JETTY SERVER COOKBOOK

Hot Recipes for the Jetty Server

IBRAHIM TASYURT

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Jetty Server Cookbook

Contents

Preface

Jetty is a Java HTTP (Web) server and Java Servlet container. While Web Servers are usually associated with serving documents to people, Jetty is now often used for machine to machine communications, usually within larger software frameworks. Jetty is developed as a free and open source project as part of the Eclipse Foundation.

The web server is used in products such as Apache ActiveMQ, Alfresco, Apache Geronimo, Apache Maven, Apache Spark, Google App Engine, Eclipse, FUSE, iDempiere, Twitter's Streaming API and Zimbra. Jetty is also the server in open source projects such as Lift, Eucalyptus, Red5, Hadoop and I2P. Jetty supports the latest Java Servlet API (with JSP support) as well as protocols HTTP/2 and WebSocket. (Source: https://en.wikipedia.org/wiki/JavaFX)

In this ebook, we provide a compilation of Jetty examples that will help you kick-start your own projects. We cover a wide range of topics, from installation and configuration, to JMX and OSGi. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

Ibrahim is a Senior Software Engineer residing in Ankara,Turkey. He holds BSc and MS degrees in Computer Engineering from Middle East Technical University(METU). Throughout his professional carrier, he has worked in Enterprise Web Application projects for public sector and telecommunications domains. Java EE, Web Services and Enterprise Application Integration are the areas he is primarily involved with.

Chapter 1

Jetty Tutorial for Beginners

In this article, we will give brief information about Jetty and provide examples of Java application deployment on Jetty. Our examples will consist of both standalone and Embedded modes of Jetty.

Jetty is a Servlet container and Web Server which is known to be portable, lightweight, robust, flexible, extensible and easy to integrate.

Jetty can be deployed as a standalone server and also can be embedded in an existing application. In addition to these, a Maven Jetty plugin is available in order to run applications in your development environment.

SPDY, WebSocket, OSGi, JMX, JNDI, JAAS are some of the technologies that Jetty integrates nicely.

Today, Jetty is widely used in many platforms both for development and production. Small to large enterprise applications. SaaS (such as Zimbra), Cloud Applications(such as Google AppEngine), Applications Servers(such as Apache Geronimo) and tools (such as SoapUI) are powered by Jetty.

Jetty is open source, hosted by Eclipse Foundation. Current version (as of June 2015) is 9.2.x. You can more detailed information on Jetty Home Page.

1.1 Jetty as a Standalone server

In the first part, we will configure Jetty as a Standalone Server.

1.1.1 Downloading and Installing Jetty

You can visit the downloads page and download the latest version (v9.2.11 currently) as an archive file in zip or tar.gz format. Size is about 13 MBs.

There is no installation procedure for Jetty. Just drop it to a folder as you wish and uncompress the downloaded archive file.

1.1.2 Prerequisites

The only prerequisite for Jetty 9 is having installed Java 7 in your environment. You can downgrade to Jetty 8 if you have Java 6. A complete Jetty-Java compatibility information can be viewed here.

1.1.3 Running Jetty

Running Jetty on the default configuration is as simple as following two steps:

- Navigate to the directory where you unpacked the downloaded archive. I will call it `JETTY_HOME`from now on.

- Run the following command:

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

When Jetty starts running successfully; it produces the a line in the log similar to the following:

```
2015-06-04 14:27:27.555:INFO:oejs.Server:main: Started @11245ms
```

By default, Jetty runs on port 8080, but we will see how to configure it in the next sections of this tutorial.

You can also check via the browser typing https://localhost:8080 as the URL. You will see a 404 error, since no application is deployed in the root context.

The response is as below:

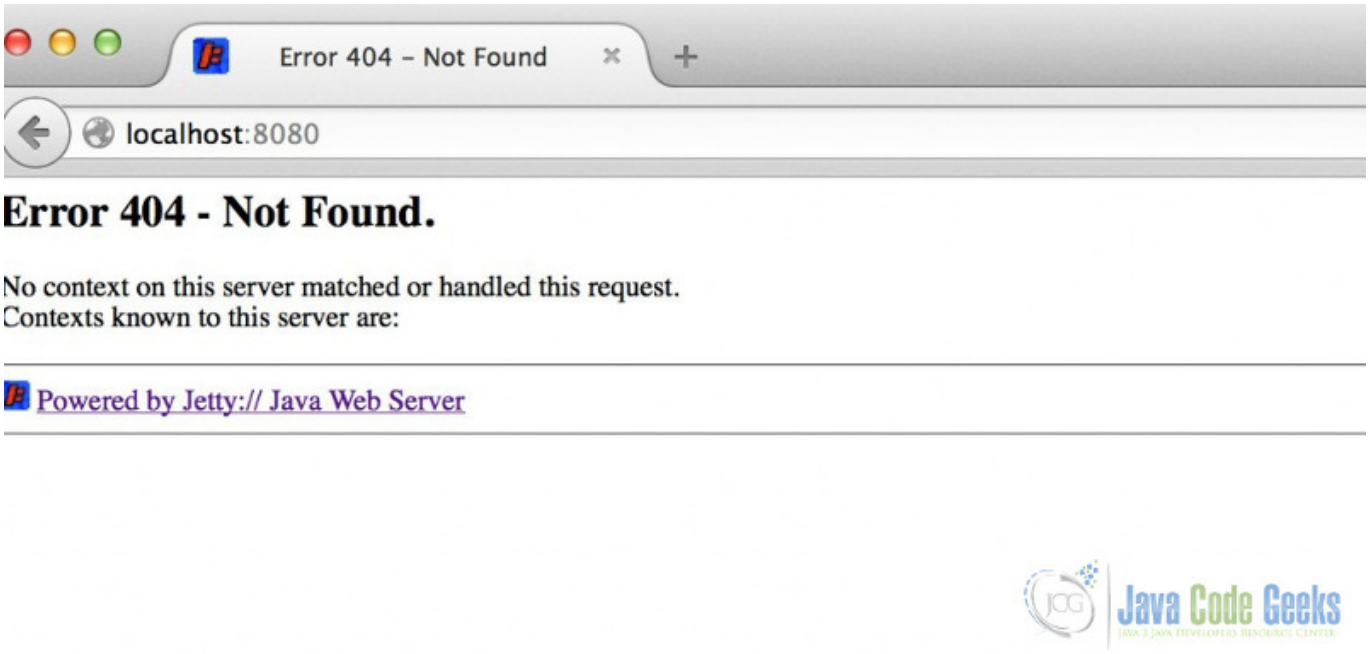

Figure 1.1: Server response when Jetty runs successfully

1.1.4 Changing the server port

As mentioned above, default port jetty is 8080. If you need to change it, you can apply following steps:

- Navigate to the ***JETTY_HOME***.
- Open the start.ini file with a text editor.
- Navigate to the line where the parameter jetty.port is configured.
- Change the parameter to the desired port number.
- Start Jetty again.

In the following segment, we set the Jetty port to 7070 instead of 8080

```
## HTTP port to listen on
jetty.port=7070
```

After we restart our server will run on port 7070.

1.1.5 Deploying Web Applications on Jetty

The procedure to deploy web applications on Jetty is as follows:

- Navigate to your *JETTY_HOME* folder.
- There is a directory named as webapps under JETTY_HOME. Navigate there.
- Drop your WAR file in that folder.

The application is initialized immediately, you do not need to restart Jetty since the `webapps` directory is continuously monitored by the server.

There are a sample web applications under `JETTY_HOME/demo-base/webapps/`. You can pick one of them (for example `async-rest.war`) and copy to the webapps directory. As you copy the WAR file, the application will be initialized.

When you type https://localhost:7070/async-rest , you can see the application initialized.

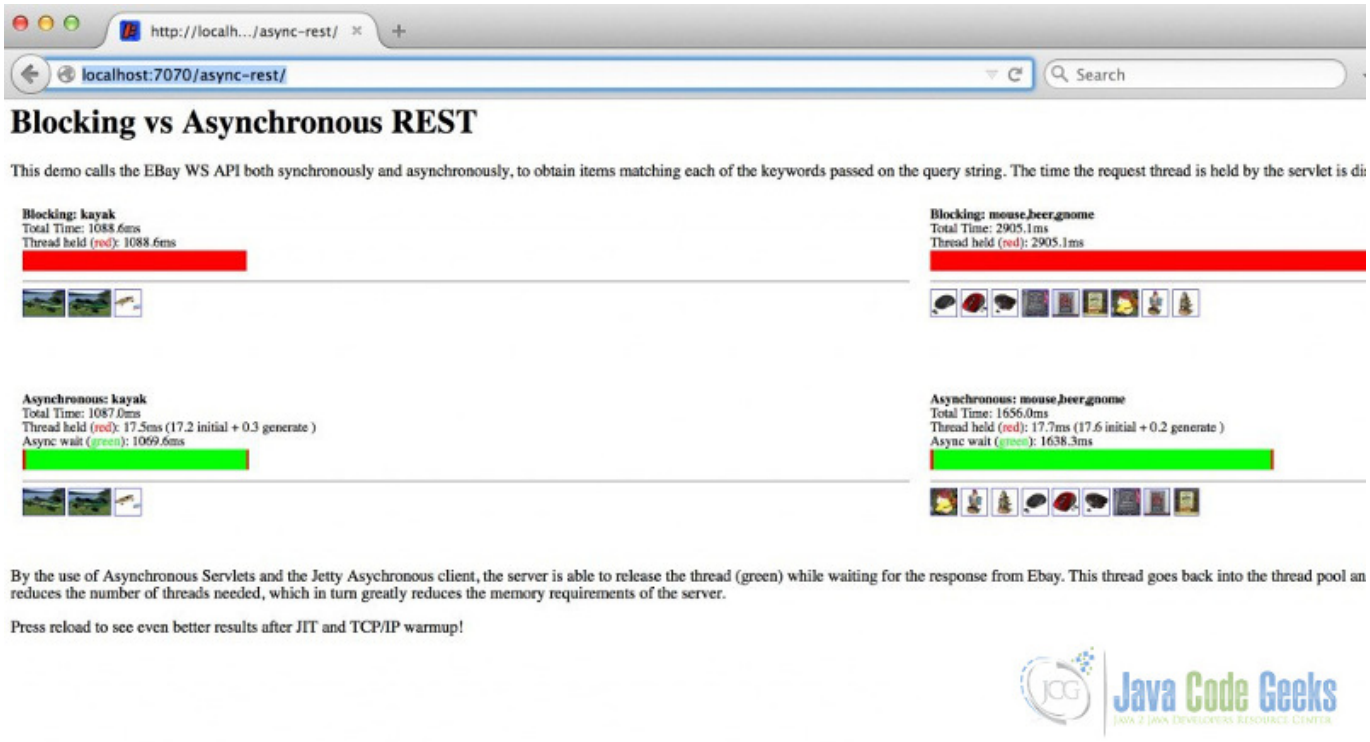

Figure 1.2: The application async-rest deployed on Jetty

1.1.6 Changing Webapps Directory

`JETTY_HOME/webapps` is the default directory to deploy your applications. But there are cases that you need to change the deployment directory. In order to do that, you should proceed as follows:

- Open the `start.ini` file under `JETTY_HOME`.
- Remove the comment before the parameter *jetty.deploy.monitoredDirName.*
- Change this parameter as you wish. Remember that the path should be relative to `JETTY_HOME` directory.

```
jetty.deploy.monitoredDirName=../jcgwebapps
```

Now we can put our WARS in the *jcgwebapps* directory, which is at the same level as our **JETTY_HOME.**

1.2 Embedding Jetty in Your Application

Until now, we have skimmed through Jetty as a standalone server. However Jetty provides another great feature. Motto of Jetty is : "Don't deploy your application in Jetty, deploy Jetty in your application". It means that, you can embed jetty in your existing (most probably non-web) applications easily. On this purpose a very convenient API is provided to the developers. In the following sections, we will see how we can accomplish this.

1.2.1 Environment

In this example, following programming environment is used:

- Java 8 (Java 7 will also do fine.)
- Apache Maven 3.x.y
- Eclipse 4.4 (Luna)

1.2.2 Creating the Maven Project

- Go to File → New →Other → Maven Project
- Tick Create a simple project and press "Next".
- Enter groupId as : com.javacodegeeks.snippets.enterprise
- Enter artifactId as : embedded-jetty-example
- Press "Finish".

Now our maven project is created.

