org.eclipse.jetty.jndi	Webapp jndi see and not change naming classes.
org.eclipse.jetty.jaas	Webapp jaas see and not change JAAS classes.
org.eclipse.jetty.iosckeyt	Webapp keystore extension.
org.eclipse.jetty.servlet	Webapp default servlet.

Absolute classname can be passed, names ending with . are treated as packages names, and names starting with - are treated as negative matches and must be listed before any enclosing packages.

Setting Server Classes

You can call the methods [`org.eclipse.jetty.webapp.WebAppContext.setServerClasses\(String Array\)`](#) or [`org.eclipse.jetty.webapp.WebAppContext.addServerClass\(String\)`](#) to allow fine control over which classes are considered Server classes.

- A web application cannot see a Server class.
- A WEB-INF class can replace a Server class.

The default server classes are:

Table 34.2. Default Server Classes

Server Classes	
-	Don't hide continuation classes. org.eclipse.jetty.continuation.
-	Don't hide naming classes. org.eclipse.jetty.jndi.
-	Don't hide jaas classes. org.eclipse.jetty.jaas.
-	Don't hide utility servlet classes if provided. org.eclipse.jetty.servlets.

Server Classes	
-	Don't hide default servlet. org.eclipse.jetty.servlet.DefaultServlet
-	Don't hide utility listeners org.eclipse.jetty.servlet.listener.
-	Don't hide websocket extension. org.eclipse.jetty.websocket.
org.eclipse.jetty.	Do hide all other Jetty classes.

Adding Extra Classpaths to Jetty

You can add extra classpaths to Jetty in several ways.

Using start.jar

If you are using ???, at startup the jetty runtime automatically loads option Jars from the top level \$jetty.home/lib directory. The default settings include:

- Adding Jars under \$jetty.home/lib/ext to the system classpath. You can place additional Jars here.
- Adding the directory \$jetty.home/resources to the classpath (may contain classes or other resources).
- Adding a single path defined by the command line parameter *path*.

Using the extraClasspath() method

You can add an additional classpath to a context classloader by calling [org.eclipse.jetty.webapp.WebAppContext.setExtraClasspath\(String\)](#) with a comma-separated list of paths. You can do so directly to the API via a context XML file such as the following:

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
...
<Set name="extraClasspath">../my/classes,../my/jars/special.jar,../my/jars/other.jar
</Set>
...
```

Using a Custom WebAppClassLoader

If none of the alternatives already described meet your needs, you can always provide a custom classloader for your webapp. We recommend, but do not require, that your custom loader subclasses [WebAppClassLoader](#). You configure the classloader for the webapp like so:

```
MyCleverClassLoader myCleverClassLoader = new MyCleverClassLoader();
...
WebAppContext webapp = new WebAppContext();
...
webapp.setClassLoader(myCleverClassLoader);
```

You can also accomplish this in a context xml file.

Starting Jetty with a Custom ClassLoader

If you start a Jetty server using a custom class loader—consider the Jetty classes not being available to the system class loader, only your custom class loader—you may run into class loading issues when

the WebAppClassLoader kicks in. By default the WebAppClassLoader uses the system class loader as its parent, hence the problem. This is easy to fix, like so:

```
context.setClassLoader(new WebAppClassLoader(this.getClass().getClassLoader(), context));
```

or

```
context.setClassLoader(new WebAppClassLoader(new MyCustomClassLoader(), context));
```

Managing 1xx Responses

The [HTTP RFC](#) allows for 1xx informational responses to be sent before a real content response. Unfortunately the servlet specification does not provide a way for these to be sent, so Jetty has had to provide non-standard handling of these headers.

100 Continue

The 100 Continue response should be sent by the server when a client sends a request with a `Expect: 100-continue` header, as the client will not send the body of the request until the 100 continue response has been sent.

The intent of this feature is to allow a server to inspect the headers and to tell the client to not send a request body that might be too large or insufficiently private or otherwise unable to be handled.

Jetty achieves this by waiting until the input stream or reader is obtained by the filter/servlet, before sending the 100 continues response. Thus a filter/servlet may inspect the headers of a request before getting the input stream and send an error response (or redirect etc.) rather than the 100 continues.

102 Processing

[RFC 2518](#) defines the 102 processing response that can be sent "when the server has a reasonable expectation that the request will take significant time to complete. As guidance, if a method is taking longer than 20 seconds (a reasonable, but arbitrary value) to process the server SHOULD return a 102 (Processing) response".

So if a request is received with the `Expect: 102-processing` header, then a filter/servlet may send a 102 response (without terminating further processing) by calling `servletResponse.sendError(102);`.

Creating a Custom Protocol

You can create custom protocols with Jetty. This page provides an example of how to do so, with Telnet as the protocol.

To create a custom Telnet protocol, complete the following tasks:

- Implement a `TelnetServerConnectionFactory`.
- Implement a `TelnetServerConnection` by extending `o.e.j.io.AbstractConnection`.
- Create a parser/interpreter for the bytes you receive (this is totally independent from Jetty).
- If needed, design an API for the application to use to process the bytes received (also independent from Jetty). The API likely has a `respond back` primitive that uses a Jetty provided `EndPoint` and `EndPoint.write(Callback, Buffer...)` to write the response bytes.

Implementing a TelnetServerConnectionFactory

Begin with an `org.eclipse.jetty.server.ServerConnector`, which you can use as is. `ServerConnector` takes a `o.e.j.server.ConnectionFactory`, which creates `o.e.j.io.Connection` objects that interpret the bytes the connector receives. You must implement `ConnectionFactory` with a `TelnetServerConnectionFactory`, where you return a `Connection` implementation (for example, `TelnetServerConnection`).

Implementing the TelnetServerConnection

For the Connection implementation you need to extend from `o.e.j.io.AbstractConnection` because it provides many facilities that you would otherwise need to re-implement from scratch.

For each Connection instance there is associated an `o.e.j.io.EndPoint` instance. Think of `EndPoint` as a specialized version of JDK's `SocketChannel`. You use the `EndPoint` to read, write, and close. You don't need to implement `EndPoint`, because Jetty provides concrete classes for you to use.

The Connection is the *passive* side (that is, Jetty calls it when there is data to read), while the `EndPoint` is the active part (that is, applications call it to write data to the other end). When there is data to read, Jetty calls `AbstractConnection.onFillable()`, which you must implement in your `TelnetServerConnection`.

A typical implementation reads bytes from the `EndPoint` by calling `EndPoint.fill(ByteBuffer)`. For examples, look at both the simpler `SPDYConnection` (in the `SPDY` client package, but server also uses it), and the slightly more complex `HttpConnection`.

Parsing the Bytes Received

After you read the bytes, you need to parse them. For the Telnet protocol there is not much to parse, but perhaps you have your own commands that you want to interpret and execute. Therefore typically every connection has an associated parser instance. In turn, a parser usually emits parse events that a parser listener interprets, as the following examples illustrate:

- In HTTP, the Jetty HTTP parser parses the request line (and emits a parser event), then parses the headers (and emits a parser event for each) until it recognizes the end of the headers (and emits another parser event). At that point, the *interpreter* or parser listener (which for HTTP is `o.e.j.server.HttpChannel`) has all the information necessary to build a `HttpServletRequest` object and can call the user code (the web application, that is, servlets/filters).
- In SPDY, the Jetty SPDY parser parses a SPDY frame (and emits a parser event), and the parser listener (an instance of `o.e.j.spdy.StandardSession`) interprets the parser events and calls user code (application-provided listeners).

With `ConnectionFactory`, `Connection`, `parser`, and `parser listeners` in place, you have configured the read side.

Designing an API to Process Bytes

At this point, server applications typically write data back to the client.

The Servlet API (for HTTP) or application-provided listeners (for SPDY) expose an interface to web applications so that they can write data back to the client. The implementation of those interfaces must link back to the `EndPoint` instance associated with the `Connection` instance so that it can write data via `EndPoint.write(Callback, ByteBuffer...)`. This is an asynchronous call, and it notifies the callback when all the buffers have been fully written.

For example, in the Servlet API, applications use a `ServletOutputStream` to write the response content. `ServletOutputStream` is an abstract class that Jetty implements, enabling Jetty to han-

dle the writes from the web application; the writes eventually end up in an `EndPoint.write(...)` call.

Tips for Designing an API

If you want to write a completely asynchronous implementation, your API to write data to the client must have a callback/promise concept: “Call me back when you are done, and (possibly) give me the result of the computation.”

SPDY’s Stream class is a typical example. Notice how the methods there exist in two versions, a synchronous (blocking) one, and an asynchronous one that takes as last parameter a Callback (if no result is needed), or a Promise (if a result is needed). It is trivial to write the synchronous version in terms of the asynchronous version.

You can use `EndPoint.write(Callback, ByteBuffer...)` in a blocking way as follows:

```
FutureCallback callback = new FutureCallback();
endPoint.write(callback, buffers);
callback.get();
```

With the snippet above your API can be synchronous or asynchronous (your choice), but implemented synchronously.

Chapter 35. Contributing to Jetty

Table of Contents

Community	411
Documentation	412
Source Control and Building	415
Coding Standards	417
Issues, Features, and Bugs	418
Contributing Patches	418
Releasing Jetty	422
Testing a Jetty Release	425

There are many ways to contribute to Jetty, from hardened developers looking to sharpen their skills to neophytes interested in working with an open source project for the first time. Here we show ways to interact with the Jetty community, give feedback to the developers and contribute patches to both core Jetty code and even this document.

Community

Developers and users alike are welcome to engage the Jetty community. We all have day jobs here so don't just ask questions on IRC and frustrated if no one answers right away. Stick around to hear an answer, one is likely coming at some point!

Mailing Lists

We have a number of options for people that wish to subscribe to our mailing lists.

- Jetty Developers List
 - Join - <https://dev.eclipse.org/mailman/listinfo/jetty-dev>
 - Archives - <http://dev.eclipse.org/mhonarc/lists/jetty-dev/>
- Jetty Users List
 - Join - <https://dev.eclipse.org/mailman/listinfo/jetty-users>
 - Archives - <http://dev.eclipse.org/mhonarc/lists/jetty-users/>
- Jetty Announcements List
 - Join - <https://dev.eclipse.org/mailman/listinfo/jetty-announce>
 - Archives - <http://dev.eclipse.org/mhonarc/lists/jetty-announce/>
- Jetty Commit List
 - Join - <https://dev.eclipse.org/mailman/listinfo/jetty-commit>
 - Archives - <http://dev.eclipse.org/mhonarc/lists/jetty-commit/>

Internet Relay Chat - IRC

Much of our conversations about Jetty occur on IRC in one location or another. Users are always welcome to come join our IRC channels and talk with us, other users, or just lurk.

irc.freenode.org - #jetty

Our primary location, we recommend that if your looking to find folks on IRC you try here.

We also have commit notifications coming to this channel on the bottom and top of the hour.

irc.codehaus.org - #jetty

Our prior location before the move to the eclipse foundation. We are idle on here.

Documentation

This document is produced using a combination of maven, git, and docbook. We welcome anyone and everyone to contribute to the content of this book. Below is the information on how to obtain the source of this book and to build it as well as information on how to contribute back to it.

Note: All contributions to this documentation are under the EPL and the copyright is assigned to Mortbay.

Tools

You will need:

git

This project is located at github so if you do not have a GitHub account already you should register for one. You can do that at [Github](#)

You can go one of two ways for using git, if you are familiar with SCM's and the command line interface then feel free to install and use git from there. Otherwise we would recommend you use the github client itself as it will help with some of the workflow involved with working with git.

maven 3

We build the project with maven 3 which can be found at [Apache Maven](#)

Getting Started (cli)

First you need to obtain the source of the documentation project.

Clone the repository:

```
$ git clone git@github.com:jetty-project/jetty-documentation.git
```

You will now have a local directory with all of the jetty-documentation. Now we move on to building it.

```
$ cd jetty-documentation  
$ mvn install
```

While maven is running you may see a lot of files being downloaded. If you are not familiar with maven, then what you are seeing is maven setting up the execution environment for generating the documentation. This build should produce xhtml, as well as any other output formats that we are working with as time goes on. The downloads are all of the java dependencies that are required to make this build work. After a while the downloading will stop and you'll see the execution of the docbkx-maven-plugin.

```
[INFO] --- docbkx-maven-plugin:2.0.13:generate-xhtml (documentation identifier) @ jetty-documentation ---
[INFO] Processing input file: jetty.xml
[INFO] Dumping to /home/jesse/src/projects/jetty/jetty-documentation/target/docbkx/generated/(gen)jetty.xml
[INFO] Applying customization parameters
[INFO] Chunking output. &lt;?xml version="1.0" encoding="UTF-8"?&gt;
[INFO] See /home/jesse/src/projects/jetty/jetty-documentation/target/docbkx/xhtml/jetty
for generated file(s)
[INFO] Executing tasks
[INFO] Executed tasks
```

The build is finished once you see a message akin to this:

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 7.014s
[INFO] Finished at: Tue Oct 25 14:15:37 CDT 2011
[INFO] Final Memory: 14M/229M
[INFO] -----
```

You may now open your web browser and browse to the first page of the html output to see what you have produced! Generally you can do this with File -> Open File -> which will open a file system browsing screen, navigate to your jetty-documentation directory and then further into target/docbkx/xhtml/jetty.html which is the first page of the produced documentation.

Note: if the build is broken, feel free to notify us.

Making Changes

Now that you have built the documentation, you want to edit it and make some changes. We'll now have to take a bit of a step back and look at how git and github works. In the above example you have cloned directly from our canonical documentation repository. Obviously we can not allow anyone immediate access to this repository so you must make a fork of it for your own and then issue back pull requests to build up documentation karma. In English that means that you would go to the url of the documentation in github:

```
https://github.com/jetty-project/jetty-documentation
```

When you are on this page you will see a little button called 'Fork' which you can click and you will be taken back to your main page on github where you have a new repository. When you checkout this repository you are free to commit to your heart's delight all the changes you so direly wish to see in the Jetty documentation. You can clone it to your local machine and build it the same way as above. So let's start small with a little example. Find some paragraph in the documentation that you think needs changed. Locate that in the local checkout and make the change. Now follow the process to push that change back into Jetty proper. Do make sure the change works and the build isn't broken though so make sure you run maven and check the output. Then commit the change.

```
$ git commit -m "Tweaked the introduction to fix a horrid misspelled word." src/docbkx/topics/introduction/topic.xml
```

This will commit the change in your local repository. You can then push the change up to your repository on github.

```
$ git push
```

Now you'll see some output showing that your change has been propagated to your repository on github. In fact if you navigate to that repository at the top of the files list you should see your comment there. Success, your change is now positioned for notifying us about it! If you click on the commit message itself you'll be taken to a screen that shows what files were changed in that commit. In the upper right corner is a button for 'Pull Request'. When you select this and follow the workflow we will then be notified of your contribution and will be able to apply it to our git repository upon review.

Thats it! You have successfully contributed to the documentation efforts of the Jetty project. After enough of these sorts of contributions and building up good community karma, you may be asked to join us as a committer on the documentation.

Conventions

Below is list of conventions that should be followed when developing documentation within this framework. These are not set in stone and should be updated as we learn more.

build before committing

We have an xmlfresh maven plugin that reformats the code to the standards for the project. This is to prevent wide spread merge conflict between commits. The line wrap is set to 120 and the indent is 2 spaces, never tab characters. There are a handful of tags that render inline to the para tags, these are found in the maven configuration of the xmlfresh-maven-plugin.

xml:id

Critically important for being able to generate url's that can be used in a persistent fashion. Without `xml:id`'s the chapters and sections will have generated id which are rooting in obscure location based voodoo. A url using these 'e12c8673' type links will not be durable across generations of the documentation. These `xml:id`'s need to be used on chapters and sections two deep, and anywhere that you intend to cross link deeper.

The `xml:id` values go into a global namespace so they must be unique across the entire document or the last example will win and any cross links will go there. We will have a maven plugin that will fail the build on detecting this soon.

version differences

in general differences in functionality within a major should go into nested sections and use titles like 'Prior to: ##' or 'In version: ##'

license blocks

each xml file should contain the license block that exists in the `jetty.xml` file and a copy has been added to the bottom of this page as well for reference. It should occupy the second through eighteenth lines while the first line should be the xml descriptor.

```
<?xml version="1.0" encoding="utf-8"?>
<!--
// =====
// Copyright (c) 1995-2012 Mort Bay Consulting Pty. Ltd.
// =====
// All rights reserved. This program and the accompanying materials
// are made available under the terms of the Eclipse Public License v1.0
// and Apache License v2.0 which accompanies this distribution.
//
// The Eclipse Public License is available at
// http://www.eclipse.org/legal/epl-v10.html
//
// The Apache License v2.0 is available at
// http://www.opensource.org/licenses/apache2.0.php
-->
```

```
//  
// You may elect to redistribute this code under either of these licenses.  
// ======  
-->
```

Some admonition examples:



Note

A note about the previous case to be aware of.



Important

Important notes are marked with an icon.



Tip

Tips that make your life easier.



Caution

Places where you have to be careful what you are doing.



Warning

Where extreme care has to be taken. Data corruption or other nasty things may occur if these warnings are ignored.

Source Control and Building

If you want to contribute to the development of jetty, you will need to work with a handful of technologies.

Source Control

Jetty uses several development trunks for its artifacts. They are mirrored on github through <http://github.com/eclipse>, or you can look through them via the Eclipse setup at the URLs below.

Primary Interest SCM URLs

These are the URLs to the GIT repositories for the Jetty code. They are for people who are working on the Jetty project, as well as for people who are interested in examining or modifying the Jetty code for their own projects.

Jetty Project Repository

<http://git.eclipse.org/c/jetty/org.eclipse.jetty.project.git>

Build and Project Infrastructure SCM URLs

These are the URLs for Jetty-related code and metadata. These are not needed to use Jetty; these are primarily of use for people who are working with Jetty-the-project (as opposed to using Jetty-the-server in their own projects).

Administrative pom.xml file

<http://git.eclipse.org/c/jetty/org.eclipse.jetty.parent.git>

Build related artifacts that release separately, common assembly descriptors, remote resources, etc.

<http://git.eclipse.org/c/jetty/org.eclipse.jetty.toolchain.git>

Project definition for generating an eclipse p2 update site

<http://git.eclipse.org/c/jetty/org.eclipse.jetty.bundles.git>

Files associated with the development of Jetty -- code styles, formatting, iplogs, etc.

<http://git.eclipse.org/c/jetty/org.eclipse.jetty.admin.git>

Build

Jetty requires the use of Java 7 and the latest releases are always recommended to build.

Jetty uses [Apache Maven 3](#) for managing its build and primary project metadata.

Building Jetty should simply be a matter of changing into the relevant directory and executing the following commands:

```
$ git clone http://git.eclipse.org/gitroot/jetty/org.eclipse.jetty.project.git  
$ cd org.eclipse.jetty.project  
$ mvn install
```

All relevant dependencies will be downloaded into your local repository automatically.



Note

Jetty has a great many test cases that run through the course of its build. Periodically we find some test cases to be more timing dependent than they should be and this results in intermittent test failures. You can help track these down by opening a bug report.

Import into Eclipse

Jetty is a Maven project. To develop Jetty in Eclipse, follow these directions:

Install m2e plugin

1. From the Eclipse menu at the top of the screen, select *Help > Eclipse Marketplace*.
2. Search for *m2e*.
3. Install the *Maven Integration for Eclipse*

Clone the git repository

Using either the egit plugin or git on the commandline (as in the build section above), obtain the jetty source.

Import the Maven Projects

1. From the Eclipse menu, select *File > Import*
2. From the Maven folder, select *Existing Maven Projects*.
3. Click *Next*.
4. In the Import Maven projects pane, click *Browse* and select the top folder of the jetty source tree.
5. Click *Next/Finish* to import all of jetty into Eclipse.

6. Wait for Eclipse and m2e to compile and set up the project.

Coding Standards

Jetty uses number of conventions for its source code.

Code Formatting

Jetty uses the code formatting the following project specifies.

[Eclipse Java Formatting](#)

Code Templates

Jetty specifies the following code templates for use by the project developers.

[Eclipse Code Templates](#)

Code Conventions

The following is an example of the Java formatting and naming styles to apply to Jetty:

```
import some.exact.ClassName;      // GOOD
import some.wildcard.package.*;   // BAD!

package org.always.have.a.package;

/* -----
 * ** Always have some javadoc
 */
class MyClassName
{
    // indent by 4 spaces.
    // use spaced to indent
    // The code must format OK with default tabsize of 8.

    private static final int ALL_CAPS_FOR_PUBLIC_CONSTANTS=1;

    // Field prefixed with __ for static of _ for normal fields.
    // This convention is no longer mandatory, but any given
    // class should either consistently use this style or not.
    private static String __staticField;
    private Object __privateField;

    // use getters and setters rather than public fields.
    public void setPrivateField(Object privateField)
    {
        __privateField=privateField;
    }

    public Object getPrivateField()
    {
        return __privateField;
    }

    public void doSomething()
        throws SomeException
    {
        Object local_variable = __privateField;
        if (local_variable==null)
        {
            // do Something
        }
    }
}
```

Issues, Features, and Bugs

As with any constantly evolving software project, there will be issues, features, and bugs. We want to know what's bugging you!

File bugs under the [RT/Jetty](#) section of [Bugzilla at Eclipse.org](#)

Use the [Guided Bug Reporting Link](#).

Contributing Patches

How to contribute a patch to the jetty project. You should first familiarize yourself with the Eclipse wiki page on [contributing via Git](#).

Sign a CLA

Traditionally Eclipse has been more than a little difficult to contribute code to as someone other than a committer on the given project. The 3 questions would come up over and over again when trying to process non-trivial contributions.

1. Did you author 100% of the content you're contributing?
2. Do you have the rights to contribute this content to Eclipse?
3. Are you willing to contribute the content under the project's license(s) (e.g. EPL)

On every bugzilla that we considered that contained more than a handful of lines of patch code we would have to ask these questions, with good reason since that is sort of important for the overall IP cleanliness argument that Eclipse makes. However it has always been a bit awkward to have the patch ready, waiting and itching to apply, but having to stop at that point and ask the three questions.

Well no longer!

We refer you to this [wonderful blog entry](#) that details out the changes far better than we could here. Long story short, we want you to sign and submit a contributor license agreement before you go any further in the process of contributing a patch. The actual agreement can be seen [on the eclipse legal site](#). If you want to sign the agreement, check out the [Eclipse CLA FAQ](#) and look at item 10 on the list.



Eclipse CLA - For the Impatient

Log into the [Eclipse projects forge](#) (you will need to create an account with the Eclipse Foundation if you have not already done so); click on "Contributor License Agreement"; and Complete the form. Be sure to use the same email address when you register for the account that you intend to use on Git commit records.

Git Diff

The simplest way to contribute a patch is to make a modification to a cloned copy of jetty and then generate a diff between the two versions. We don't really like this approach, but it is difficult to ignore how easy it is for the contributor. Just remember, you still need to create a CLA as mentioned above.