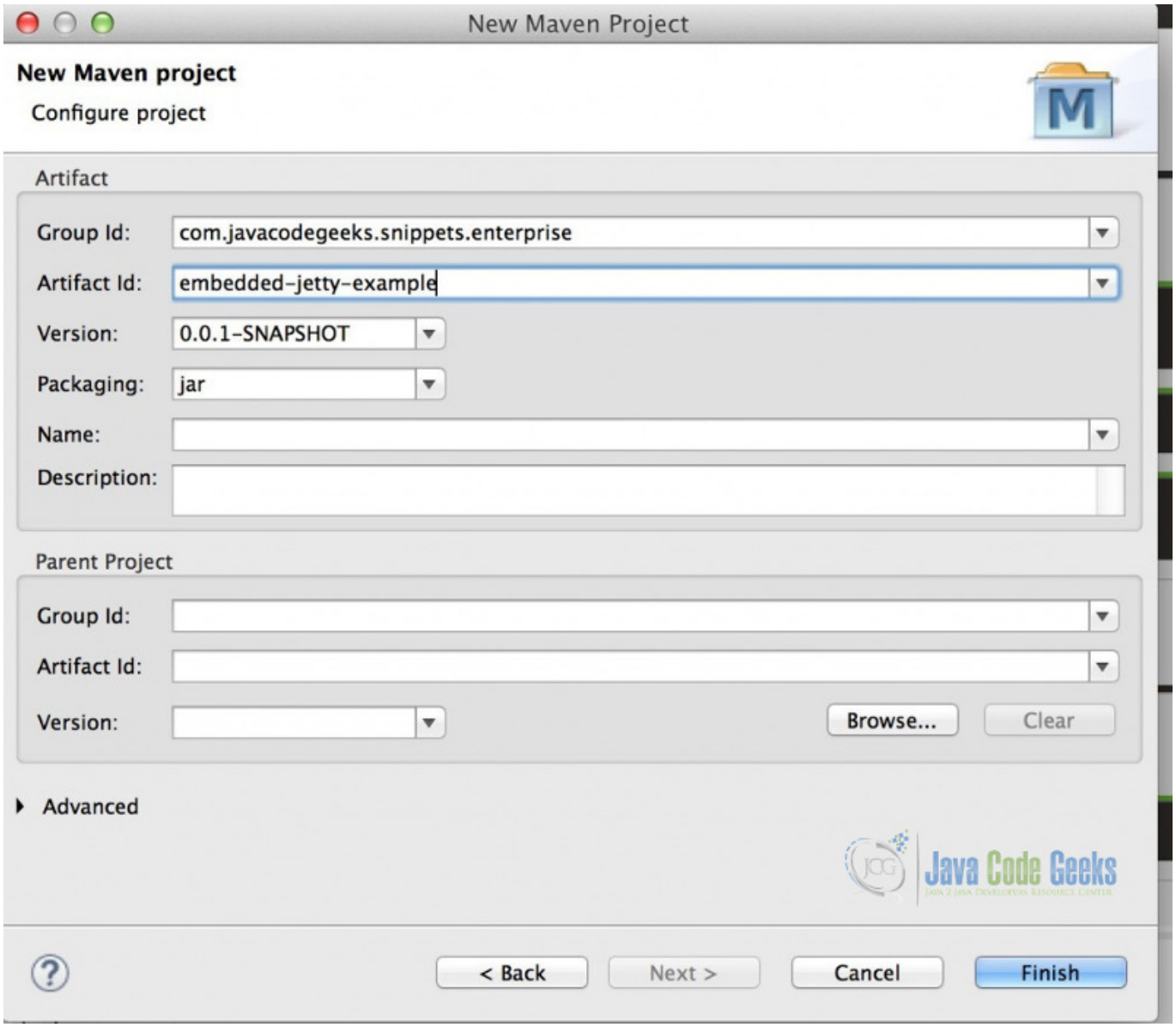

Figure 1.3: Creating simple Maven project in Eclipse

1.2.3 Adding dependencies for Embedded Jetty

Following Maven dependencies have to be added in the project:

- jetty-server : Core Jetty Utilities
- jetty-servlet: Jetty Servlet Utilities

You have to add these dependencies to your pom.xml. After the dependencies are added, your pom.xml seems as follows:

```
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.javacodegeeks.snippets.enterprise</groupId>
        <artifactId>embedded-jetty-example</artifactId>
        <version>0.0.1-SNAPSHOT</version>

        <dependencies>
```

```
                <!--Jetty  dependencies start here-->
                <dependency>
                        <groupId>org.eclipse.jetty</groupId>
                        <artifactId>jetty-server</artifactId>
                        <version>9.2.11.v20150529</version>
                </dependency>

                <dependency>
                        <groupId>org.eclipse.jetty</groupId>
                        <artifactId>jetty-servlet</artifactId>
                        <version>9.2.11.v20150529</version>
                </dependency>
                <!--Jetty  dependencies end here-->

        </dependencies>
</project>
```

Now our project configuration is complete and we are ready to go.

1.2.4 Creating Embedded Jetty Server Programmatically

Now we are going to create an Embedded Jetty Server programmatically. In order to keep things simple, we will create the Server in the main() method of our application.

In order to this, you can proceed as follows:

- Create package com.javacodegeeks.snippets.enterprise.embeddedjetty.
- Create a class named EmbeddedJettyMain.
- Add a main method to this class.

The code that creates and starts and Embedded Jetty is as follows:

EmbeddedJettyMain.java

```
package com.javacodegeeks.snippets.enterprise.embeddedjetty;

import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.servlet.ServletContextHandler;

import com.javacodegeeks.snippets.enterprise.embeddedjetty.servlet.ExampleServlet;

public class EmbeddedJettyMain {

        public static void main(String[] args) throws Exception {

                Server server = new Server(7070);
                ServletContextHandler handler = new ServletContextHandler(server, "/example ↩
                    ");
                handler.addServlet(ExampleServlet.class, "/");
                server.start();

        }

}
```

- In the first line(Line 12), we create a Server on port **7070**.

- In the next line(Line 13), we create a ServletContextHandler with the context path* /example*
- In Line 14, we bind the servlet class ExampleServlet (which is described below) to this servlet context handler created in the previous line.
- On the last line, we start the server.

ServletContextHandler is a powerful facility enabling creation and configuration of Servlets and Servlet Filters programatically.

ExampleServlet is a simple HttpServlet, does nothing but returning a constant output EmbeddedJetty as the response.

ExampleServlet.java

```
package com.javacodegeeks.snippets.enterprise.embeddedjetty.servlet;

import java.io.IOException;

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

import org.eclipse.jetty.http.HttpStatus;

public class ExampleServlet extends HttpServlet {

        @Override
        protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                        throws ServletException, IOException {

                resp.setStatus(HttpStatus.OK_200);
                resp.getWriter().println("EmbeddedJetty");
        }
}
```

1.2.5 Running Embedded Jetty

Run the EmbeddedJettyMain class through the Eclipse Run, Embedded Jetty starts to run on the defined port(7070).

You can access the application through your browser on the following URL: https://localhost:7070/example

Here you can see the response below:

Figure 1.4: ExampleServlet response

1.3 Conclusion

In this article, we have provided brief information on Jetty and discussed the steps to create standalone and Embedded Jetty servers.

Download

You can download the source code for Embedded Jetty example here: **EmbeddedJettyExample**

Chapter 2

How to Install Jetty Application Server

Jetty is an open-source Servlet container and Application Server which is known to be lightweight, portable, robust, flexible, extensible and providing support for various technologies like SPDY, WebSocket, OSGi, JMX, JNDI, and JAAS. Jetty is very convenient for development and also widely used in production environments.

In this post, we are going to detail how to install and configure a Jetty Server. We are first going to describe how to setup and run a standalone Jetty. Thereafter we will mention some configuration options and skim through the modular architecture of Jetty.

Jetty presents Standalone, Embedded and Jetty Maven Plugin modes of operation. In this post we are going to use standalone Jetty.

2.1 Environment

In this post, we are going to use following environment:

- Java 8
- Jetty 9.3.2.v20150730

However, it should be noted that; the material presented in this post is applicable for any Java versions later than Java 5, and any Jetty installations of version 9.x.y. Jetty does not require any 3rd party libraries except having Java installed in your PATH.

2.2 Downloading Jetty

Jetty binaries can be downloaded from the Jetty Homepage. Binaries are available in *zip* and *tgz* formats. Jetty is fully cross-platform so same binaries are valid for both Java and Unix environments.

2.3 Running Jetty

After downloading the binaries, having your Jetty server up and running is really easy. First you have to extract the *zip* (or *tgz*) archive to a convenient directory. After extracting the binaries, you have to navigate to the directory (_jetty-distribution-9.3.2.v20150730 _in this example). We will call it _JETTY_HOME _from now on in this post.

In JETTY_HOME you have to run the following shell command in order to start Server:

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

This command yields to following output:

```
2015-08-30 20:57:07.486:INFO::main: Logging initialized @361ms
2015-08-30 20:57:07.541:WARN:oejs.HomeBaseWarning:main: This instance of Jetty is not  ↩
    running from a separate {jetty.base} directory, this is not recommended. See  ↩
    documentation at http:www.eclipse.orgjettydocumentationcurrentstartup.html
2015-08-30 20:57:07.688:INFO:oejs.Server:main: jetty-9.3.2.v20150730
2015-08-30 20:57:07.705:INFO:oejdp.ScanningAppProvider:main: Deployment monitor [file: ↩
    Usersibrahimjcgexamplesjettyjetty-distribution-9.3.2.v20150730webapps] at interval 1
2015-08-30 20:57:07.729:INFO:oejs.ServerConnector:main: Started ServerConnector@7a1d7e18{ ↩
    HTTP1.1,[http1.1]}{0.0.0.0:8080}
2015-08-30 20:57:07.730:INFO:oejs.Server:main: Started @606ms
```

Seeing the last line like `2015-08-30 20:57:07.730:INFO:oejs.Server:main:Started @606ms` means that: you have successfully started Jetty. In order to verify successful start you can navigate to https://localhost:8080/ and see the following response:

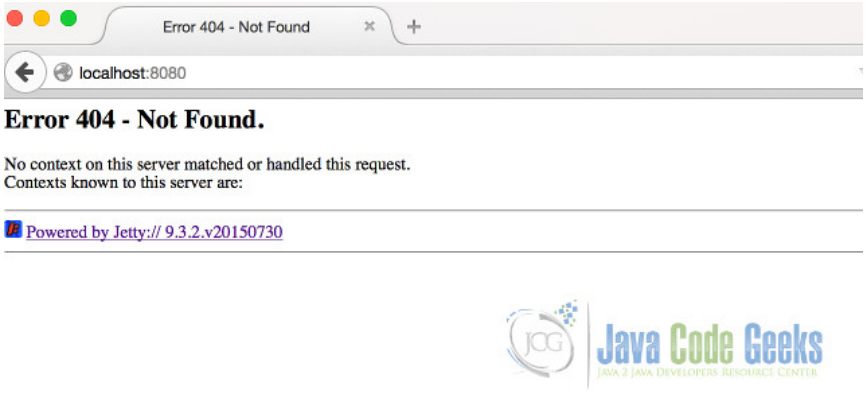

Figure 2.1: Jetty Server Response

2.4 Running Web Applications In Jetty

It is very to install your web applications (WARs) in Jetty. All you have to do is dropping the WAR file under `JETTY_HOME/webapps`. You do not even need to restart Jetty. `webapps` directory is monitored periodically and new applications are deployed automatically.

2.5 Anatomy of the JETTY_HOME Directory

When you examine the content of the JETTY_HOME you will see following directories:

- bin
- demo-base
- etc
- lib
- logs
- modules
- resources
- start.d

- webapps

Some of these directories are worth mentioning:

- etc: The XML configuration of Jetty modules defined in this directory.
- lib: As in most *Java* applications, *JAR* files are stored in *lib* directory.
- logs: When logging is enabled, log files are created under this directory.
- modules: Jetty modules are defined under *modules* directory
- resources: The external configuration resources (like logging configuration resources) are usually placed under this directory.
- start.d: The modules activated through command line are configured through this directory.
- webapps: The web applications (WAR files) running in our Jetty Server are dropped in this directory.

In *JETTY_HOME* directory, `start.ini` and `start.jar` files exist `start.ini` is the configuration file where most Jetty configuration is performed. `start.jar` is the initial executable file that initiates startup of the server.

2.6 Basic Configuration

2.6.1 Changing the Jetty Port

By default, Jetty runs on 8080. In order to change it to 7070 or some other port, you have to do the following:

- Open `start.ini` under *JETTY_HOME*.
- Add this line `jetty.port=7070`
- Save and close the file.

When you start the Jetty again it will run on port 7070.

2.6.2 Changing the webapps Directory

`JETTY_HOME/webapps` is the default directory to deploy your applications. If you need to change it for some reason, the steps to be followed are as follows:

- Open the `start.ini`.
- Remove the comment before the parameter *jetty.deploy.monitoredDirName*
- Change this parameter as you wish(E.g: `jetty.deploy.monitoredDirName=../webapps2`)
- Save and close the file.