From the top level of the cloned project:

```
$ git diff > #####.patch
```

The hash marks should be the bugzilla issue that you will be attaching the issue to. All patches coming into jetty must come in through bugzilla for IP tracking purposes, even if they are coming in as commits through gerrit. Depending on the size of the patch the patch itself may be flagged as +iplog where it is subject to lawyer review and inclusion with our iplog from here to eternity. We are sorry we are unable to apply patches that we receive via email. So if you have the bugzilla issue created already just attach the issue and feel free to bug us on the section called "Internet Relay Chat - IRC" to take a look. If there is no bugzilla issue yet, create one, make sure the patch is named appropriately and attach it.

When the developer reviews the patch and goes to apply it they will use:

```
$ git apply < #####.patch
```

If you want to be a nice person, test your patch on a clean clone to ensure that it applies cleanly. Nothing frustrates a developer quite like a patch that doesn't apply.

RECOMMENDED - Git Format Patch

Another approach if you want your name in shiny lights in our commit logs is to use the format patch option. With this approach you commit into your cloned copy of jetty and use the git format patch option to generate what looks like an email message containing all of the commit information. This applies as a commit directly when we apply it so it should be obvious that as with the normal diff we must accept these sorts of patches only via bugzilla. Make sure your commit is using the email that you registered in your CLA or no amount of pushing the in world from us will get past the eclipse git commit hooks. When you do your commit to your local repo it is also *vital* that you "sign-off" on the commit using "git commit -s". Without the sign-off, your patch cannot be applied to the jetty repo because it will be rejected by the eclipse git commit hooks.

From the top level of the cloned project:

Make your changes and commit them locally using `git commit -s`:

```
$ git commit -s
```

Then use `git log` to identify the commit(s) you want to include in your patch:

```
commit 70e29326fe904675f772b88a67128c0b3529565e
Author: John Doe <john.doe@who.com>
Date: Tue Aug 2 14:36:50 2011 +0200 353563:
HttpDestinationQueueTest too slow
```

Use `git format-patch` to create the patch:

```
$ git format-patch -M -B 70e29326fe904675f772b88a67128c0b3529565e
```

This will create a single patch file for each commit since the specified commit. The names will start with 0001-[commitmessage].patch. See <http://www.kernel.org/pub/software/scm/git/docs/git-format-patch.html> for details.

When a developer goes to apply this sort of patch then we must assume responsibility for applying it to our codebase from the IP perspective. So we much be comfortable with the providence of the patch

and that it is clear of potential issues. This is not like a diff where you get to edit it and clean up issues before it gets applied. The commit is recorded locally and the developer will then have a chance to make additional commits to address any lingering issues. It is critically important that developers applying these sorts of patches are fully aware of what is being committed and what they are accepting.

To apply the patch the developer will use a command like:

```
$ git am 0001-353563-HttpDestinationQueueTest-too-slow.patch
```

Providing it applies cleanly there will now be a commit in their local copy and they can either make additional commits or push it out.



Note

It is intended that developers are also able to counter-sign the patch by using the '-s' option with the 'git am' command. However as the git hook that processes the commit currently has a bug it is recommended that developers do NOT use the -s option. See https://bugs.eclipse.org/bugs/show_bug.cgi?id=415307

Git Amend

Alternatively, for troublesome patches that do not seem to apply cleanly with git am, you can use git commit --amend to modify the author and signoff the commit. For example:

```
$ git checkout -b patch
$ git apply john-doe.patch
$ git commit -a -m "<Original commit message from John Doe>"
```

At this point the patch is committed with the committer's name on a local branch

```
$ git commit --amend --author "John Doe <john.doe@who.com>" --signoff
```

Now the patch has the right author and it has been signed off

```
$ git checkout master
$ git merge patch
```

Now the local branch has been merged into master with the right author

```
$ git branch -d patch
$ git push
```

Gerrit for Contributors

Prepare yourself for using gerrit by following the steps described here: <https://git.eclipse.org/r/Documentation/user-upload.html>

Then clone the jetty-project:

```
$ git clone ssh://git.eclipse.org:29418/jetty/org.eclipse.jetty.project
```

Stop. Make sure you have a bugzilla issue open for the change that you want to commit into gerrit for review. Also when you are ready to commit your change make sure that your commit starts with the bugzilla id. For example your commit should look like this:

```
$ git commit -m "[Bug 343532] fixed issue" <files>
```

Make sure to properly set the changeIds as described here: <https://git.eclipse.org/r/Documentation/user-changeid.html>

Make your changes, commit them as usual with git. Once done do:

```
$ git push ssh://git.eclipse.org:29418/jetty/org.eclipse.jetty.project HEAD:refs/for/master
```

Note the magic: HEAD:refs/for/master. Without gerrit will not permit you to push.

Review your changes on your gerrit dashboard: <https://git.eclipse.org/r/#/>

Finally get some coffee and relax. You've contributed something to jetty, woohooo. :)

Gerrit for Committers

Using gerrit is pretty simple. Once you have logged in you should first register your interest in the project repositories you want to be notified of patches on. This option is located under your user Settings near your name in the upper right hand corner. Click on 'Watched Projects' and then start typing 'jetty' into the project name box and it will offer completions for the jetty projects.

Now that you are watching the appropriate projects you should get email notifications of commits into gerrit. It is critical that you follow strict committer IP process when reviewing commits. Egregious violations of this process can result into committer status issues.

On the top of the screen click My -> Watched Changes for a list of the commits available for review. These represent people waiting to have their work reviewed. Once you select an issue to review you will see a list of changes at the bottom, click on these and look over the changes. You can have a dialog back and forth with the user through the comments on the patch. Providing they used the ChangeId setup correctly they will be able to issue updates to their patch for further review. Once you are happy with a review change set click on the Review button. This brings up a screen with three important questions.

Verified

Have you verified if this patch is acceptable.

Code Review

How acceptable is this review change set? Are you willing to testify that it is good enough or does it bear further review.

IP Clean

Does this commit follow acceptable IP guidelines? If anything looks suspicious then follow up with comments to clarify any potential issues. Commits with minor changes to existing classes and projects are fine and should reference a corresponding bugzilla id in their commit message.

Completely new features and large contributions require a corresponding CQ reference.

Assuming sufficient approval through the review process there will be an option to Review and Submit and the commit will pass through into jetty proper. You may also review and comment without submitting so the Review button is useful for clearly stating the reviewers comments and concerns regarding the three items mentioned above. Multiple reviewers are able to collaborate on a given patch as well.

Releasing Jetty

There are a number of steps to releasing jetty. It is not just limited to running a couple of maven commands and then moving onto bigger and better things. There are a number of process related issues once the fun maven bits have been completed.

Jetty Release Notes

This release script is for jetty-9 (to release jetty-7 or jetty-8 see older documentation).

1. Pick your version identification strings.

These follow a strict format and will be used when prompted during step 6 below.

```
Release Version          : 9.0.0.v20130322  (v[year][month][day])
Next Development Version : 9.0.1-SNAPSHOT
Tag Name                : jetty-9.0.0.v20130322
```

2. We use the 'release-9' branch to avoid problems with other developers actively working on the master branch.

```
// Get all of the remotes
$ git pull origin
// Create a local tracking branch (if you haven't already)
$ git branch --track release-9 refs/remotes/origin/release-9
// Check out your local tracking branch.
$ git checkout release-9
// Merge from master into the branch (this becomes your point in time
// from master that you will be releasing from)
$ git merge --no-ff master
```

3. Update the VERSION.txt with changes from the git logs, this populates the resolves issues since the last release.

```
$ mvn -N -Pupdate-version
```

4. Edit the VERSION.txt file to set the 'Release Version' at the top alongside the Date of this release.

```
$ vi VERSION.txt
```

5. Make sure everything is commit'd and pushed to git.eclipse.org

```
$ git commit -m "Updating VERSION.txt top section" VERSION.txt  
$ git push origin release-9
```

6. Prepare the Release

NOTE: This step updates the <version> elements in the pom.xml files, does a test build with these new versions, and then commits the pom.xml changes to your local git repo. The `eclipse-release` profile is required on the prepare in order to bring in the jetty aggregates as that profile defines a module which is ignored otherwise.

```
$ mvn release:prepare -DreleaseVersion=9.0.0.v20130322 \  
-DdevelopmentVersion=9.0.1-SNAPSHOT \  
-Dtag=jetty-9.0.0.v20130322 \  
-Peclipse-release
```

7. Perform the Release

NOTE: This step performs the release and deploys it to a oss.sonatype.org staging repository.

```
$ mvn release:perform
```

8. Set up files for next development versions.

Edit `VERSION.txt` for 'Next Development Version' at the top. Do not date this line.

Make sure everything is commit'd and pushed to git.eclipse.org

```
$ vi VERSION.txt  
$ git commit -m "Updating VERSION.txt top section" VERSION.txt  
$ git push origin release-9
```

9. Close the staging repository on oss.sonatype.org

10. Announce stage to the mailing list for testing.

11. Once the staged repository has been approved by the rest of the committers.

a. Release the staging repository to maven central on oss.sonatype.org

b. Merge back the changes in release-9 to master

```
$ git checkout master  
$ git merge --no-ff release-9  
$ git push origin master
```

Building and Deploying Aggregate Javadoc and Xref

Define the jetty.eclipse.website server entry in your .m2/settings.xml file. You'll need to have access to the dev.eclipse.org machine to perform these actions. If you don't know if you have access to this then you probably don't and will need to ask a project leader for help.

To build and deploy the aggregate javadoc and jxr bits:

```
$ cd target/checkout  
$ mvn -Paggregate-site javadoc:aggregate jxr:jxr  
$ mvn -N site:deploy
```

This will generate the aggregate docs and deploy them to the /home/www/jetty/<project version>/jetty-project directory on download.eclipse.org. The last step is to ssh to that machine and adjust the placement of the apidocs and xref directories to remove the jetty-project folder from the url so that we maintain a clean and consistent api url scheme.

```
$ ssh $COMMITTER_ID@dev.eclipse.org  
$ cd /home/data/httpd/download.eclipse.org/jetty/<VERSION>;  
$ mv jetty-project/apidocs .  
$ mv jetty-project/xref .  
$ rmdir jetty-project
```

The end result should be api documents accessible from <http://download.eclipse.org/jetty/<project version>/api-docs> and <http://download.eclipse.org/jetty/<project version>/xref>

Deploying Distribution Files

Since we also provide alternative locations to download jetty distributions we need to copy these in to place. There are a couple of scripts that will take care of this although they need to be localized to your particular execution environment and you need to have authorization to put stuff where it needs to go. These scripts are located at: <http://git.eclipse.org/c/jetty/org.eclipse.jetty.admin.git/tree/release-scripts>. Once these are setup you can deploy a release to eclipse and codehaus with the following incantation.

```
$ ./promote-to-eclipse.sh 9.0.0.v20130322  
( for 7 and 8 releases remember the codehaus part)  
$ ./promote-to-codehaus.sh 8.1.0.v20130322 8.1.0
```

Each of these scripts will download all of the relevant files from maven central and then copy them into the correct location on eclipse and codehaus infrastructure. On the eclipse side of it they will also adjust the documentation links if they remain broken as well as regenerate all of the html files on the eclipse download site.

Updating Stable Links

Since we are not allowed to have symbolic links on the download site we have to log into the machine manually and remove the previous stable directory and update it with a new release. Maintaining the

conventions we use on the site will allow all 'stable' links to be stable and not needed to update to the latest major Jetty build version:

```
$ ssh <user>@build.eclipse.org
$ cd ~downloads/jetty/
$ rm -Rf stable-9
$ cp -r <version> stable-9
$ ./index.sh
```

This needs to be done for all Eclipse Jetty releases (regardless of version). In addition we have to work to reduce the footprint of jetty on the primary eclipse download resources so we want to move older releases to the eclipse archive site.

```
$ cd ~/downloads/jetty
$ mv <old release> /home/data/httpd/archive.eclipse.org/jetty/
```

Periodically we need to do the same for the osgi P2 repositories to keep the size of our downloads directory at a reasonable size.

Release Documentation

There are two git repositories you need to be aware of for releasing jetty-documentation. The jetty-documentation is located in our github repository and the jetty-website is located at eclipse.

jetty-documentation

<https://github.com/jetty-project/jetty-documentation>

jetty-website

<http://git.eclipse.org/c/www.eclipse.org/jetty.git>

Since we want an firm documentation url:

1. checkout the jetty-website and copy the contents of documentation/current into a directory called documentation/\$version
2. Edit the index.html file in the documentation directory and add the newly released documentation url. Make sure you follow the other examples and include the rel="nofollow" attribute on the link so that search engines do not crawl newly created documentation, otherwise we are subject to duplicate content penalties in SEO.

Lastly, the \${project.version} of the pom.xml is the latest released version of the Jetty so when making a new release we want to update this so that our documentation is being generated with useful defaults in many of our examples. Throughout much of the documentation we use \${project.version} which is resolved in the build to make examples that can be cut and paste straight out of the documentation and automatically use the latest version available, so once the release is available in maven central update the version in the documentation's pom.xml and push it. The automated build will update the [documentation/current/](#) url.

Testing a Jetty Release

To test a Jetty release, complete the following steps for each release you want to test:

1. Download the staged release:

```
wget https://oss.sonatype.org/content/repositories/jetty-[reponumber]/org/eclipse/jetty/jetty-distribution/[jetty-version]/jetty-distribution-9.[jetty-minor-version].tar.gz
```

2. Extract to a directory of your choice.

3. Start jetty:

```
cd [installdir] ; java -jar start.jar
```

4. If there are no exceptions, proceed. Otherwise, investigate.

5. Open http://localhost:8080/ in a browser. In the examples section click "Test Jetty Webapp". You should see the test.war webapp.

6. Go through ALL the tests and verify that everything works as expected.

7. In the examples section click "JAAS Test" and verify that everything works as expected.

8. In the examples section click "JNDI Test" and verify that everything works as expected.

9. In the examples section click "Servlet 3.1 Test" and verify that everything works as expected.

10. Verify that hot deployment works.

```
cd [installdir] ;  
touch [pathToJettyDistribution]/webapps/demo/test.xml
```

11. Verify that test.war gets redeployed in STDOUT.

12. Verify that the spdy example webapp and spdy-proxy do work

```
cd jetty_src/jetty-spdy/spdy-example-webapp  
mvn jetty:run-forked
```

13. Browse to https://localhost:8443 and verify that all looks ok

14. Stop the server with CTRL+C and restart it in proxy mode:

```
mvn -Pproxy jetty:run-forked
```

15. Browse to http://localhost:8080 and https://localhost:8443 and verify that all looks ok

Testing CometD

1. Clone CometD.

```
clone git://github.com/cometd/cometd.git
git clone git://github.com/cometd/cometd.git
```

2. Edit pom.xml and update jetty-version.

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <jetty-version>7.6.2.v20120308</jetty-version>
    <jetty-plugin-version>${jetty-version}</jetty-plugin-version>
    <slf4j-version>1.6.4</slf4j-version>
    <spring-version>3.1.0.RELEASE</spring-version>
</properties>
<repositories>
<repository>
    <id>Jetty Staging</id>
    <url>https://oss.sonatype.org/content/repositories/jetty-988/</url>
</repository>
</repositories>
```

3. Build Cometd:

```
mvn clean install
```

4. Be patient.

5. Run the loadtest as it is described here: <http://cometd.org/documentation/2.x/howtos/loadtesting>. Keep the default values, but make sure that you raise the clients setting to 1000. Run the loadtest until "JIT compilation time" is close to a value of zero (about 10k calls).
6. Make sure that the performance results are reasonably fast. On a MacBook Pro i7 2.6ghz dualcore produces the following:

```
=====
Statistics Started at Fri Mar 09 13:44:35 CET 2012
Operative System: Mac OS X 10.7.3 amd64
JVM : Oracle Corporation Java HotSpot(TM) 64-Bit Server VM runtime 23.0-b16 1.7.0_04-ea-b14
Processors: 4
System Memory: 99.583336% used of 30.0 GiB
Used Heap Size: 36.490677 MiB
Max Heap Size: 1920.0 MiB
Young Generation Heap Size: 896.0 MiB
-----
Testing 1000 clients in 100 rooms, 10 rooms/client
Sending 1000 batches of 10x50 bytes messages every 10000 ?s
[GC [PSYOUNGGEN: 786432K->8736K(917504K)] 823650K->45954K(1966080K), 0.0309940 secs]
[Times: user=0.09 sys=0.00, real=0.03 secs]
[GC [PSYOUNGGEN: 795168K->11424K(917504K)] 832386K->48642K(1966080K), 0.0513360 secs]
[Times: user=0.13 sys=0.00, real=0.05 secs]
[GC [PSYOUNGGEN: 797856K->14560K(917504K)] 835074K->51778K(1966080K), 0.0432940 secs]
[Times: user=0.12 sys=0.00, real=0.05 secs]
[GC [PSYOUNGGEN: 800992K->15680K(917504K)] 838210K->52898K(1966080K), 0.0491200 secs]
[Times: user=0.14 sys=0.00, real=0.05 secs]
[GC [PSYOUNGGEN: 802112K->17568K(917504K)] 839330K->54786K(1966080K), 0.0484950 secs]
[Times: user=0.14 sys=0.00, real=0.05 secs]
[GC [PSYOUNGGEN: 804000K->17600K(917504K)] 841218K->54818K(1966080K), 0.0456460 secs]
[Times: user=0.14 sys=0.01, real=0.05 secs]
```

```
[GC [PSYOUNG: 804032K->19488K(917504K)] 841250K->56706K(1966080K), 0.0542000 secs]
[Times: user=0.15 sys=0.00, real=0.05 secs]
[GC [PSYOUNG: 805920K->20224K(917504K)] 843138K->57442K(1966080K), 0.0486350 secs]
[Times: user=0.16 sys=0.00, real=0.05 secs]
[GC [PSYOUNG: 806656K->20192K(917504K)] 843874K->57410K(1966080K), 0.0566690 secs]
[Times: user=0.15 sys=0.00, real=0.06 secs]
[GC [PSYOUNG: 806624K->21152K(917504K)] 843842K->58370K(1966080K), 0.0536740 secs]
[Times: user=0.16 sys=0.00, real=0.06 secs]
[GC [PSYOUNG: 807584K->21088K(917504K)] 844802K->58306K(1966080K), 0.0576060 secs]
[Times: user=0.18 sys=0.00, real=0.06 secs]
[GC [PSYOUNG: 807520K->22080K(917504K)] 844738K->59298K(1966080K), 0.0663300 secs]
[Times: user=0.19 sys=0.01, real=0.06 secs]
-----
Statistics Ended at Fri Mar 09 13:45:21 CET 2012
Elapsed time: 45826 ms
    Time in JIT compilation: 52 ms
    Time in Young Generation GC: 606 ms (12 collections)
    Time in Old Generation GC: 0 ms (0 collections)
Garbage Generated in Young Generation: 9036.513 MiB
Garbage Generated in Survivor Generation: 21.65625 MiB
Garbage Generated in Old Generation: 0.0 MiB
Average CPU Load: 156.54865/400
-----

Outgoing: Elapsed = 45820 ms | Rate = 218 messages/s - 21 requests/s - ~0.083 Mib/s
Waiting for messages to arrive 996960/999045
All messages arrived 999045/999045
Messages - Success/Expected = 999045/999045
Incoming - Elapsed = 45945 ms | Rate = 21743 messages/s - 9496 responses/s(43.68%) - ~8.295 Mib/s
Messages - Wall Latency Distribution Curve (X axis: Frequency, Y axis: Latency):
@           _ 24 ms (8765, 0.88%)
@           _ 45 ms (58952, 5.90%)
@           _ 67 ms (87065, 8.71%)
@           _ 88 ms (113786, 11.39%)
@           _ 109 ms (167426, 16.76%)
@           _ 131 ms (176163, 17.63%) ^50%
@           _ 152 ms (123182, 12.33%)
@           _ 174 ms (90918, 9.10%)
@           _ 195 ms (67209, 6.73%) ^85%
@           _ 216 ms (46989, 4.70%)
@           _ 238 ms (24975, 2.50%) ^95%
@           _ 259 ms (16509, 1.65%)
@           _ 281 ms (8454, 0.85%) ^99%
@           _ 302 ms (4324, 0.43%)
@           _ 323 ms (2955, 0.30%)
@           _ 345 ms (957, 0.10%) ^99.9%
@           _ 366 ms (204, 0.02%)
@           _ 388 ms (144, 0.01%)
@           _ 409 ms (25, 0.00%)
@           _ 430 ms (43, 0.00%)
Messages - Wall Latency 50th%/99th% = 117/275 ms
Messages - Wall Latency Min/Ave/Max = 2/123/430 ms
Messages - Network Latency Min/Ave/Max = 1/114/417 ms
Thread Pool - Concurrent Threads max = 239 | Queue Size max = 1002 | Queue Latency avg/max = 12/101 ms
```

- Deploy cometd.war to the webapps directory of the jetty-distribution tested above.

```
cp cometd-demo/target/cometd-demo-[version].war [pathToJetty]/jetty-distribution-[jetty-version]/webapps/
```

- Start jetty and make sure there are no exceptions.

```
cd [pathToJetty] && java -jar start.jar
```

9. Go through all pages of the demo and test them:

```
http://localhost:8080/cometd-demo-2.4.1-SNAPSHOT/
```

If all tests are green, you are done!

Chapter 36. Reference Section

Table of Contents

Jetty XML Syntax	430
Jetty XML Usage	442
jetty.xml	443
jetty-web.xml	444
jetty-env.xml	445
webdefault.xml	446
Jetty override-web.xml	448

Jetty XML Syntax

The Jetty XML syntax is a straightforward mapping of XML elements to a Java API so that POJOs can be instantiated and getters, setters, and methods called. It is very similar to Inversion Of Control (IOC) or Dependency Injection (DI) frameworks like Spring or Plexus (but it predates all of them). Typically Jetty XML is used by `jetty.xml` to configure a Jetty server or by a `context.xml` file to configure a ContextHandler or subclass, but you can also use the mechanism to configure arbitrary POJOs.

This page describes the basic syntax of Jetty XML configuration. See [Jetty XML Usage](#) for information on how you can use and combine Jetty XML. See configuration files for specific examples.

Basic XML Configuration File Example

The following XML configuration file creates some Java objects and sets some attributes:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure.dtd">
<Configure id="foo" class="com.acme.Foo">
    <Set name="name">demo</Set>
    <Set name="nested">
        <New id="bar" class="com.acme.Bar">
            <Arg>true</Arg>
            <Set name="wibble">10</Set>
            <Set name="wobble">xyz</Set>
            <Set name="parent"><Ref refid="foo"/></Set>
            <Call name="init">
                <Arg>false</Arg>
            </Call>
        </New>
    </Set>

    <Ref refid="bar">
        <Set name="wibble">20</Set>
        <Get name="parent">
            <Set name="name">demo2</Set>
        </Get>
    </Ref>
</Configure>
```

The XML above is equivalent to the following Java code:

```
com.acme.Foo foo = new com.acme.Foo();
foo.setName("demo");

com.acme.Bar bar = new com.acme.Bar(true);
bar.setWibble(10);
bar.setWobble("xyz");
```

```

bar.setParent(foo);
bar.init(false);

foo.setNested(bar);

bar.setWibble(20);
bar.getParent().setName("demo2");