Now we can put our WARs in the `webapps2` directory, which is at the same level as our *JETTY_HOME*

2.7 Modular Architecture of Jetty

Jetty runs on a modular architecture which means that many facilities and integrations are presented as modules. HTTP, HTTPS, SSL, logging, JMX, JNDI, WebSockets and many other features are implemented as separate modules. Some common modules such as HTTP, JSP and WebSocket are activated by default. The others (such as HTTPS, JMX etc.) have to be activated manually.

2.7.1 Anatomy of a Single Module

The modules are listed under `JETTY_HOME/modules` directory as `mod` files. `mod` files state the required JAR files to be activated (which are under `JETTY_HOME/lib` directory) and XML configuration files (which are under `JETTY_HOME/etc` directory) and other resources to be activated as the module is activated.

For example, when you view `JETTY_HOME/modules/logging.mod` content of, you will see something like the following:

xml etc/jetty-logging.xml

files logs/

lib lib/logging/**.jar resources/

The configuration states that logging is configured through `etc/jetty-logging.xml`; and required JARs are under `lib/logging`. In addition to these, `logs` directory is required for this module.

2.7.2 Activating Modules through Command Line

There are two ways to activate Jetty modules. The first way is activating through command line:

```
java -jar start.jar --add-to-startd=logging
```

The command above creates the file `logging.ini` under *JETTY_HOME*. Related configuration can be found in this file. After configuring logging, you can start Jetty again and observe that logging is active.

2.7.3 Activating Modules through start.ini

The second way to activate a module is adding the module to the `start.ini`:

```
--module=logging
```

This is equivalent to the command line presented in the first alternative with a subtle difference. This time, nothing is created under `start.d` ; so all further configuration should be done in this same `start.ini` file.

Personally, I would prefer the second alternative since all active modules are listed in a single file (`start.ini`) however there is no problem with the first approach either.

2.7.4 Configuring the Modules

As mentioned above, `mod` files tell us about the relevant XML configuration files, which are under `JETTY_HOME/etc`, for the module. Jetty modules are configured through these XML files.

For example logging module states `jetty-logging.xml` is relevant for logging configuration. One can alter this file to modify logging configuration.

2.8 Conclusion

In this post, we have defined the related steps to install and configure a standalone Jetty server. Further information can be obtained through the official documentation of Jetty.

Chapter 3

Jetty web.xml Configuration Example

In this example, we will configure Jetty web applications through deployment descriptor files. Typically in a Java web application, the deployment descriptor file is named as web.xml, which includes application-wide configuration. In addition to this, a common deployment descriptor can be defined for a Jetty. This common descriptor is container-wide and includes configuration which is applicable to all of the web applications deployed in Jetty. Typically, this common file is named as webdefault.xml, however it is possible to rename and relocate this file.

In this post, we are going to discuss some fundamental elements (but not all of them) of the deployment descriptors and apply these elements to our Jetty web applications both for container-wide and application specific configurations. Unlike our previous Jetty examples, this time we will mainly utilize Embedded Jetty; however at the end of the example we will show how relevant configuration can be applied in Standalone Jetty container.

3.1 Deployment Descriptor file (a.k.a web.xml)

Deployment Descriptor is an XML file that contains the configuration of a Java Web Application. In Java web applications, it should be in the WEB-INF directory of the web application(or WAR file) and it should be named as "web.xml". Root element of this XML is named as web-app. Below you can see a simple web.xml file.

```
<web-app xmlns="https://java.sun.com/xml/ns/javaee" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://java.sun.com/xml/ns/javaee
              https://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
        version="3.0">
        <display-name>Webapp2</display-name>
</web-app>
```

The element display-name is optional too and serves as an informative field for some GUI tools.

There are a number of elements available in a deployment descriptor file. Full list of available elements can be viewed here. In this example we will skim through the following:

- welcome-file-list
- servlet/ servlet-mapping
- filter/ filter-mapping
- listener

As we have mentioned above, web.xml file stores configurations per application. However, it is possible to specify a common deployment descriptor that holds configuration for multiple web applications. In Jetty, this file is named as webdefault.xml, however this file can be renamed and its location can be configured. Structure of webdefault.xml is no different from a web.xml file.

Another remark is necessary at this point. After Java Servlet Spec 3.0, web.xml is not necessary for a web application and the same configuration can be performed through Java classes and annotations. However, in this example, we will configure our applications with XML files.

3.2 Structure of the Example

In this example we will create two simple web applications (named as webapp1 and webapp2) with their web.xml files, in an embedded Jetty container and provide a common deployment descriptor(webdefault.xml) for them. Thereafter we will configure these applications through webdefault.xml and web.xml files. For common configurations, we are going to modify webdefault.xml and we will observe that both webapp1 and webapp2 are affected from this modifications. In order to demonstrate application specific configurations, we are going to modify the web.xml of webapp1 and we will keep webapp2 configuration as it is. We will see that our application specific configuration only applies to webapp1.

3.3 Environment in the Example

In this example, following programming environment will be used:

- Java 7
- Maven 3.x.y
- Eclipse Luna(as the IDE)
- Jetty v9.2.11 (In Embedded Jetty examples, we will add Jetty libraries through Maven)

3.4 Creating the Maven Project

We will create the Maven project in Eclipse, applying the steps below:

- Go to File → New →Other → Maven Project
- Tick Create a simple project and press “Next”.
- Enter groupId as : com.javacodegeeks.snippets.enterprise
- Enter artifactId as : jetty-webxml-example
- Press “Finish”.

3.5 Creating Embedded Jetty Server and Sample Web Applications

3.5.1 Web Application Configuration

We will configure two simple applications in this example namely webapp1 and webapp2 which are identical initially.

In order to create webapp1, following steps should be followed:

- Create folder webapp1 under the directory /src/main. (src/main/webapp1).
- Create a folder named WEB-INF under src/main/webapp1.
- Create an initial web.xml file under src/main/webapp1/WEB-XML. Content of this web.xml is given below.
- Create a simple html file named jcgindex.html under src/main/webapp1.

The initial web.xml is as follows:

```
<web-app xmlns="https://java.sun.com/xml/ns/javaee" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://java.sun.com/xml/ns/javaee
              https://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
        version="3.0">
        <display-name>Webapp1</display-name>
</web-app>
```

jcgindex.html is a simple html file with following content:

```
<html>
Jetty Webapp 1 Index Page
</html>
```

Now webapp1 is ready to deploy. Our second application, webapp2, can be prepared repeating the same steps described above.(Replacing webapp1 expressions with webapp2 of course).

We will also place a webdefault.xml in our project. In order to do this, following steps should be followed:

- Create webdefault folder under src/main.
- Place a webdefault.xml file under src/main/webdefault.

The file webdefault.xml can be obtained from a standalone Jetty installation. The location is JETTY_HOME/etc/webdefault.xml.

3.5.2 Creating Embedded Jetty

As mentioned above, we are going to run our web applications on Embedded Jetty. For the sake of simplicity, our Embedded Jetty will run through the main class.

EmbeddedJettyWebXmlConfigurationMain code, with descriptive comments is as follows:

EmbeddedJettyWebXmlConfigurationMain.java

```
package com.javacodegeeks.snippets.enterprise.embeddedjetty;

import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.server.handler.HandlerCollection;
import org.eclipse.jetty.webapp.WebAppContext;

public class EmbeddedJettyWebXmlConfigurationMain {

        public static void main(String[] args) throws Exception {

                Server server = new Server(8080);

                // Handler for multiple web apps
                HandlerCollection handlers = new HandlerCollection();

                // Creating the first web application context
                WebAppContext webapp1 = new WebAppContext();
                webapp1.setResourceBase("src/main/webapp1");
                webapp1.setContextPath("/webapp1");
                webapp1.setDefaultsDescriptor("src/main/webdefault/webdefault.xml");
                handlers.addHandler(webapp1);

                // Creating the second web application context
                WebAppContext webapp2 = new WebAppContext();
                webapp2.setResourceBase("src/main/webapp2");
                webapp2.setContextPath("/webapp2");
```

```
                webapp2.setDefaultsDescriptor("src/main/webdefault/webdefault.xml");
                handlers.addHandler(webapp2);

                // Adding the handlers to the server
                server.setHandler(handlers);

                // Starting the Server
                server.start();
                System.out.println("Started!");
                server.join();

        }
}
```

First we create a Server on port 8080. Then we initialize a HandlerCollection, which allows to create multiple web application contexts on a single Server. Thereafter we set the context path and resource base(src/main/webappX) for both web applications. In addition to these, we set default deployment descriptor path (src/main/webdefault/webdefault.xml). After we configure web application contexts we attach these to the HandlerCollections. Lastly we start our embedded server.

When we run our main class, our server starts on port 8080. We can access the two web applications via https://localhost:8080/-webapp1 and https://localhost:8080/webapp2.

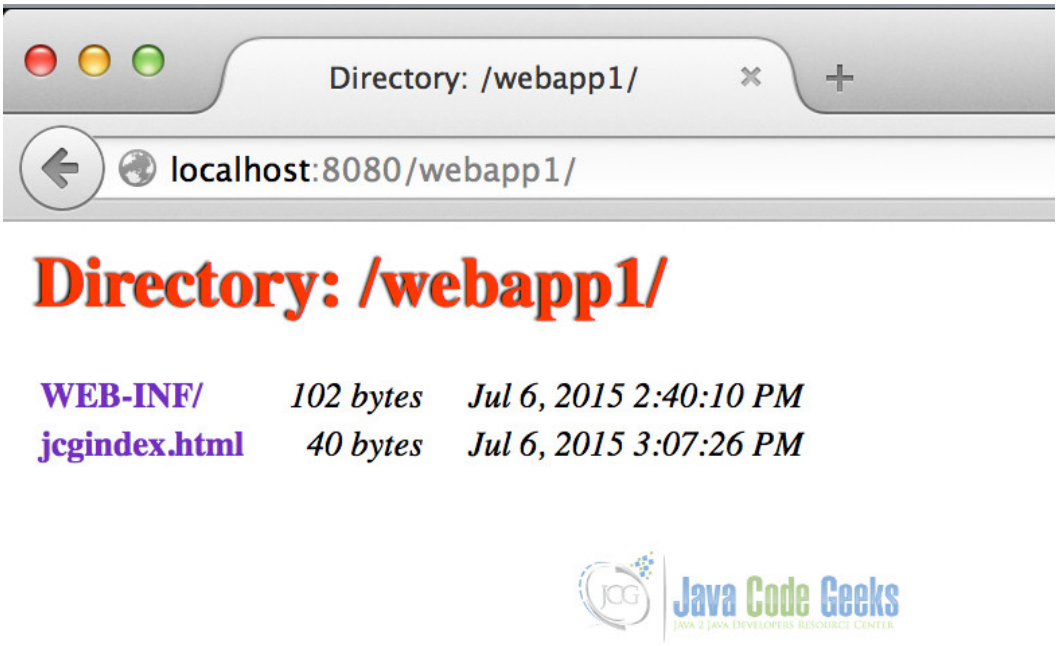

Figure 3.1: WebApp1

Figure 3.2: WebApp2

3.6 Configuring welcome-file-list

welcome-file-list is an element in deployment descriptor which defines a set of lookup files that are automatically looked up upon a request on context root(for example https://localhost:8080/webapp1). Typically in a web application, a welcome file is index.html(or index.htm, index.jsp etc.). When an HTTP request hits the context root, one of the files defined in this list is retrieved to the user. The initial welcome file list in the webdefault.xml is initially as follows:

```
<welcome-file-list>
                <welcome-file>index.html</welcome-file>
                <welcome-file>index.htm</welcome-file>
                <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

None of our sample applications have one of the files listed above. As a result, we obtain a directory listing, when we try to access the context root. (Directory listing can also be disabled by setting dirAllowed parameter of Default Servlet to false in the webdefault.xml). When we add jcgindex.html to the welcome file list, our welcome file list looks as follows:

```
<welcome-file-list>
                <welcome-file>index.html</welcome-file>
                <welcome-file>jcgindex.html</welcome-file>
                <welcome-file>index.htm</welcome-file>
                <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

Now when we try to access context roots(https://localhost:8080/webapp1 or https://localhost:8080/webapp2), the outcome is the content of jcgindex.html. A sample response page can be viewed below:

Figure 3.3: WebApp1 index file

Here we have configured welcome-file-list through webdefault.xml. It is possible to do it through the web.xml also.

3.7 Configuring Servlets

Servlet is a Java class extending server capabilities through HTTP request-response model. Servlets form the backbone of Java Web applications. Deployment descriptors are utilized to configure Servlets in a web application.