```

Overview

Understanding DTD and Parsing

The document type descriptor ([configure.dtd](#)) describes all valid elements in a Jetty XML configuration file using the Jetty IoC format. The first two lines of an XML must reference the DTD to be used to validate the XML like:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN" "http://
www.eclipse.org/jetty/configure_9_0.dtd">
...

```

Typically a good XML editor will fetch the DTD from the URL and use it to give syntax highlighting and validation while a configuration file is being edited. Some editors also allow DTD files to be locally cached. The URL may point to configure.dtd if you want the latest current version, or to a specific version like configure_9_0.dtd if you want a particular validation feature set.

Files that conform to the configure.dtd format are processed in Jetty by the [XmlConfiguration](#) class which may also validate the XML (using a version of the DTD from the classes jar file), but is by default run in a forgiving mode that tries to work around validation failures.

Jetty XML Configuration Scope

The configuration of object instances with Jetty IoC XML is done on a scoped basis, so that for any given XML element there is a corresponding Object in scope and the nested XML elements apply to that. The outer most scope is given by a Configure element and elements like Call, New and Get establish new scopes. The following example uses the name fields to explain the scope

```

<Configure class="com.example.Foo">
  <Set name="fieldOnFoo">value</Set>
  <Set name="fieldOnFoo">
    <New class="com.example.Bar">
      <Set name=fieldOnBar>value</Set>
      <Call name="methodOnBarWithNoArgs"/>
    </New>
  </Set>

  <Call name="methodOnFoo">
    <Arg>value for first arg of methodOnFoo</Arg>
    <Arg><New class="com.example.Bar"/></Arg>
    <Set name="fieldOnObjectReturnedByMethodOnFoo">value</Set>
    <Call name="methodOnObjectReturnedByMethodOnFooWithNoArgs"/>
  </Call>
</Configure>

```

Coercing Arguments to a Type

When trying to match XML elements to java elements, Jetty XmlConfiguration may need to coerce values to match method arguments. By default it does so on a best effort basis, but you can also

specify explicit types with the `type` attribute. Supported values for type are: String, Character, Short, Byte, Integer, Long, Boolean, Float, Double, char, short, byte, int, long, boolean, float, double, URL, InetAddress, InetAddrPort, void

Referring to a Class

If you do not specify the classname, Jetty assumes you are calling the method on the object that is current in scope (eg the object of the surrounding Configure, New or Get clause). If the class attribute is specified to a fully-qualified class name, then it is either used to create a new instance (Configure and New elements) or is used to access a static (Call, Set or Get elements).

Referring to an Object

You can use the `id` attribute to store a reference to the current object when first creating or referring to this object. You can then use the [Ref element](#) to reference the object later. The `id` must be unique for each object you create.

<Configure>

This is the root element that specifies the class of object that is to be configured. It is usually either the Server, in `jetty.xml`, or a WebAppContext in `jetty-web.xml`.

Attribute	Required	Description
<code>id</code>	no	A reference to the object that was created. If you define multiple Configure elements with the same <code>id</code> , they will be treated as one object, even if they're in different files. You can use this to break up configuration of an object (such as the Server) across multiple files.
<code>class</code>	no	The fully qualified class-name of the object to be configured. Could be <code>org.eclipse.jetty.server.Server</code> , <code>org.eclipse.jetty.webapp.WebAppContext</code> , a handler, etc.

Can Contain

[Set element](#), [Get element](#), [Put element](#), [Call element](#), [New element](#), [Ref element](#), [Array element](#), [Map element](#), [Property element](#)

Examples

Basic Example

```
<Configure class="org.eclipse.jetty.server.Server">
  <Set name="port">8080</Set>
</Configure>
```

This is equivalent to:

```
org.eclipse.jetty.server.Server server = new org.eclipse.jetty.server.Server();
server.setPort(8080);
```

Using id to break up configuration of one object across multiple files

(etc/jetty.xml)

```
<Configure id="Server" class="org.eclipse.jetty.server.Server">
  <!-- basic configuration here -->
</Configure>
```

(etc/jetty-logging.xml)

```
<Configure id="Server" class="org.eclipse.jetty.server.Server">
  <!-- assumes that you have the basic server configuration set up; this file only
       contains additional configuration for logging -->
</Configure>
```

Then run the combined configuration using:

```
java -jar start.jar etc/jetty.xml jetty-logging.xml
```

<Set>

A Set element maps to a call to a setter method or field on the current object. It can contain text and/or elements such as Call, New, SystemProperty, etc., as values. The name and optional type attributes are used to select the setter method. If you do not specify a value type, white space is trimmed out of the value. If it contains multiple elements as values, they are added as strings before being converted to any specified type.

Attribute	Required	Description
name	yes	the name of the setter method to call, or the field to set. If the name given is xxx, then a setXxx method is used. If the setXxx method cannot be found, then the xxx field is used.
type	no	the declared type of the argument. See also discussion of type for Arg for how to define null and empty string values.
class	no	if present, then this Set is treated as a static set method invocation

Can Contain

value text, [Get element](#), [Call element](#), [New element](#), [Ref element](#), [Array element](#), [Map element](#), [System Property element](#), [Property element](#)

Examples

Basic Example

```
<Configure id="server" class="org.eclipse.jetty.server.Server">
  <Set name="port">8080</Set>
</Configure>
```

Set via a System Property

```
<Configure id="server" class="org.eclipse.jetty.server.Server">
```

```
<Set name="port"><SystemProperty name="jetty.port" /></Set>
</Configure>
```

Creating a NewObject and Setting It on the Server

```
<Configure id="server" class="org.eclipse.jetty.server.Server">
  <Set name="threadPool">
    <New class="org.eclipse.jetty.util.thread.QueuedThreadPool">
      <Set name="minThreads">10</Set>
      <Set name="maxThreads">1000</Set>
    </New>
  </Set>
</Configure>
```

This is equivalent to:

```
org.eclipse.jetty.server.Server server = new org.eclipse.jetty.server.Server();
org.eclipse.jetty.util.thread.QueuedThreadPool threadPool = new
  org.eclipse.jetty.util.thread.QueuedThreadPool();
threadPool.setMinThreads(10);
threadPool.setMaxThreads(1000);

server.setThreadPool(threadPool);
```

Invoking a Static Setter

```
<Configure id="server" class="org.eclipse.jetty.server.Server">
  <Set class="org.eclipse.jetty.util.log.Log" name="logToParent">loggerName</Set>
</Configure>
```

<Get>

A Get element maps to a call to a getter method or field on the current object. It can contain nested elements such as Set, Put, Call, etc.; these act on the object returned by the Get call.

Attribute	Required	Description
name	yes	the name of the getter method to call, or the field to get. If the name given is xxx, then a getXxx method is used. If the getXxx method cannot be found, then the xxx field is used.
class	no	if present, then this Get is treated as a static getter or field.
id	no	if present, then you can use this id to refer to the returned object later.

Can Contain

[Set element](#), [Get element](#), [Put element](#), [Call element](#), [New element](#), [Ref element](#), [Array element](#), [Map element](#), [Property element](#)

Examples

Basic Example

This simple example doesn't do much on its own. You would normally use this in conjunction with a <Ref id="Logger" />.

```
<Configure id="server" class="org.eclipse.jetty.server.Server">
  <Get id="Logger" class="org.eclipse.jetty.util.log.Log" name="log"/>
</Configure>
```

Invoking a Static Getter and Call Methods on the Returned Object

```
<Configure id="server" class="org.eclipse.jetty.server.Server">
  <Get class="java.lang.System" name="out">
    <Call name="println">
      <Arg>Server version
    is: <Get class="org.eclipse.jetty.server.Server" name="version"/></Arg>
    </Call>
  </Get>
</Configure>
```

<Put>

A Put element maps to a call to a put method on the current object, which must implement the Map interface. It can contain text and/or elements such as Call, New, SystemProperty, etc. as values. If you do not specify a no value type, white space is trimmed out of the value. If it contains multiple elements as values, they are added as strings before being converted to any specified type.

Attribute	Required	Description
name	yes	used as the put key
type	no	forces the type of the value. See also discussion of type for Arg for how to define null and empty string values.

Can Contain

[value text](#) , [Get element](#), [Call element](#), [New element](#), [Ref element](#), [Array element](#), [Map element](#), [System Property element](#), [Property element](#)

Example

```
<Get name="someKindOfMap">
  <Put name="keyName">objectValue</Put>
</Get>
```

<Call>

A Call element maps to an arbitrary call to a method on the current object. It can contain a sequence of Arg elements followed by a sequence of configuration elements, such as Set, Put, Call. The <Arg>s are passed as arguments to the method; the sequence of configuration elements act on the object returned by the original call.

Attribute	Required	Description
name	yes	the name of the arbitrary method to call. The method called will use the exact name you provide it.
class	no	if present, then this Call is treated as a static method.
id	no	if present, you can use this id to refer to any object returned by the call, for later use.

Can Contain

[Arg element](#), [Set element](#), [Get element](#), [Put element](#), [Call element](#), [New element](#), [Ref element](#), [Array element](#), [Map element](#), [Property element](#)

Examples

Basic example

```
<Call name="doFoo">
  <Arg>bar</Arg>
  <Set name="test">1, 2, 3</Set>
</Call>
```

This is equivalent to:

```
Object o2 = o1.doFoo("bar");
o2.setTest("1, 2, 3");
```

Invoking a static method

```
<Call class="com.acme.Foo" name="setString">
  <Arg>somestring</Arg>
</Call>
```

which is equivalent to:

```
com.acme.Foo.setString("somestring");
```

Invoking the Actual Method Instead of Relying on Getter/Setter Magic

```
<Configure id="Server" class="org.eclipse.jetty.server.Server">
  <Call name="getPort" id="port" />
  <Call class="com.acme.Environment" name="setPort">
    <Arg>
      <Ref refid="port"/>
    </Arg>
  </Call>
</Configure>
```

which is equivalent to:

```
org.mortbay.jetty.Server server = new org.mortbay.jetty.Server();
com.acme.Environment.setPort( server.getPort() );
```

<Arg>

An Arg element can be an argument of either a method or a constructor. Use it within ??? and ???.

It can contain text and/or elements, such as Call, New, SystemProperty, etc., as values. The optional type attribute can force the type of the value. If you don't specify a type, white space is trimmed out of the value. If it contains multiple elements as values, they are added as strings before being converted to any specified type.

Attribute	Required	Description
type	no	force the type of the argument. If you do not provide a value for the element, if you use type of

Attribute	Required	Description
		"String", the value will be the empty string (""), otherwise it is null.

Can Contain

value text, [Get element](#), [Call element](#), [New element](#), [Ref element](#), [Array element](#), [Map element](#), [System Property element](#), [Property element](#)

Examples

Basic examples

```
<Arg>foo</Arg> <!-- String -->
<Arg>true</Arg> <!-- Boolean -->
<Arg>1</Arg> <!-- int, long, short, float, double -->
<Arg><Ref refid="foo" /></Arg> <!-- any object; reference a previously created object
with id "foo", and pass it as a parameter -->
<Arg></Arg> <!-- null value -->
<Arg type="String"></Arg> <!-- empty string "" -->
```

Coercing Type

This explicitly coerces the type to a boolean:

```
<Arg type="boolean">False</Arg>
```

As a Parameter

Here are a couple of examples of [Arg element](#) being used as a parameter to methods and to constructors:

```
<Call class="com.acme.Environment" name="setFoo">
  <Arg>
    <New class="com.acme.Foo">
      <Arg>bar</Arg>
    </New>
  </Arg>
</Call>
```

This is equivalent to:

```
com.acme.Environment.setFoo(new com.acme.Foo("bar"));
```

```
<New class="com.acme.Baz">
  <Arg>
    <Call id="bar" class="com.acme.MyStaticObjectFactory" name="createObject">
      <Arg>2</Arg>
    </Call>
  </Arg>
</New>
```

This is equivalent to:

```
new com.acme.Baz(com.acme.MyStaticObjectFactory.createObject(2));
```

<New>

Instantiates an object. Equivalent to new in Java, and allows the creation of a new object. A New element can contain a sequence of [Arg element](#)'s, followed by a sequence of configuration elements

(Set, Put, etc). [Arg element](#)'s are used to select a constructor for the object to be created. The sequence of configuration elements then acts on the newly-created object.

Attribute	Required	Description
class	yes	fully qualified classname, which determines the type of the new object that is instantiated.
id	no	gives a unique name to the object which can be referenced later by Ref elements.

Can Contain

[Arg element](#), [Set element](#), [Get element](#), [Put element](#), [Call element](#), [New element](#), [Ref element](#), [Array element](#), [Map element](#), [Property element](#)

Examples

Basic example

```
<New class="com.acme.Foo">
  <Arg>bar</Arg>
</New>
```

which is equivalent to:

```
com.acme.Foo foo = new com.acme.Foo("bar");
```

Instantiate with the Default Constructor

```
<New class="com.acme.Foo" />
```

which is equivalent to:

```
com.acme.Foo foo = new com.acme.Foo();
```

Instantiate with Multiple Arguments, Then Configuring Further

```
<New id="foo" class="com.acme.Foo">
  <Arg>bar</Arg>
  <Arg>baz</Arg>
  <Set name="test">1, 2, 3</Set>
</New>
```

which is equivalent to:

```
Object foo = new com.acme.Foo("bar", "baz");
foo.setTest("1, 2, 3");
```

<Ref>

A Ref element allows a previously created object to be referenced by a unique id. It can contain a sequence of elements, such as Set or Put which then act on the referenced object. You can also use a Ref element as a value for other elements such as Set and Arg.

The Ref element provides convenience and eases readability. You can usually achieve the effect of the Ref by nesting elements (method calls), but this can get complicated very easily. The Ref element

makes it possible to refer to the same object if you're using it multiple times, or passing it into multiple methods. It also makes it possible to split up configuration across multiple files.

Attribute	Required	Description
refid	yes	the unique identifier used to name a previously created object.

Can Contain

[Set element](#), [Get element](#), [Put element](#), [Call element](#), [New element](#), [Ref element](#), [Array element](#), [Map element](#), [Property element](#)

Examples

Basic example

Use the referenced object as an argument to a method call or constructor:

```
<Get id="foo" name="xFoo" />
<Set name="test"><Ref refid="foo"/></Set>
```

This is equivalent to:

```
foo = getXFoo();
setSomeMethod(foo);
```

Manipulating the Object Returned by Ref

```
<Get id="foo" name="xFoo" />
<Ref refid="foo">
  <Set name="test">1, 2, 3</Set>
</Ref>
```

This is equivalent to:

```
foo = getXFoo();
foo.setTest("1, 2, 3");
```

Ref vs. Nested Elements

Here is an example of the difference in syntax between using the Ref element, and nesting method calls. They are exactly equivalent:

```
<!-- using Ref in conjunction with Get -->
<Configure id="Server" class="org.eclipse.jetty.server.Server">
  <Get id="Logger" class="org.eclipse.jetty.util.log.Log" name="log"/>
  <Ref refid="Logger">
    <Set name="debugEnabled">true</Set>
  </Ref>
</Configure>
<!-- calling the setter directly on the object returned by Get -->
<Configure id="Server" class="org.eclipse.jetty.server.Server">
  <Get class="org.eclipse.jetty.util.log.Log" name="log">
    <Set name="debugEnabled">true</Set>
  </Get>
</Configure>
```

Here is a more practical example, taken from the handler configuration section in `etc/jetty.xml`:

```
<Set name="handler">
```

```

<New id="Handlers" class="org.eclipse.jetty.server.handler.HandlerCollection">
  <Set name="handlers">
    <Array type="org.eclipse.jetty.server.Handler">
      <Item>
        <!-- create a new instance of a ContextHandlerCollection named "Contexts" -->

<New id="Contexts" class="org.eclipse.jetty.server.handler.ContextHandlerCollection"/>
  </Item>
  <Item>

<New id="DefaultHandler" class="org.eclipse.jetty.server.handler.DefaultHandler"/>
  </Item>
  <Item>
    <!-- create a new instance of a RequestLogHandler named "RequestLog" -->

<New id="RequestLog" class="org.eclipse.jetty.server.handler.RequestLogHandler"/>
  </Item>
  </Array>
</Set>
</New>
</Set>

<Call name="addBean">
  <Arg>
    <New class="org.eclipse.jetty.deploy.ContextDeployer">
      <!-- pass in the ContextHandlerCollection object ("Contexts") that was created
earlier, as an argument -->
      <Set name="contexts"><Ref refid="Contexts"/></Set>
    </New>
  </Arg>
</Call>

<!-- configure the RequestLogHandler object ("RequestLog") that we created earlier -->
<Ref refid="RequestLog">
  ...
</Ref>

```

<Array>

An Array element allows the creation of a new array.

Attribute	Required	Description
type	no	specify what types of items the array can contain.
id	no	unique identifier you can use to refer to the array later.

Can Contain

[Item element](#)

Example

```

<Array type="java.lang.String">
  <Item>value0</Item>
  <Item><New class="java.lang.String"><Arg>value1</Arg></New></Item>
</Array>

```

This is equivalent to:

```
String[] a = new String[] { "value0", new String("value1") };
```

<Item>

An Item element defines an entry for Array and Map elements.

Attribute	Required	Description
type	no	force the types of value.
id	no	unique identifier that you can use to refer to the array later.

Can Contain

[Get element](#), [Call element](#), [New element](#), [Ref element](#), [Array element](#), [Map element](#), [System Property element](#), [Property element](#)

<Map>

A Map element allows the creation of a new HashMap and to populate it with (key, value) pairs.

Attribute	Required	Description
id	no	unique identifier you can use to refer to the map later.

Can Contain

[Entry element](#)

Example

```
<Map>
  <Entry>
    <Item>keyName</Item>
    <Item><New class="java.lang.String"><Arg>value1</Arg></New></Item>
  </Entry>
</Map>
```

This is equivalent to:

```
Map m = new HashMap();
m.put("keyName", new String("value1"));
```

<Entry>

An Entry element contains a key-value [Item element](#) pair for a Map.

Can Contain

[Item element](#)

<SystemProperty>

A SystemProperty element gets the value of a JVM system property. It can be used within elements that accept values, such as Set, Put, Arg.

Attribute	Required	Description
name	yes	property name
default	no	a default value as a fallback
id	no	unique identifier which you can use to refer to the array later.

Can Contain

Cannot contain anything.

Example

```
<SystemProperty name="jetty.port" default="8080"/>
```

That is equivalent to:

```
System.getProperty("jetty.port", "8080");
```

Both try to retrieve the value of jetty.port. If jetty.port is not set, then 8080 is used.

<Property>

A Property element allows arbitrary properties to be retrieved by name. It can contain a sequence of elements, such as Set, Put, Call that act on the retrieved object.

Attribute	Required	Description
name	yes	property name
default	no	a default value as a fallback
id	no	unique identifier which you can use to refer to the array later.

Can Contain

[Set element](#), [Get element](#), [Put element](#), [Call element](#), [New element](#), [Ref element](#), [Array element](#), [Map element](#), [Property element](#)

Example

```
<Property name="Server">
<Call id="jdbcIdMgr" name="getAttribute">
  <Arg>jdbcIdMgr</Arg>
</Call>
</Property>
```

Jetty XML Usage

Jetty provides an XML-based configuration. It is grounded in Java's Reflection API. Classes in the `java.lang.reflect` represent Java methods and classes, such that you can instantiate objects and invoke their methods based on their names and argument types. Behind the scenes, Jetty's XML config parser translates the XML elements and attributes into Reflection calls.

Using `jetty.xml`

To use `jetty.xml`, specify it as a configuration file when running Jetty.

```
java -jar start.jar etc/jetty.xml
```



Note

If you start Jetty without specifying a configuration file, Jetty automatically locates and uses the default installation `jetty.xml` file. Therefore `java -jar start.jar` is equivalent to `java -jar start.jar etc/jetty.xml`.

Using Multiple Configuration Files

You are not limited to one configuration file; you can use multiple configuration files when running Jetty, and Jetty will configure the appropriate server instance. The ID of the server in the `<Configuration>` tag specifies the instance you want to configure. Each server ID in a configuration file creates a new server instance within the same JVM. If you use the same ID across multiple configuration files, those configurations are all applied to the same server.

Setting Parameters in Configuration Files

You can set parameters in configuration files either with system properties (using `<SystemProperty>`) or properties files (using `<Property>`) passed via the command line. For example, this code in `jetty.xml` allows the port to be defined on the command line, falling back onto 8080 if the port is not specified:

```
<Set name="port"><SystemProperty name="jetty.port" default="8080"/></Set>
```

Then you modify the port while running Jetty by using this command:

```
java -Djetty.port=8888 -jar start.jar etc/jetty.xml
```

An example of defining both system properties and properties files from the command line:

```
java -Djetty.port=8888 -jar start.jar myjetty.properties etc/jetty.xml etc/other.xml
```

jetty.xml

`jetty.xml` is the default configuration file for Jetty, typically located at `$JETTY_HOME/etc/jetty.xml`. Usually the `jetty.xml` configures:

- The Server class (or subclass if extended) and global options.
- A ThreadPool (min and max thread).
- Connectors (ports, timeouts, buffer sizes, protocol).
- The handler structure (default handlers and/or a contextHandlerCollections).
- The deployment manager that scans for and deploys webapps and contexts.
- Login services that provide authentication checking.
- A request log.

Not all Jetty features are configured in `jetty.xml`. There are several optional configuration files that share the same format as `jetty.xml` and, if specified, concatenate to it. These configuration files are also stored in `$JETTY_HOME/etc/`, and examples of them are in [SVN Repository](#). The selection of which configuration files to use is controlled by `???` and the process of merging configuration is described in the section called “Jetty XML Usage”.

Root Element

`jetty.xml` configures an instance of the `Jetty org.eclipse.jetty.server.Server`.

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure.dtd">

<Configure id="Server" class="org.eclipse.jetty.server.Server">
...
</Configure>
```

Examples

`$JETTY_HOME/etc` contains the default `jetty.xml`, as well as other sample configuration files (`jetty-*.xml`) which can be passed to the server via the command line.

Additional Resources

- the section called “Jetty XML Syntax” –In-depth reference for Jetty-specific configuration XML syntax.
- the section called “`jetty-web.xml`” –Configuration file for configuring a specific webapp.

jetty-web.xml

`jetty-web.xml` is a Jetty configuration file that you can bundle with a specific web application. The format of `jetty-web.xml` is the same as the section called “`jetty.xml`” – it is an XML mapping of the Jetty API.

This document offers an overview for using the `jetty-web.xml` configuration file. For a more in-depth look at the syntax, see the section called “Jetty XML Syntax”.