A servlet configuration has two main parts:

- Servlet Definition
- Servlet Mapping

Servlet definition defines the class of the Servlet along with a unique name defined by the developer. A Servlet class is a Java class implementing javax.servlet.Servlet interface. Typically, extending javax.servlet.http.HttpServlet is a common practice in servlet implementation.

A servlet definition in deployment descriptor looks like as follows:

```
<servlet>
                <servlet-name>myServlet</servlet-name>
                <servlet-class>org.example.MyServlet</servlet-class>
</servlet>
```

Servlet mapping, defines the URL pattern that is going to be handled by the specified servlet. Multiple URL patterns can be assigned to a single servlet by defining multiple mappings. Here we reference the Servlet by the unique name we decided in the Servlet definition part. The example below defines a servlet mapping that assigns the URLS (/somepath) to our servlet (myServlet):

```
<servlet-mapping>
                <servlet-name>myServlet</servlet-name>
                <url-pattern>/somePath/*</url-pattern>
</servlet-mapping>
```

Having this definition in webdefault.xml, the servlet definition and mapping will be effective for all applications deployed in our Jetty container.

To sum up, following steps have to be applied in order to create and map a servlet.

- Create a servlet definition in deployment descriptor (webdefault.xml for container-wide or web.xml of the desired webapp for the application specific configuration).
- Create a servlet mappings for this servlet.
- Implement the Servlet class.

An XML configuration in webdefault.xml that defines a CommonServlet and maps it to /common pattern both of the web applications is as follows:

```
<servlet>
        <servlet-name>commonServlet</servlet-name>
        <servlet-class> com.javacodegeeks.snippets.enterprise.embeddedjetty.servlet. ↩
            CommonServlet </servlet-class>
</servlet>

<servlet-mapping>
        <servlet-name>commonServlet</servlet-name>
        <url-pattern>/common/*</url-pattern>
</servlet-mapping>
```

The implementation of the Common Servlet is as follows:

CommonServlet.java

```
package com.javacodegeeks.snippets.enterprise.embeddedjetty.servlet;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServlet;

public class CommonServlet extends HttpServlet {

        @Override
        public void service(ServletRequest req, ServletResponse res) throws  ↩
            ServletException, IOException {

                res.getOutputStream().print("Common Servlet Response");
        }
}
```

The CommonServlet simply returns a String as the response. When we run the example, we will get the response below from both applications through https://localhost:8080/webapp1/common and https://localhost:8080/webapp2/common.

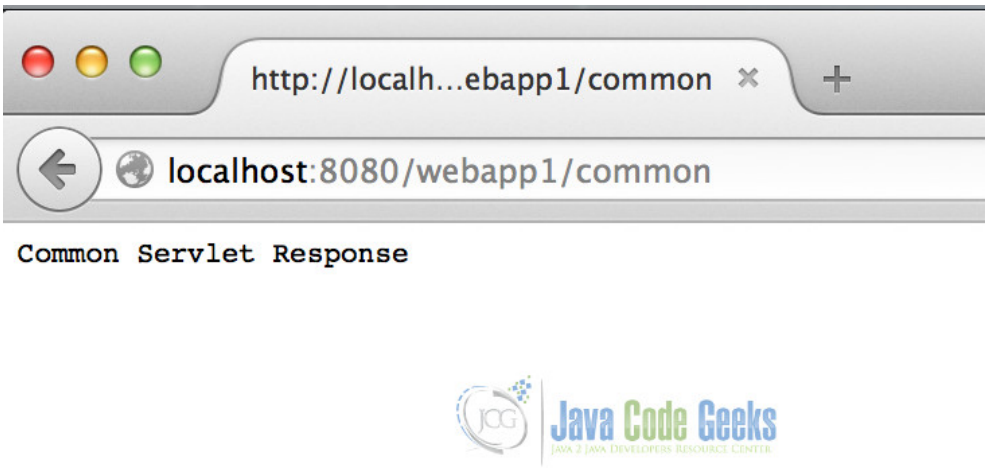

Figure 3.4: Common Server Response

The configuration above is valid for both applications since it is defined in webdefault.xml. In order to be specific for a single application, we should define the servlet and its mappings in web.xml of the relevant application.

We can add the following servlet configuration to the web.xml of webapp1.

```
<servlet>
        <servlet-name>specificServlet</servlet-name>
        <servlet-class>com.javacodegeeks.snippets.enterprise.embeddedjetty.servlet. ↩
            SpecificServlet</servlet-class>
</servlet>

<servlet-mapping>
        <servlet-name>specificServlet</servlet-name>
        <url-pattern>/specific/*</url-pattern>
</servlet-mapping>
```

The SpecificServlet implementation is as follows:

SpecificServlet.java

```
package com.javacodegeeks.snippets.enterprise.embeddedjetty.servlet;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServlet;

public class SpecificServlet extends HttpServlet {

        @Override
        public void service(ServletRequest req, ServletResponse res) throws  ↩
            ServletException, IOException {

                res.getOutputStream().print("Application Specific Servlet Response");
        }
}
```

When we run our example and try to access the URL https://localhost:8080/webapp1/specific/ we will get the following response:

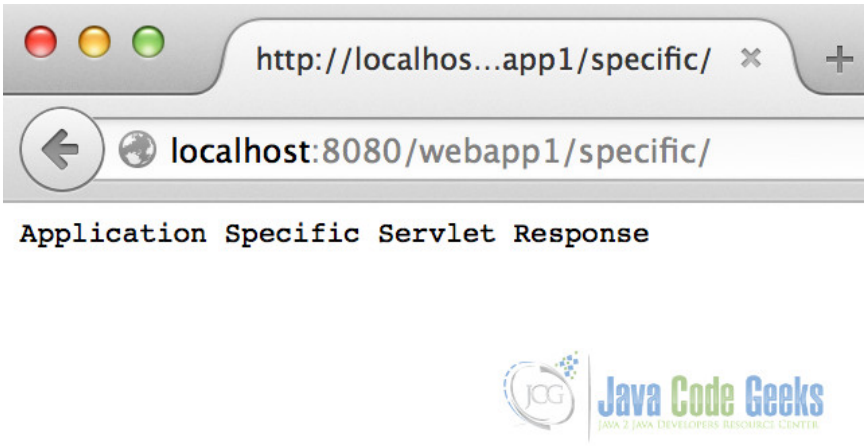

Figure 3.5: Response

This configuration is webapp1 specific. When we try to access webapp2 with the same URL pattern (https://localhost:8080/-webapp2/specific/); we will get a 404 Error immediately.

There are a lot to mention on the Servlet subject and configuration; however they are beyond the scope of this example.

3.8 Configuring Servlet Filters

Servlet Filter is one of the key building blocks in a Java Web Application. Servlet filters intercept HTTP requests/response before and after Servlet invocation. They have many uses: Decorating requests and responses, logging or blocking them for security reasons are among those. Servlet Filter mechanism follows the Chain of Responsibility design pattern. A simple Servlet Filter is as follows:

CommonFilter.java

```
package com.javacodegeeks.snippets.enterprise.embeddedjetty.filter;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class CommonFilter implements Filter {

        FilterConfig filterConfig = null;

        public void init(FilterConfig filterConfig) throws ServletException {
                this.filterConfig = filterConfig;
        }

        public void destroy() {
        }
```

```
        public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse ↩
            , FilterChain filterChain)
                        throws IOException, ServletException {

                System.out.println("Common first!");
                filterChain.doFilter(servletRequest, servletResponse);
                System.out.println("Common last!");


        }

}
```

The first print line is invoked when the request is intercepted. The control is delegated to the next filter in the responsibility chain. The last print line is invoked after the rest of the chain completes its work.

Definition of the Servlet filters is very similar to the Servlet: We have to define the filter and map URLs to this filter. We can configure the CommonFilter in webdefault.xml as follows:

```
<filter>
        <filter-name>CommonFilter</filter-name>
        <filter-class>com.javacodegeeks.snippets.enterprise.embeddedjetty.filter. ↩
            CommonFilter</filter-class>

</filter>
<filter-mapping>
        <filter-name>CommonFilter</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

Here we have mapped this filter to the all URLs under our both web applications. When we run the example and try to access any URL of these applications, We observe the following lines in the server output:

```
Common first!
Common last!
```

As in the servlets, the configuration in webdefault.xml is valid for both applications. In order to be application specific, you can define another filter in the web.xml of webapp1 and implement is as follows:

```
<filter>
                <filter-name>SpecificFilter</filter-name>
                <filter-class>com.javacodegeeks.snippets.enterprise.embeddedjetty.filter. ↩
                    SpecificFilter</filter-class>

        </filter>
        <filter-mapping>
                <filter-name>SpecificFilter</filter-name>
                <url-pattern>/*</url-pattern>
        </filter-mapping>
```

SpecificFilter.java

```
package com.javacodegeeks.snippets.enterprise.embeddedjetty.filter;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
```

```
public class SpecificFilter implements Filter {

        FilterConfig filterConfig = null;

        public void init(FilterConfig filterConfig) throws ServletException {
                this.filterConfig = filterConfig;
        }

        public void destroy() {
        }

        public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse ↩
            , FilterChain filterChain)
                        throws IOException, ServletException {

                System.out.println("Specific Filter first!");
                filterChain.doFilter(servletRequest, servletResponse);
                System.out.println("Specific Filter last!");


        }

}
```

When we run the server and try to access a URL in webapp1, we will observe the following server output:

```
Common first!
Specific Filter first!
Specific Filter last!
Common last!
```

Here you can see, first line is the first print line of the CommonFilter; that is followed by the first and last print lines of the SpecificFilter. The output is finalized with the last print line of the CommonFilter. This output sequence summarizes the mechanism of filter chain of Java web apps.

Since the SpecificFilter is configured only for webapp1; when we try to access webapp2; we will only observe the outputs of the CommonFilter.

3.9 Configuring Servlet Context Listeners

ServletContextListener is another core block in Java Web applications. It is an interface whose implementations are invoked upon web application context creation and and destruction events.

A concrete ServletContextListener has to implement two methods:

- contextInitialized
- contextDestroyed

A sample implementation, printing the context path of the implementation is as follows:

CommonListener.java

```
package com.javacodegeeks.snippets.enterprise.embeddedjetty.listener;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class CommonListener implements ServletContextListener {
```

```
        public void contextInitialized(ServletContextEvent sce) {
                System.out.println("Context initialized:"+sce.getServletContext(). ↩
                    getContextPath());

        }

        public void contextDestroyed(ServletContextEvent sce) {


        }

}
```

In webdefault.xml, listener configuration is below:

```
<listener>
        <listener-class>com.javacodegeeks.snippets.enterprise.embeddedjetty.listener. ↩
            CommonListener</listener-class>
</listener>
```

When we start the server, the listener will be invoked for both webapp1 and webapp2. We will obtain the following server output:

```
2015-07-07 16:01:18.648:INFO::main: Logging initialized @295ms
2015-07-07 16:01:18.736:INFO:oejs.Server:main: jetty-9.2.11.v20150529
2015-07-07 16:01:18.857:INFO:oejw.StandardDescriptorProcessor:main: NO JSP Support for / ↩
    webapp1, did not find org.eclipse.jetty.jsp.JettyJspServlet
Context initialized:/webapp1
2015-07-07 16:01:18.884:INFO:oejsh.ContextHandler:main: Started o.e.j.w. ↩
    WebAppContext@58134517{/webapp1,file:/Users/ibrahim/Documents/workspace_jcg/jetty-webxml ↩
    -example/src/main/webapp1/,AVAILABLE}
2015-07-07 16:01:18.900:INFO:oejw.StandardDescriptorProcessor:main: NO JSP Support for / ↩
    webapp2, did not find org.eclipse.jetty.jsp.JettyJspServlet
Context initialized:/webapp2
2015-07-07 16:01:18.902:INFO:oejsh.ContextHandler:main: Started o.e.j.w. ↩
    WebAppContext@226a82c4{/webapp2,file:/Users/ibrahim/Documents/workspace_jcg/jetty-webxml ↩
    -example/src/main/webapp2/,AVAILABLE}
2015-07-07 16:01:18.919:INFO:oejs.ServerConnector:main: Started ServerConnector@691a7f8f{ ↩
    HTTP/1.1}{0.0.0.0:8080}
2015-07-07 16:01:18.920:INFO:oejs.Server:main: Started @569ms
Started!
```

Again, we may wish to configure a listener for a single web application. Then we should define our listener configuration in the related web.xml with the required implementation.

```
<listener>
                <listener-class>com.javacodegeeks.snippets.enterprise.embeddedjetty. ↩
                    listener.SpecificListener</listener-class>
</listener>
```

SpecificListener.java

```
package com.javacodegeeks.snippets.enterprise.embeddedjetty.listener;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class SpecificListener implements ServletContextListener {

        public void contextInitialized(ServletContextEvent sce) {
                System.out.println("Specific Context initialized:"+sce.getServletContext(). ↩
                    getContextPath());
```

```
        }

        public void contextDestroyed(ServletContextEvent sce) {
                // TODO Auto-generated method stub

        }

}
```

When we start the server, we will see that SpecificListener is invoked upon creation of webapp1 context. The output is as follows:

```
2015-07-07 16:20:33.634:INFO::main: Logging initialized @210ms
2015-07-07 16:20:33.740:INFO:oejs.Server:main: jetty-9.2.11.v20150529
2015-07-07 16:20:33.864:INFO:oejw.StandardDescriptorProcessor:main: NO JSP Support for / ↩
    webapp1, did not find org.eclipse.jetty.jsp.JettyJspServlet
Context initialized:/webapp1
Specific Context initialized:/webapp1
2015-07-07 16:20:33.895:INFO:oejsh.ContextHandler:main: Started o.e.j.w. ↩
    WebAppContext@4450d156{/webapp1,file:/Users/ibrahim/Documents/workspace_jcg/jetty-webxml ↩
    -example/src/main/webapp1/,AVAILABLE}
2015-07-07 16:20:33.907:INFO:oejw.StandardDescriptorProcessor:main: NO JSP Support for / ↩
    webapp2, did not find org.eclipse.jetty.jsp.JettyJspServlet
Context initialized:/webapp2
2015-07-07 16:20:33.908:INFO:oejsh.ContextHandler:main: Started o.e.j.w. ↩
    WebAppContext@731f8236{/webapp2,file:/Users/ibrahim/Documents/workspace_jcg/jetty-webxml ↩
    -example/src/main/webapp2/,AVAILABLE}
2015-07-07 16:20:33.926:INFO:oejs.ServerConnector:main: Started ServerConnector@50a7bc6e{ ↩
    HTTP/1.1}{0.0.0.0:8080}
2015-07-07 16:20:33.926:INFO:oejs.Server:main: Started @506ms
Started!
```

Note that, the common configuration elements are invoked before application specific ones.

3.10 Configuration in Standalone Jetty Server

In this example, we have performed deployment descriptor configuration in an Embedded Jetty container. For the standalone Jetty(v9.2.11), the the path of default configuration (webdefault.xml) is under JETTY_HOME/etc. If you want to change location of the default configuration path, then you have to alter the defaultsDescriptor element of JETTY_HOME/etc/jetty-deploy.xml.

3.11 Conclusion

In this post, we have provided information on Web application configuration in Jetty through the deployment descriptor files. We have skimmed through configuration of main building blocks of a web application configuration(Servlets, Servlet Filters and Listeners). While providing examples, we have emphasized that Jetty allows both container-wide and application specific configuration.

Download

You can download the full source code of this example here: **JettyWebXmlExample**

Chapter 4

Jetty Servlet Example

In this example, we will show you how to make use of Jetty - Java HTTP Web Server and servlet container and run a sample servlet on this server. Jetty is an open source web server developed by Eclipse Foundation. As a part of this example we will create an eclipse project which will have our servlet code and to deploy on jetty, we will configure that project in a war file.

4.1 Environment

In this example, following environment will be used:

- Eclipse Kepler 4.3 (as IDE)
- Jetty - version 9.2.15 v20160210
- Java - version 7
- Java Servlet Library - servlet-api-3.0

4.2 Jetty Servlet Example

4.2.1 Structure of the example

In this example, we are going to write a simple servlet and run that servlet on Jetty web server. We will package our servlet project in a WAR file. We can then deploy this war file on running jetty server and it will dynamically detect our servlet.

4.2.2 Running Jetty

Make sure you download the correct version of Jetty from Download Jetty. Certain versions of jetty only run with certain versions of Java. You might run into an error `java:unsupported major:minor version 52.0`. Once you extract downloaded jetty zip file on your machine, you can open a command prompt and navigate to directory `/demo-base` and run `java -jar ../start.jar`, this will start our jetty web server. To verify everything is alright with our jetty installation, launch a web browser and go to url https://localhost:8080, it should show a Jetty welcome page.

4.2.3 Creating Example Servlet

We will create a Dynamic Web Project in eclipse. Follow the steps below:

- Go to File → New Project → Web → Dynamic Web Project

- Provide a name for your project "FirstServletJetty" and choose Target runtime as None

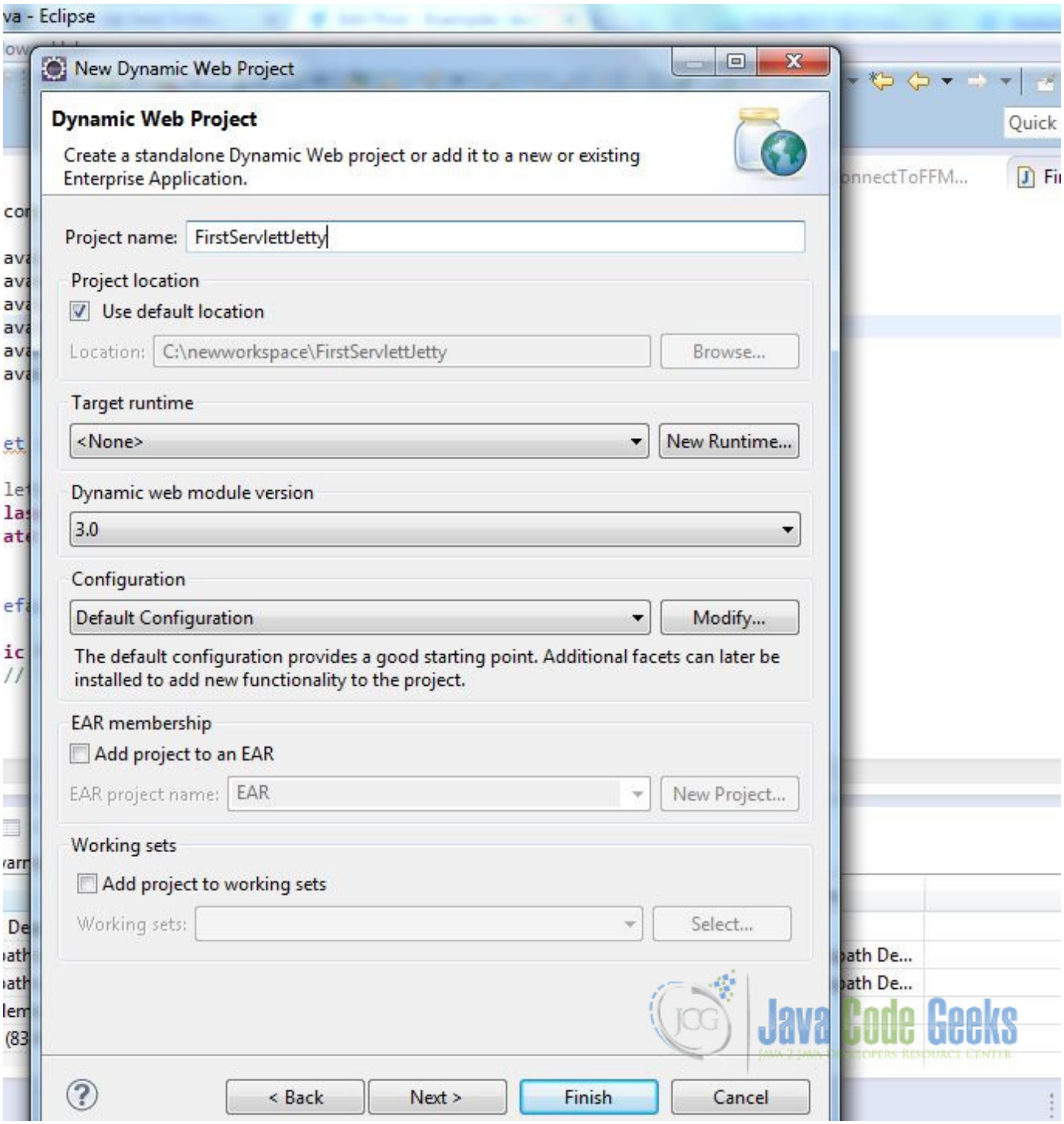

Figure 4.1: Create Dynamic Web Project in Eclipse

- Click Next. On Web Module screen, select the checkbox for "Generate web.xml deployment descriptor"

After creating our project, we will need to sort out dependencies to write our first servlet. Download servlet-api-3.0.jar file and import that in our project's build path.

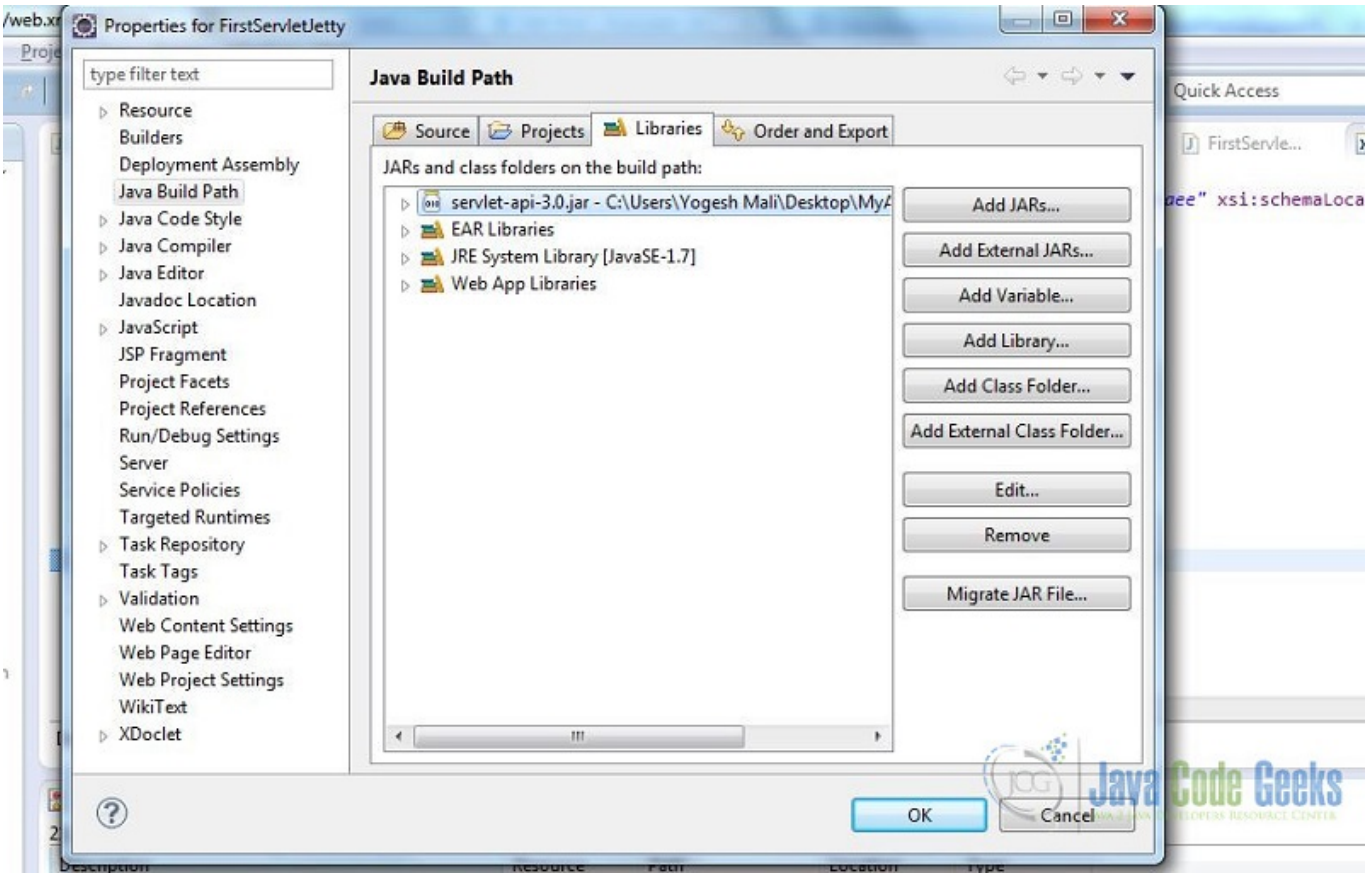

Figure 4.2: Add servlet-api jar file to build path

- Go to Src folder in project directory and right click to select New Servlet
- Enter package name: com.javacodegeeksexample
- Enter servlet name: FirstServlet
- Keep default options and click Finish

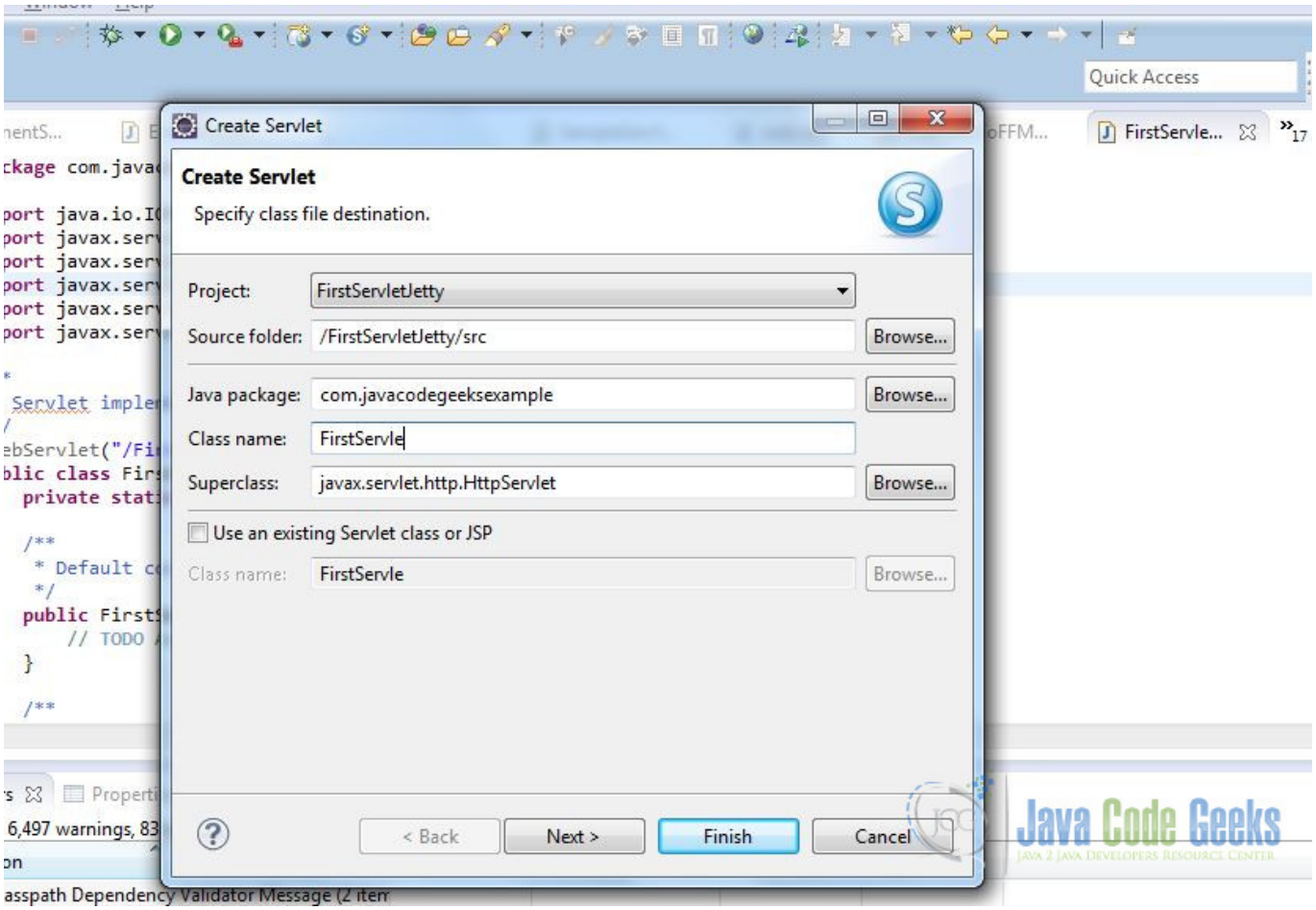

Figure 4.3: Create First Servlet

4.2.4 Modifying Example Servlet

Now we can write our code in the servlet we just created. We can write our code in `doGet` or `doPost`. We will write very simple print statement to see how our servlet behave once deployed on webserver.

This is how the final code of FirstServlet looks:

FirstServlet.java