Root Element

`jetty-web.xml` applies on a per-webapp basis, and configures an instance of `org.eclipse.jetty.webapp.WebAppContext`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/
configure.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
...
</Configure>
```



Caution

Make sure you are applying the configuration to an instance of the proper class. `jetty-web.xml` configures an instance of `WebAppContext`; `jetty.xml` configures an instance of `Server`.

Using jetty-web.xml

Place the `jetty-web.xml` into your web application's `WEB-INF` folder. When Jetty deploys a web application, it looks for a file called `WEB-INF/jetty-web.xml` or `WEB-INF/web-jetty.xml` within the web application (or WAR) and applies the configuration found there. Be aware that `jetty-web.xml` is called *after* all other configuration has been applied to the web application.

jetty-web.xml Examples

The distribution contains an example of `jetty-web.xml` inside the `WEB-INF` folder of the test webapp war (`$JETTY_HOME/webapps/test.war/WEB-INF/jetty-web.xml`).

Additional jetty-web.xml Resources

- the section called “Jetty XML Syntax” –in-depth reference for Jetty-specific configuration XML syntax.
- the section called “`jetty.xml`” –configuration file for configuring the entire server

jetty-env.xml

`jetty-env.xml` is an optional Jetty file that configures JNDI resources for an individual webapp. The format of `jetty-env.xml` is the same as the section called “`jetty.xml`” –it is an XML mapping of the Jetty API.

When Jetty deploys a web application, it automatically looks for a file called `WEB-INF/jetty-env.xml` within the web application (or WAR), and sets up the webapp naming environment so that naming references in the `WEB-INF/web.xml` file can be resolved from the information provided in the `WEB-INF/jetty-env.xml` and the section called “`jetty.xml`” files. You define global naming resources on the server via `jetty.xml`.

jetty-env.xml Root Element

Jetty applies `jetty-env.xml` on a per-webapp basis, and configures an instance of `org.eclipse.jetty.webapp.WebAppContext`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN" "http://
www.eclipse.org/jetty/configure.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
..
</Configure>
```

Caution



Make sure you are applying the configuration to an instance of the proper class. `jetty-env.xml` configures an instance of `WebAppContext`, and not an instance of `Server`.

Using jetty-env.xml

Place the `jetty-env.xml` file in your web application's `WEB-INF` folder.

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN" "http://
jetty.mortbay.org/configure.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">

    <!-- Add an EnvEntry only valid for this webapp -->
    <New id="gargle" class="org.eclipse.jetty.plus.jndi.EnvEntry">
```

```

<Arg>gargle</Arg>
<Arg type="java.lang.Double">100</Arg>
<Arg type="boolean">true</Arg>
</New>

<!-- Add an override for a global EnvEntry -->
<New id="wiggle" class="org.eclipse.jetty.plus.jndi.EnvEntry">
    <Arg>wiggle</Arg>
    <Arg type="java.lang.Double">55.0</Arg>
    <Arg type="boolean">true</Arg>
</New>

<!-- an XADatasource -->
<New id="mydatasource99" class="org.eclipse.jetty.plus.jndi.Resource">
    <Arg>jdbc/mydatasource99</Arg>
    <Arg>
        <New class="com.atomikos.jdbc.SimpleDataSourceBean">
            <Set name="xaDataSourceClassName">org.apache.derby.jdbc.EmbeddedXADataSource</Set>
            <Set name="xaDataSourceProperties">databaseName=testdb99;createDatabase=create</Set>
            <Set name="UniqueResourceName">mydatasource99</Set>
        </New>
    </Arg>
</New>

</Configure>

```

Additional jetty-env.xml Resources

- the section called “Jetty XML Syntax” –In-depth reference for Jetty-specific configuration XML syntax.
- the section called “jetty.xml” –Configuration file for configuring the entire server.

webdefault.xml

The webdefault.xml file saves web applications from having to define a lot of house-keeping and container-specific elements in their own web.xml files. For example, you can use it to set up mime-type mappings and JSP servlet-mappings. Jetty applies webdefault.xml to a web application *before* the application's own WEB-INF/web.xml, which means that it cannot override values inside the webapp's web.xml. It uses the the section called “jetty.xml” syntax. Generally, it is convenient for all webapps in a Jetty instance to share the same webdefault.xml file. However, it is certainly possible to provide differentiated webdefault.xml files for individual web applications.

The webdefault.xml file is located in \$(jetty.home)/etc/webdefault.xml.

Using webdefault.xml

You can specify a custom configuration file to use for specific webapps, or for all webapps. If you do not specify an alternate defaults descriptor, the \$JETTY-HOME/etc/jetty-deploy.xml file will configure jetty to automatically use \$JETTY_HOME/etc/webdefault.xml.

Creating a Custom webdefault.xml for One WebApp

You can specify a custom webdefault.xml for an individual web application in that webapp's the section called “jetty.xml” as follows:

```

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
    ...

```

```
<!-- Set up the absolute path to the custom webdefault.xml -->
<Set name="defaultsDescriptor">/my/path/to/webdefault.xml</Set>
...
</Configure>
```

The equivalent in code is:

```
import org.eclipse.jetty.webapp.WebAppContext;

...
WebAppContext wac = new WebAppContext();
...
//Set up the absolute path to the custom webdefault.xml.
wac.setDefaultsDescriptor("/my/path/to/webdefault.xml");
...
```

Alternatively, you can use a the section called “Jetty Classloading” to find the resource representing your custom webdefault.xml.

Creating a Custom webdefault.xml for Multiple WebApps

If you want to apply the same custom webdefault.xml to a number of webapps, provide the path to the file in the section called “jetty.xml” in the \$JETTY_HOME/etc/jetty-deploy.xml file:

```
<Set name="defaultsDescriptor"><Property name="jetty.home" default="." />/other/path/
to/another/webdefault.xml</Set>
```

Using the Jetty Maven Plugin

Similarly, when using the [Jetty Maven Plugin](#) you provide a customized webdefault.xml file for your webapp as follows:

```
<project>
...
<plugins>
    <plugin>
        ...
        <artifactId>jetty-maven-plugin</artifactId>
        <configuration>
            <webAppConfig>
                ...
                <defaultsDescriptor>/my/path/to/webdefault.xml</defaultsDescriptor>
            </webAppConfig>
        </configuration>
    </plugin>
    ...
</plugins>
...
</project>
```

Additional Resources

- the section called “jetty-web.xml” –Reference for web.xml files
- the section called “Jetty override-web.xml” –Information for this web.xml -formatted file, applied after the webapp's web.xml webapp.

- the section called “jetty.xml” –Reference for jetty.xml files

Jetty override-web.xml

To deploy a web application or WAR into different environments, most likely you will need to customize the webapp for compatibility with each environment. The challenge is to do so without changing the webapp itself. You can use a jetty.xml file for some of this work since it is not part of the webapp. But there are some changes that jetty.xml cannot accomplish, for example, modifications to servlet init-params and context init-params. Using webdefault.xml is not an option because Jetty applies webdefault.xml to a web application *before* the application's own WEB-INF/web.xml, which means that it cannot override values inside the webapp's web.xml.

The solution is override-web.xml. It is a web.xml file that Jetty applies to a web application *after* the application's own WEB-INF/web.xml, which means that it can override values or add new elements. You define it per-webapp, using the the section called “Jetty XML Syntax”.

Using override-web.xml

You can specify the override-web.xml to use for an individual web application, in that webapp's the section called “jetty-web.xml”.

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
...
<!-- Set up the path to the custom override descriptor,
relative to your ${jetty.home} directory or to the current directory -->
<Set name="overrideDescriptor"><SystemProperty name="jetty.home" default="."/>/my/path/
to/override-web.xml</Set>
...
</Configure>
```

The equivalent in code is:

```
import org.eclipse.jetty.webapp.WebAppContext;

...
WebAppContext wac = new WebAppContext();
...
//Set the path to the override descriptor, based on your ${jetty.home} directory
wac.setOverrideDescriptor(System.getProperty("jetty.home")+"/my/path/to/override-
web.xml");
...
```

Alternatively, use the classloader (the section called “Jetty Classloading”) to get the path to the override descriptor as a resource.

Using the Jetty Maven Plugin

Use the <overrideDescriptor> tag as follows:

```
<project>
...
<plugins>
```

```
<plugin>
  ...
  <artifactId>jetty-maven-plugin</artifactId>
  <configuration>
    <webAppConfig>
      ...
      <overrideDescriptor>src/main/resources/override-web.xml</
overrideDescriptor>
      </webAppConfig>
    </configuration>
  </plugin>
  ...
</plugins>
...
</project>
```

Additional Resources

- the section called “`webdefault.xml`” –Information for this `web.xml` -formatted file, applied before the webapp's `web.xml` webapp.
- the section called “`jetty.xml`” –Reference for `jetty.xml` files

Chapter 37. Troubleshooting

Table of Contents

Troubleshooting Zip Exceptions	450
Troubleshooting Locked Files on Windows	450
Preventing Memory Leaks	452
Jetty Security Reports	455

This is a collection of helpful tricks and tips that we have come across to address odd issues that might arise.

Troubleshooting Zip Exceptions

A Zip exception occurs when Jetty rereads a Jar or WAR file.

The JVM maintains a cache of zip file indexes, and does not support hot replacement of zip files. Thus if you redeploy a web application using the same WAR or Jar files, exceptions occur when Jetty rereads the jars. See [Oracle Bug 4774421](#) for more information.

Remedy

The remedy is to avoid hot replacing Jar or WAR files, which can be difficult if you are using the [Webapp Provider](#). You can use the following techniques to reduce exposure to this issue:

- Deploy unpacked classes in the WEB-INF/classes directory rather than as a Jar file under WEB-INF/lib.
- Deploy all WAR and Jar files with a version number in their filename or path. If the code changes, a new version number applies, avoiding the cache problem.
- Deploy a packed WAR file with the [setExtractWAR](#) option set to true. This causes the WAR to be extracted to a [temporary directory](#) and thus to a new location. This might not be sufficient if you want to hot-replace and re-extract the WAR, so you might also need to use [WebApplicationContext.setCopyWebInf\(true\)](#), which (re)copies just the WEB-INF directory to a different location.
- Deploy an unpacked WAR file with the [setCopyWebDir](#) option set to true. This causes the directory to be extracted to a new location.

If you have problems with [Windows file-locking](#) preventing static file editing (such as JSP or HTML), use the [WebApplicationContext.setCopyWebDir\(true\)](#) option.

Troubleshooting Locked Files on Windows

Jetty buffers static content for webapps such as HTML files, CSS files, images, etc. If you are using NIO connectors, Jetty uses memory-mapped files to do this. The problem is that on Windows, memory mapping a file causes the file to lock, so that you cannot update or replace the file. Effectively this means that you have to stop Jetty to update a file.

Remedy

Jetty provides a configuration switch in the `webdefault.xml` file for the `DefaultServlet` that enables or disables the use of memory-mapped files. If you are running on Windows and are having file-locking problems, you should set this switch to disable memory-mapped file buffers.