```
package com.javacodegeeksexample;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class FirstServlet
 */
@WebServlet("/FirstServlet")
public class FirstServlet extends HttpServlet {
        private static final long serialVersionUID = 1L;
```

```
    /**
     * Default constructor.
     */
    public FirstServlet() {
        // TODO Auto-generated constructor stub
    }

        /**
         * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
         */
        protected void doGet(HttpServletRequest request, HttpServletResponse response)  ↩
            throws ServletException, IOException {

                response.setContentType("text/html");
                response.getWriter().println("First Servlet on Jetty - Java Code Geeks");
        }

        /**
         * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response ↩
             )
         */
        protected void doPost(HttpServletRequest request, HttpServletResponse response)  ↩
            throws ServletException, IOException {

        }

}
```

Save your source code file and build the project in eclipse.

4.2.5 Deploying your servlet on Jetty

- Save project → Export → Web → WAR file

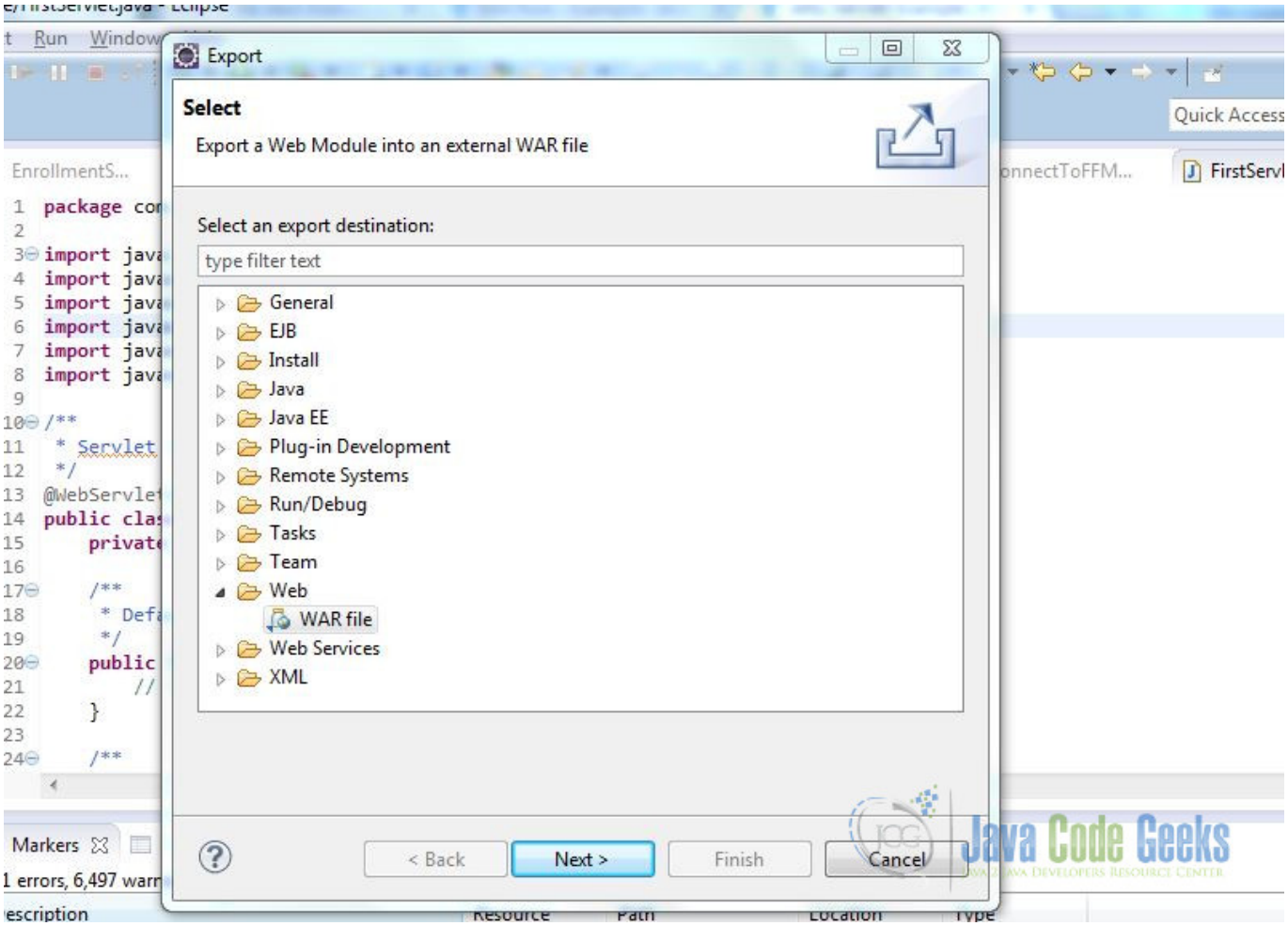

Figure 4.4: Export the project as WAR file

- Save this WAR file in /demo-base/webapps directory
- If jetty is already running, it should detect your new servlet deployed OR you can restart the jetty server

4.2.6 Running the Servlet

To verify everything is correct, you can access this servlet in webbrowser at https://localhost:8080/FirstServletJetty/FirstServlet

Figure 4.5: Running the servlet in browser

Text on this webpage is coming from whatever we printed in `doGet` method.

4.2.7 More with Servlet

This was a very simple servlet to run on jetty. We can do more complicated `jsp` or `html` pages to call servlets to handle requests.

Let's add a simple form on an html page and do get action and post action subsequently.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title> First HTML PAGE </title>
</head>
<body>
<p>Let's Test doPost.
<form method="POST" action="FirstServlet"/>
<input name="field" type="text" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

Now if you see this html page has a `form` with method `POST` which will get called through action `FirstServlet`. Once the form is submitted, `FirstServlet` will call `doPost` to handle the request posted through form submission.

We can modify `doPost` method to read parameters posted through form.

```
/**
* @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
*/
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ↩
    ServletException, IOException {
        response.getWriter().println("In POST - First Servlet content - Java code geeks");
        response.getWriter().println(request.getParameter("field"));

}
```

Most companies build their login pages through form like this and then in doPost handles the submission of those forms for authentication. Once we build our project and export it as a war file to deploy on server, we can access our html page like this https://localhost:8080/FirstServletJetty/FirstPage.html. Output will look like below

Figure 4.6: First Page Sample Form Submission - doPost

You can type something in the textbox and press `Submit`. Result will be as shown below

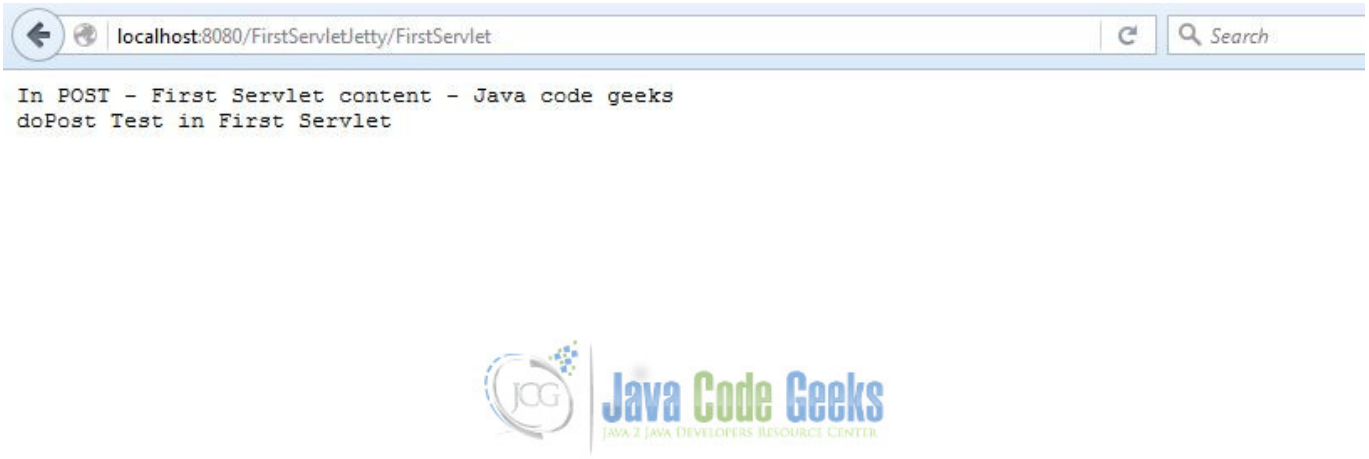

Figure 4.7: doPost result

4.3 Conclusion

In this example, we saw how to deploy a simple servlet on a jetty web server. Another way of mapping your servlet is by adding a servlet-mapping in web.xml of your Dyanmic Web Project, but in that case you will need to remove the annotations `@WebServlet` in your java source code.

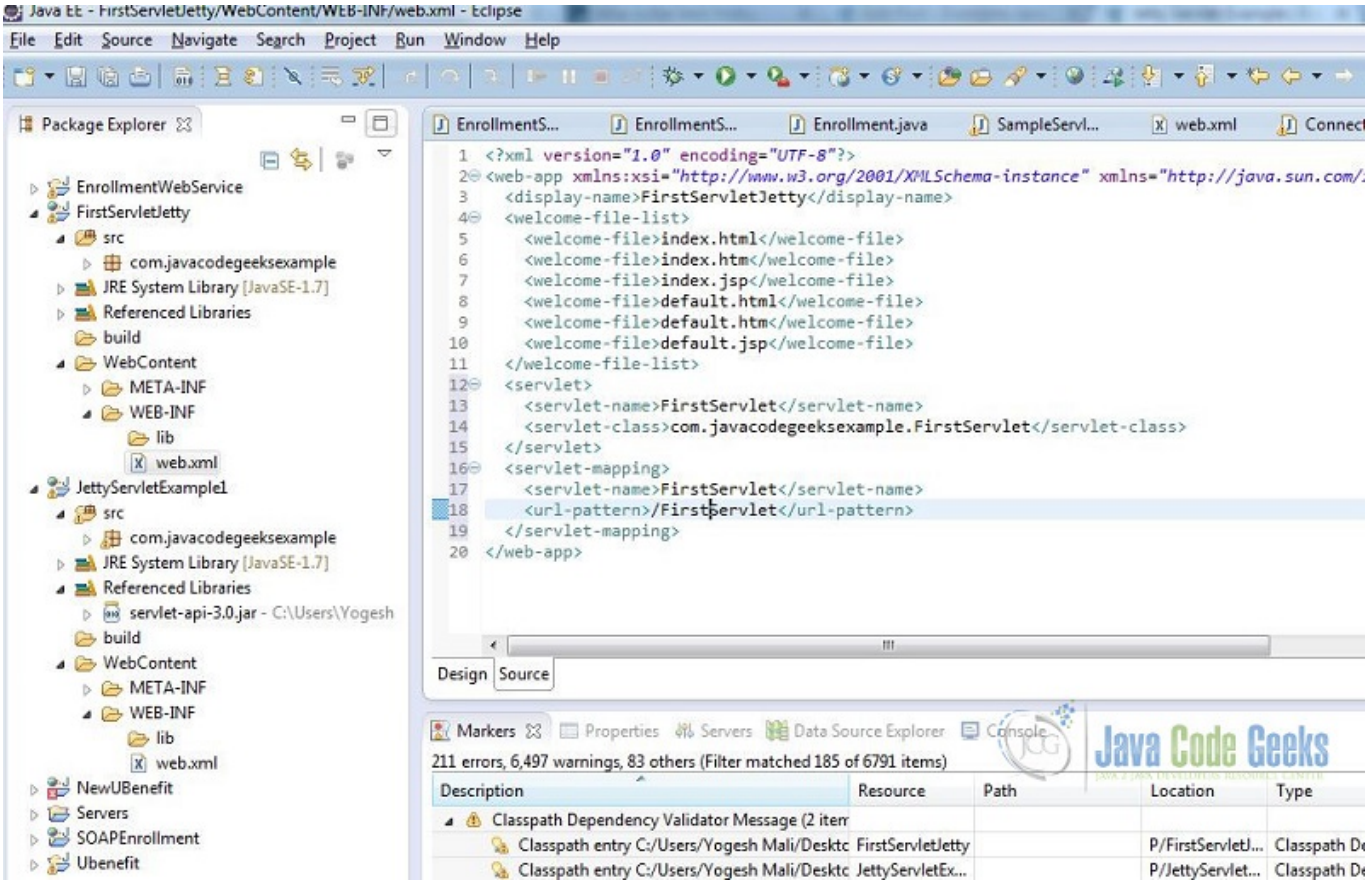

Figure 4.8: Mapping the servlet

4.4 Download the eclipse project

This was an example of Servlet on Jetty.

Download

You can download the full source code of this example here: **JettyServletExample**

Chapter 5

Jetty Logging Configuration Example

In this example, we will discuss logging capabilities of Jetty. We will first enable the logging module in Jetty and configure it afterwards. As in the previous Jetty examples, we will start with standalone Jetty; thereafter we will configure logging for Embedded Jetty server too.

We are going to use Jetty v9.2.11 in this example, along with Java 8 (7 is also fine) and Apache Maven 3 as the environment. In addition to these, logging frameworks SLF4J and Logback will be utilized to configure logging in Jetty.

5.1 Logging in Jetty

Jetty has its own logging Logging layer which had emerged before any popular Java logging frameworks (around 1995). But Jetty does not mandate its logging layer. Other modern logging frameworks (SLF4J with Logback or Log4j or any other) can be used in Jetty logging; moreover one can wire his own logging implementation to extend Jetty's logging capabilities.

Jetty determines its logging behavior according to the following rules:

- First, the value of the property `org.eclipse.jetty.util.log.class` is checked. If defined, the logger implementation is that class.
- If `org.slf4j.Logger` exists in the classpath, logging is decided as SLF4J.
- Otherwise, `org.eclipse.jetty.util.log.StdErrLog` is the default logging behavior.

In this example we are going to first configure Jetty with with default behavior, thereafter we will enhance it with Logback and SLf4J.

5.2 Environment

In this example, following programming environment is used:

- Java 8 (Java 7 is also OK for this example)
- Jetty v9.x (We have used v9.2.11)
- Apache Maven 3.x.y (for Embedded Jetty Example)
- Eclipse Luna(for Embedded Jetty Example)

5.3 Enabling Logging in Jetty

Jetty 9 has a modular architecture, which means that different features (logging, SSL, SPDY, websockets etc.) are implemented as modules. These modules have to be turned on or off based on the needs.

The modules of Jetty are enabled or disabled through the `start.ini` file under your JETTY_HOME.

In order to activate logging module, the steps needed are below:

- Navigate to the `JETTY_HOME`
- Open `start.ini`.
- Add following line to `start.ini` as save the file:

```
--module=logging
```

By enabling the logging module, we have activated these files:

- `JETTY_HOME/modules/logging.mod`
- `JETTY_HOME/etc/jetty-logging.xml`

Further configuration will be performed via modifying these files.

Since we have not performed any logging configuration yet, Jetty will use by default `org.eclipse.jetty.util.log.StdErrLoglogger` (the 3rd option among the ones listed above).

Before you start Jetty, check the `JETTY_HOME/logs` directory and see that it is empty. Now you can start jetty running the following command in your `JETTY_HOME`.

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

Now you can see the output similar to the following:

```
2015-06-27 16:59:09.091:INFO::main: Redirecting stderr/stdout to /Users/ibrahim/jcgexamples ↩
    /jetty/jetty-distribution-9.2.11.v20150529/logs/2015_06_27.stderrout.log
```

The output line means that the Jetty now logs to the file `yyyy_mm_dd.stderrout` (yyyy_mm_dd is based on the current date) under `JETTY_HOME/logs` directory. You can see the log files in this directory. If you can see the log file under the logs directory, it means that we have successfully enabled logging module of Jetty.

5.4 Configuring SLF4J with Logback in Jetty

As we have mentioned earlier; it is possible to use any popular Java logging framework with Jetty. In this part, we will configure our Jetty with SLF4J and Logback.

In order to configure SLF4J with Logback, first we need to have following JAR files:

- SLF4J API
- logback-core
- logback classic

After you get these JAR files, we have to copy these under your Jetty installation with these steps:

- Create the directory logging under JETTY_HOME.

- Copy these 3 JAR files to this directory (JETTY_HOME/logging).

After adding the files to our classpath, we should (although not mandatory) add a `logback.xml` file to the directory `JETTY_HOME/resources`. In case you did not have one, an example file is provided below.

logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration scan="true">
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <charset>utf-8</charset>
            <Pattern>[%p] %c - %m%n</Pattern>
        </encoder>
    </appender>


    <logger name="org.eclipse" level="INFO"/>



    <contextListener class="ch.qos.logback.classic.jul.LevelChangePropagator">
        <resetJUL>true</resetJUL>
    </contextListener>

    <root level="DEBUG">
        <appender-ref ref="CONSOLE"/>
    </root>

</configuration>
```

When you start Jetty with the Logback configuration, you will observe a different log output in your `JETTY_HOME/logs` directory. You can increase verbosity of your logging output changing logging level of “org.eclipse” logger from “INFO” to “DEBUG”. When you restart your Jetty, you will see a more verbose log.

5.5 Changing the Location and Name of the Jetty Log Files

By default, Jetty logs to is `yyyy_mm_dd.stderrout.log` file under `JETTY_HOME/logs`. You can modify the location of the log files and the log file names. These configurations are performed via `logging.mod` and `jetty-logging.xml` files.

In order to define a new location for the log files, The needed steps are below:

- Navigate to `JETTY_HOME/modules` directory.
- Open `logging.mod` file.
- Uncomment the line with the parameter with `jetty.logs`
- In order to set the new location (newlogs for example), set the parameter as `jetty.logs=newlogs`. Please note that the location can be either relative to your `JETTY_HOME` or absolute.
- Save the file and close.
- Create a directory named `newlogs` under your `JETTY_HOME`.

When you start your Jetty again, you will observe that your logs are created under `JETTY_HOME/newlogs` directory.

In order to change the filename of the outputs, you have to alter `jetty-logging.xml` file:

- Navigate to `JETTY_HOME/etc` directory.

- Open `jetty-logging.xml` file.
- Replace `yyyy_mm_dd.stderrout.log` with your preferred file name (for instance `yyyy_mm_dd.javacodegeeks.log`).
- Save and close the file.

When you restart your Jetty, you will see log files are named `yyyy_mm_dd.javacodegeeks.log` based on the current date.

5.6 Logging Configuration of Embedded Jetty

In the previous sections, we have explained how we can enable and configure logging in standalone Jetty. From now on, we are going to discuss logging configuration on Embedded Jetty. As in the standalone example, we will first start with the default logging facility of Jetty thereafter we will configure SLF4J and Logback.

5.6.1 Environment

As mentioned above, the programming environment is as follows:

- Java 8 (or Java 7)
- Jetty v9.x (v9.2.11 in this example)
- Apache Maven 3.x.y
- Eclipse Luna (or any convenient IDE)

5.6.2 Creating the Project

We will first create the Maven project in Eclipse, applying the steps below:

- Go to File → New →Other → Maven Project
- Tick Create a simple project and press “Next”.
- Enter groupId as : com.javacodegeeks.snippets.enterprise
- Enter artifactId as : jetty-logging-example
- Press “Finish”.

5.6.3 Maven Dependencies

We need to add only jetty-server dependency to our `pom.xml`. Default logging does not require any additional dependency. The dependency entry looks as follows in the pom:

```
<dependency>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-server</artifactId>
        <version>9.2.11.v20150529</version>
</dependency>
```

For the SLF4J, Logback example we are going to need additional dependencies (logback-classic). We will address this in the related section. In the source code of this example, you can simply comment out additional dependencies.

5.6.4 Default Logging Example

After adding configuring our pom, we are now ready to code. In order to keep things simple in this example, we are going to create our Embedded Jetty server in our main class.

Our main class is `JettyLoggingMain` under the package `com.javacodegeeks.snippets.enterprise.enterprise.jettylogging`.

The source code of `JettyLoggingMain` decorated with the descriptive comment lines is as follows:

JettyLoggingMain.java

```
package com.javacodegeeks.snippets.enterprise.enterprise.jettylogging;

import java.io.PrintStream;

import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.util.RolloverFileOutputStream;
import org.eclipse.jetty.util.log.Log;

public class JettyLoggingMain {

        public static void main(String[] args) throws Exception {

                //We are configuring a RolloverFileOutputStream with file name pattern  and ↩
                     appending property
                RolloverFileOutputStream os = new RolloverFileOutputStream(" ↩
                    yyyy_mm_dd_jcglogging.log", true);

                //We are creating a print stream based on our RolloverFileOutputStream
                PrintStream logStream = new PrintStream(os);

                //We are redirecting system out and system error to our print stream.
                System.setOut(logStream);
                System.setErr(logStream);

                //We are creating and starting out server on port 8080
                Server server = new Server(8080);
                server.start();

                //Now we are appending a line to our log
                Log.getRootLogger().info("JCG Embedded Jetty logging started.", new Object ↩
                    []{});

                server.join();

        }

}
```

In the code, we have first created a `RolloverFileOutputStream` object . We created this object with two parameters.