The default `webdefault.xml` file is found in the jetty distribution under the `etc/` directory or in the `jetty-webapp-$\{version}\.jar` artifact at `org/eclipse/jetty/webapp/webdefault.xml`. Edit the file in the distribution or extract it to a convenient disk location and edit it to change `useFileMappedBuffer` to false. The easiest option is to simply edit the default file contained in the jetty distribution itself.

```
<init-param>
  <param-name>useFileMappedBuffer</param-name>
  <param-value>true</param-value> <!-- change to false -->
</init-param>
```

Make sure to apply your custom `webdefault.xml` file to all of your webapps. You can do that by changing the configuration of the Deployment Manager in `etc/jetty-deploy.xml`.

```
<Call id="webappprovider" name="addAppProvider">
  <Arg>
    <New class="org.eclipse.jetty.deploy.providers.WebAppProvider">
      .
      .
      .
      <!-- this should be the new custom webdefault.xml or change should be made in this
file -->
      <Set name="defaultsDescriptor"><Property name="jetty.home" default="." />/etc/
webdefault.xml</Set>
      <Set name="scanInterval">1</Set>
      <Set name="extractWars">true</Set>
      .
      .
      .
    </New>
  </Arg>
</Call>
```

Alternatively, if you have individually configured your webapps with context xml files, you need to call the `WebAppContext.setDefaultsDescriptor(String path)` method:

```
<New id="myWebAppContext" class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath"></Set>
  <Set name="war">./webapps/fredapp</Set>
  <Set name="defaultsDescriptor">/home/fred/jetty/mywebdefaults.xml</Set>
  .
  .
  .
</New>
```

Instead, you could redefine the `DefaultServlet` in your `web.xml` file, making sure to set `useFileMappedBuffer` to false:

```
<web-app ...>
  ...
  <servlet>
    <servlet-name>Default</servlet-name>
    <servlet-class>org.eclipse.jetty.servlet.DefaultServlet</servlet-class>
    <init-param>
      <param-name>useFileMappedBuffer</param-name>
      <param-value>false</param-value>
    </init-param>
    <load-on-startup>0</load-on-startup>
  </servlet>
  ...
</web-app>
```

Alternate Remedy

You can force a WebAppContext to always copy a web app directory on deployment. The base directory of your web app (ie the root directory where your static content exists) will be copied to the [temp directory](#). Configure this in an xml file like so:

```
<New id="myWebAppContext" class="org.eclipse.jetty.webapp.WebAppContext">
<Set name="contextPath"/></Set>
<Set name="war">./webapps/fredapp</Set>
<Set name="copyWebDir">true</Set>
.
.
</New>
```

Note



Be careful with this option when using an explicitly set [temp directory](#) name - as the name of the temp directory will not unique across redeployments, copying the static content into the same directory name each time may not avoid the locking problem.

Preventing Memory Leaks

If you have memory leaks, and you have thoroughly investigated tools like jconsole, yourkit, jprofiler, jvisualvm or any of the other profiling and analysis tools, and you can eliminate your code as the source of the problem, read the following sections about how to prevent memory leaks in your application.

Preventing WebApp Classloader Pinning

Note



This feature is available for Jetty 7.6.6 and later.

Code that keeps references to a webapp classloader can cause memory leaks. These leaks fall generally into two categories: static fields and daemon threads.

- A static field is initialized with the value of the classloader, which happens to be a webapp classloader; as Jetty undeploys and redeploys the webapp, the static reference lives on, meaning garbage collecting cannot occur for the webapp classloader.
- When Jetty starts as a daemon thread and is outside the lifecycle of the webapp, threads have references to the context classloader that created them, leading to a memory leak if that classloader belongs to a webapp. For a good discussion of the issue see [Anatomy of a PermGen Memory Leak](#).

We provide a number of [workaround classes](#) that preemptively invoke the problematic code with the Jetty classloader, thereby ensuring the webapp classloader is not pinned. Be aware that since some of the problematic code creates threads, you should be selective about which preventers you enable, and use only those that are specific to your application.

Preventers

Jetty includes the following preventers.

Preventer Name	Problem Addressed
AppContextLeakPreventer	The call to <code>AppContext.getAppContext()</code> keeps a static reference to the context classloader. The

Preventer Name	Problem Addressed
	JRE can invoke ApplicationContext in many different places.
AWTLeakPreventer	The <code>java.awt.Toolkit</code> class has a static field that is the default toolkit. Creating the default toolkit causes the creation of an <code>EventQueue</code> , which has a <code>ClassLoader</code> field initialized with the thread context class loader. See JBoss bug AS7-3733 .
DOMLeakPreventer	DOM parsing can cause the webapp classloader to be pinned, due to the static field <code>RuntimeException</code> of <code>com.sun.org.apache.xerces.internal.parsers.Ab</code> . Oracle bug 6916498 specifically mentions that a heap dump might not identify the GCRoot as the uncollected loader, making it difficult to identify the cause of the leak.
DriverManagerLeakPreventer	The number of threads dedicated to accepting incoming connections.
GCThreadLeakPreventer	Calls to <code>sun.misc.GC.requestLatency</code> create a daemon thread that keeps a reference to the context classloader. A known caller of this method is the RMI impl. See Stackoverflow: Does java garbage collection log entry 'Full GC system' mean some class called System.gc()?
Java2DLeakPreventer	<code>sun.java2d.Disposer</code> keeps a reference to the classloader. See ASF bug 51687 .
LDAPLeakPreventer	If <code>com.sun.jndi.LdapPoolManager</code> class is loaded and the system property <code>com.sun.jndi.ldap.connect.pool.timeout</code> is set to a nonzero value, a daemon thread starts and keeps a reference to the context classloader.
LoginConfigurationLeakPreventer	The <code>javax.security.auth.login.Configuration</code> class keeps a static reference to the thread context classloader.
SecurityProviderLeakPreventer	Some security providers, such as <code>sun.security.pkcs11.SunPKCS11</code> start a deamon thread that traps the thread context classloader.

Configuring Preventers

You can individually enable each preventer by adding an instance to a Server with the `addBean(Object)` call. Here's an example of how to do it in code with the `org.eclipse.jetty.util.preventers.AppContextLeakPreventer`:

```
Server server = new Server();
server.addBean(new AppContextLeakPreventer());
```

You can add the equivalent in code to the `$JETTY_HOME/etc/jetty.xml` file or any jetty xml file that is configuring a Server instance. Be aware that if you have more than one Server instance

in your JVM, you should configure these preventers on just *one* of them. Here's the example from code put into xml:

```
<Configure id="Server" class="org.eclipse.jetty.server.Server">

    <Call name="addBean">
        <Arg>
            <New class="org.eclipse.util.preventers.AppContextLeakPreventer"/>
        </Arg>
    </Call>

</Configure>
```

JSP Bugs: Permgen Problems

The JSP engine in Jetty is Jasper. This was originally developed under the Apache Tomcat project, but over time many different projects have forked it. All Jetty versions up to 6 used Apache-based Jasper exclusively, with Jetty 6 using Apache Jasper only for JSP 2.0. With the advent of JSP 2.1, Jetty 6 switched to using Jasper from Sun's [Glassfish](#) project, which is now the reference implementation.

All forks of Jasper suffer from a problem whereby using JSP tag files puts the permgen space under pressure. This is because of the classloading architecture of the JSP implementation. Each JSP file is effectively compiled and its class loaded in its own classloader to allow for hot replacement. Each JSP that contains references to a tag file compiles the tag if necessary and then loads it using its own classloader. If you have many JSPs that refer to the same tag file, the tag's class is loaded over and over again into permgen space, once for each JSP. See [Glassfish bug 3963](#) and [Apache bug 43878](#). The Apache Tomcat project has already closed this bug with status WON'T FIX, however the Glassfish folks still have the bug open and have scheduled it to be fixed. When the fix becomes available, the Jetty project will pick it up and incorporate into our release program.

JVM Bugs

This section describes garbage collection and direct ByteBuffer problems.

Garbage Collection Problems

One symptom of a cluster of JVM related memory issues is the OOM exception accompanied by a message such as `java.lang.OutOfMemoryError: requested xxxx bytes for xxx. Out of swap space?`

[Oracle bug 4697804](#) describes how this can happen in the scenario when the garbage collector needs to allocate a bit more space during its run and tries to resize the heap, but fails because the machine is out of swap space. One suggested work around is to ensure that the JVM never tries to resize the heap, by setting min heap size to max heap size:

```
java -Xmx1024m -Xms1024m
```

Another workaround is to ensure you have configured sufficient swap space on your device to accommodate all programs you are running concurrently.

Direct ByteBuffers

Exhausting native memory is another issue related to JVM bugs. The symptoms to look out for are the process size growing, but heap use remaining relatively constant. Both the JIT compiler and nio

ByteBuffers can consume native memory. [Oracle bug 6210541](#) discusses a still-unsolved problem whereby the JVM itself allocates a direct ByteBuffer in some circumstances while the system never garbage collects, effectively eating native memory. Guy Korland's blog discusses this problem [here](#) and [here](#). As the JIT compiler consumes native memory, the lack of available memory may manifest itself in the JIT as OutOfMemory exceptions such as Exception in thread "CompilerThread0" java.lang.OutOfMemoryError: requested xxx bytes for ChunkPool::allocate. Out of swap space?

By default, Jetty allocates and manages its own pool of direct ByteBuffers for io if you configure thenio SelectChannelConnector. It also allocates MappedByteBuffers to memory-map static files via theDefaultServlet settings. However, you could be vulnerable to this JVM ByteBuffer allocation problem if you have disabled either of these options. For example, if you're on Windows, you may have disabled the use of memory-mapped buffers for the static file cache on the DefaultServlet to avoid the file-locking problem.

Jetty Security Reports

The following sections provide information about Jetty security issues.

Table 37.1. Resolved Issues

yyyy/mm/dd	ID	Exploitable	Severity	Affects	Fixed Version	Comment
2013/11/27	PT-2013-65	medium	high	>=9.0.0 <9.0.5	9.0.6 418014	Alias checking disabled by NTFS errors on Windows.
2013/07/24	413684	low	medium	>=7.6.9 <9.0.5	7.6.13,8.1.13,9.0.0 413684	Constraints bypassed if unix symlink alias checker used on windows
2011/12/29	CERT2011-003 CVE-2011-4461	high	medium	All versions	7.6.0.RC0 Jetty-367638	Added ContextHandler.setMaxFormKeys(intkeys) to limit the number of parameters (default 1000).
2009/11/05	CERT2011-003 CVE-2011-003	high	JVM<1.6u19	jetty-7.0.1.v20091105 jetty-6.1.22	09bf25 around by turning off SSL renegotiation in Jetty. If using JVM > 1.6u19 setAllowRenegotiate(true) may be called on connectors.	
2009/06/18	Jetty-1042	low	high	<=6.1.18, <=7.0.0.M4	6.1.19, 7.0.0.Rc0	Cookie leak between requests sharing a connection.
2009/04/30	CERT402580	medium	high	<=6.1.16, <=7.0.0.M2	5.1.15, 6.1.18, 7.0.0.M2 Jetty-1004	View arbitrary disk content in some specific configurations.

yyyy/mm/dd	ID	Exploitable	Severity	Affects	Fixed Version	Comment
2007/12/	CERT553235 CVE-2007-6672	high	medium	6.1.rc0-6.1.6	6.1.7 CERT553235	Static content visible in WEB-INF and past security constraints.
2007/11/	CERT438616 CVE-2007-5614	low		<6.1.6	6.1.6rc1 (patch in CVS for jetty5)	Single quote in cookie name.
2007/11/	CERT237888 >CVE-2007-5613	low		<6.1.6	6.1.6rc0 (patch in CVS for jetty5)	XSS in demo dup servlet.
2007/11/	CERT212984 >CVE-2007-5615	medium	medium	<6.1.6	6.1.6rc0 (patch in CVS for jetty5)	CRLF Response splitting.
2006/11/	CVE-2006-6969	low	high	<6.1.0, <6.0.2, <5.1.12, <4.2.27	6.1.0pre3, 6.0.2, 5.1.12, 4.2.27	Session ID predictability.
2006/06/	CVE-2006-2759	medium	medium	<6.0.*, <6.0.0Beta17	6.0.0Beta17	JSP source visibility.
2006/01/05		medium	medium	<5.1.10	5.1.10	Fixed //security constraint bypass on Windows.
2005/11/	CVE-2006-2758	medium	medium	<5.1.6	5.1.6, 6.0.0Beta4	JSP source visibility.
2004/02/	JSSE 1.0.3_01	medium	medium	<4.2.7	4.2.7	Upgraded JSSE to obtain downstream security fix.
2002/09/22		high	high	<4.1.0	4.1.0	Fixed CGI servlet remove exploit.
2002/03/12		medium		<3.1.7	4.0.RC2, 3.1.7	Fixed // security constraint bypass.
2001/10/21	medium		high	<3.1.3	3.1.3	Fixed trailing null security constraint bypass.