First one is the filename pattern. In order to specify date in the log file, this filename has to include a pattern like `yyyy_mm_dd`. Otherwise, Jetty will simply create a file with the name specified (without any date information). In this example we have named this pattern as `yyyy_mm_dd_jcglogging.log`.

The second parameter is `append`. When set to true, the logger will append to an existing file for each restart. Otherwise, it will create a new file (with a timestamp information) at each restart. In this example, we set the parameter as “true”.

Then we have created a `PrintStream` and provided our `RolloverFileOutputStream` as the argument. We have directed `sysout` and `syserr` to this `PrintStream`.

Now our logging configuration is complete. In the following lines of code, we start our Embedded Server and append a simple log line.

When we run our main class, our server starts at port 8080. Our logging file (`2015_06_28_jcglogging.log`) is created at our project directory. The content looks like as follows:

```
2015-06-28 00:46:36.181:INFO::main: Logging initialized @134ms
2015-06-28 00:46:36.212:INFO:oejs.Server:main: jetty-9.2.11.v20150529
2015-06-28 00:46:36.241:INFO:oejs.ServerConnector:main: Started ServerConnector@2077d4de{ ↩
    HTTP/1.1}{0.0.0.0:8080}
2015-06-28 00:46:36.242:INFO:oejs.Server:main: Started @198ms
2015-06-28 00:46:36.242:INFO::main: JCG Embedded Jetty logging started.
```

5.6.5 SLF4J and Logback Example

In the first part, we have created an Embedded Jetty with default configuration. In order to configure `SLF4J` and `Logback`, you have to apply two steps:

- Add `SLF4J` and `Logback` dependencies to your pom.xml (in addition to Jetty server dependencies).
- Add a logback.xml file to your classpath.(This step is optional but needed for detailed configuration). You can copy the logback.xml you have used in the standalone example under `src/main/resources`.

The dependency to be added is:

- ch.qos.logback:logback-classic (v1.0.7)

This single dependency also fetches logback-core and SLF4J form the Maven repository. After adding this dependency, your dependencies section in the `pom.xml` looks as follows:

```
<dependencies>

                <dependency>
                        <groupId>org.eclipse.jetty</groupId>
                        <artifactId>jetty-server</artifactId>
                        <version>9.2.11.v20150529</version>
                </dependency>

                <dependency>
                        <groupId>ch.qos.logback</groupId>
                        <artifactId>logback-classic</artifactId>
                        <version>1.0.7</version>
                </dependency>
        </dependencies>
```

For the Logback example, you don't have to modify any single line of code. As we have mentioned above, when Jetty finds SLF4J in the classpath, it will automatically switches to SLF4J(case 2). When you run the same main class of the previous example, you will see the SLF4J log in `yyyy_mm_dd_jcglogging.log`.

```
[INFO] org.eclipse.jetty.util.log - Logging initialized @367ms
[INFO] org.eclipse.jetty.server.Server - jetty-9.2.11.v20150529
[INFO] org.eclipse.jetty.server.ServerConnector - Started ServerConnector@25b26eee{HTTP ↩
    /1.1}{0.0.0.0:8080}
[INFO] org.eclipse.jetty.server.Server - Started @435ms
[INFO] org.eclipse.jetty.util.log - JCG Embedded Jetty logging started.
```

Now our example with Embedded Jetty is complete.

5.7 Conclusion

In this post we have first configured a standalone Jetty server for logging. We have started with enabling logging in Jetty. Then we have configured Jetty both for default Jetty logging and SLF4-Logback logging. Thereafter we have repeated same configuration programmatically for an embedded Jetty Server.

For further configuration with other parameters and logging frameworks, you can refer to the official Jetty documentation on logging.

5.8 Download the Source Code

Download

You can download the full source code of this example here: **JettyLoggingExample**

Chapter 6

Jetty Resource Handler Example

In this example, we will elaborate Resource Handlers in Jetty. Jetty Handlers are classes that are used for handling the incoming requests. They implement the interface org.eclipse.jetty.server.Handler on their specific purpose. Resource Handler is a specific Handler implementation whose purpose is serving static content (images, html pages or other) through a Jetty Server.

In this example, we are going to start with an Embedded Jetty example and configure it programmatically to serve static content via a Resource Handler. Later on, we are going to configure a Resource Handler through XML configuration files in a standalone Jetty server.

6.1 Environment

In the example, following environment will be used:

- Java 7
- Maven 3.x.y
- Eclipse Luna(as the IDE)
- Jetty v9.2.11 (In Embedded Jetty example, we will retrieve Jetty libraries through Maven)

6.2 Creating the Maven Project for the Embedded Example

We will create the Maven project in Eclipse, applying the steps below:

- Go to File → New →Other → Maven Project
- Tick Create a simple project and press “Next”.
- Enter groupId as : com.javacodegeeks.snippets.enterprise
- Enter artifactId as : jetty-resourcehandler-example
- Press “Finish”.

After creating the project, we have to add following dependency to our pom.xml:

```
<dependency>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-server</artifactId>
        <version>9.2.11.v20150529</version>
</dependency>
```

6.3 Creating Sample Static Content

In this part, we are going to create trivial static content that is going to be served through our Embedded Jetty Server. First we have to going to create a directory in order to store the content (it is named as "Resource Base" in Jetty terminology), then we are going to put a simple text file in the in it (the content that is going to be served). The steps can be summed up as follows:

- Create a directory named **jcgresources** under the Eclipse project folder. That is going to be our resource base in this example.
- Create a text file **jcg1.txt** with some trivial content under the directory `PROJECT_BASE/jcgresources`.

Now we are good to continue with the programming part.

6.4 Programmatically Creating Resource Handlers in Embedded Jetty

After creating the static content, now we are going to create an embedded Jetty server programmatically. As in our previous examples, we are going to run the Embedded Jetty within our main class in order to keep things simple.

First we are going to provide the Java source of our main class, which is decorated with comment lines. Afterwards, we are going to discuss the comment lines in order to detail our example. Below you can find the source code of the main class:

EmbeddedJettyResourceHandlerMain.java

```
package com.javacodegeeks.snippets.enterprise.embeddedjetty;

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

public class EmbeddedJettyResourceHandlerMain {

        public static void main(String[] args) throws Exception {

                Server server = new Server(8080);

                //1.Creating the resource handler
                ResourceHandler resourceHandler= new ResourceHandler();

                //2.Setting Resource Base
                resourceHandler.setResourceBase("jcgresources");

                //3.Enabling Directory Listing
                resourceHandler.setDirectoriesListed(true);

                //4.Setting Context Source
                ContextHandler contextHandler= new ContextHandler("/jcg");

                //5.Attaching Handlers
                contextHandler.setHandler(resourceHandler);
                server.setHandler(contextHandler);


                // Starting the Server

                server.start();
                System.out.println("Started!");
                server.join();

        }
}
```

Now we are going to expand the commented lines:

6.4.1 Creating the Resource Handler

ResourceHandler is the class that handles the requests to the static resources. It provides a number of properties to configure.

6.4.2 Setting Resource Base

Resource Base is the root directory of the for the static content. It is relative to the Java application. In the previous section, we had created the resource base under the Eclipse project; thus we are setting Resource Base relative to this project base. It is also possible to set an absolute path, or a path relative to the Java classpath for the Resource Base.

6.4.3 Enabling Directory Listing

Directory Listing enables listing of the contents in the resource directories. It is disabled by default. When enabled, Jetty will provide a simple HTML page listing the directory content; otherwise, it will give an HTTP 403 error.

6.4.4 Setting Context Source

This part is optional, When we create and set a context handler, we are able to set a context root `/jcg`, so we are going to able to access our resources through https://localhost:8080/jcg. If not set, we https://localhost:8080 would point to our resource base.

6.4.5 Attaching Handlers

This part is a boiler plate code that attaches the handler to the server.

6.5 Running the Server

When we run the application, our server will start on port 8080. As mentioned above, we can access the resources through https://localhost:8080/jcg. When we open this URL, the output will be as follows:

Figure 6.1: Directory listing for /jcg

Through this listing page, we can access the available resources.

6.6 Other Configuration

In the previous sections, we have provided sample configuration for resource handling. Jetty provides a variety of configuration options for resource handling that are not going to be detailed in this example. Some of them are:

- Customizing the style of the directory listing with a CSS file.
- Setting a welcome page.
- Configuring multiple resources pages
- Customizing the available content types

6.7 Standalone Jetty Example

Now we are going to configure Resource Handler for Standalone Jetty. The configuration is similar to the Embedded one, just in XML format. The steps required can be summarized as follows:

- Open `jetty.xml` file which is under JETTY_HOME/etc.
- Add the Resource Handler XML configuration to the handler element(which is given below)
- Save the file and run Jetty.

The handler element in `jetty.xml` seems as follows:

```
<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 class="org.eclipse.jetty.server.handler.ContextHandler">
                                  <Set name="contextPath">/jcg</Set>
                                  <Set name="handler">
                                          <New class="org.eclipse.jetty.server.handler. ↩
                                              ResourceHandler">
                                               <Set name="directoriesListed">true</Set>
                                               <Set name="resourceBase">/Users/ibrahim/ ↩
                                                   jcgexamples/jcgresources</Set>
                                          </New>
                                  </Set>
                        </New>
                  </Item>
                   <Item>
                         <New id="Contexts" class="org.eclipse.jetty.server.handler. ↩
                             ContextHandlerCollection"/>
                   </Item>
                   <Item>
                         <New id="DefaultHandler" class="org.eclipse.jetty.server.handler. ↩
                             DefaultHandler"/>
                   </Item>

                 </Array>
                </Set>
          </New>
        </Set>
```

Here, we have set the context root as `/jcg`; enabled directory listing and set the resource base (but this time with an absolute path).

6.8 Conclusion

In this example we have configured Resource Handler for Jetty in order to serve static content. We have provided configuration both Embedded and Standalone modes of Jetty.

Download

You can download the full source code of this example here: **jetty-resourcehandler-example**

Chapter 7

Jetty JMX Example

JMX technology provides a simple, standard way of managing resources such as applications, devices, and services. Jetty itself does not provide a GUI based console for management/monitoring, however ii presents a solid integration with JMX, which enables us to monitor/manage Servers through JMX.

In this post we are going to discuss JMX integration of Jetty. We will start with an Embedded Jetty example. We will first configure our embedded server to be accessible through JMX; thereafter we are going to incorporate Managed Objects in Jetty style. After the embedded example, we are going to show how we can enable JMX in a standalone Jetty Server. During the example, we are going to monitor and administer our Jetty through JConsole.

In Jetty, the main constructs such as handlers and holders are also JMX beans. This makes almost every single piece of Jetty observable or controllable through JMX. In addition this, Jetty enables creation of JMX objects(MBeans) through annotations(which is an extension to standard MBean capabilities).

7.1 Environment

In the example, following environment will be used:

- Java 8 (Java 7 is also OK.)
- Maven 3.x.y
- Eclipse Luna(as the IDE)
- Jetty v9.2.11 (In Embedded Jetty example, we will add Jetty libraries through Maven.)
- JConsole(which is already bundled with your Java)

7.2 JMX with Embedded Jetty

7.2.1 Structure of the Example

In this example, we are going to enable Jetty for an Embedded Jetty Server programmatically. Our embedded server will have a deployed simple application with a simple servlet. Thereafter we are going to implement Managed Object with Jetty annotations. The Maven project will be packaged as a WAR; so that it can be deployed also on a standalone server.

7.2.2 Creating the Maven Project

We will create the Maven project in Eclipse, applying the steps below:

- Go to File → New →Other → Maven Project
- Tick Create a simple project and press “Next”.
- Enter groupId as : com.javacodegeeks.snippets.enterprise
- Enter artifactId as : jetty-jmx-example
- Select packaging as “war”.
- Press “Finish”.

After creating our project, we are going to add following dependencies to our `pom.xml`.

- `org.eclipse.jetty:jetty-server`
- `org.eclipse.jetty:jetty-webapp`
- `org.eclipse.jetty:jetty-jmx`

The first two dependencies are common for almost all embedded Jetty applications. The third one(`jetty-jmx`) enables us to integrate Jetty with JMX. After adding the dependencies, the dependency section of our `pom.xml` seems as follows:

```
<dependencies>
        <!--Jetty dependencies start here -->
        <dependency>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-server</artifactId>
                <version>9.2.11.v20150529</version>
        </dependency>


        <dependency>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-webapp</artifactId>
                <version>9.2.11.v20150529</version>
        </dependency>

        <dependency>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-jmx</artifactId>
                <version>9.2.11.v20150529</version>
        </dependency>

        <!--Jetty dependencies end here -->

</dependencies>
```

7.2.3 Enabling JMX Programmatically

In order to keep things simple, we are going to implement our Jetty Server through our Main class of the project. You can see the `JettyJmxExampleMain` class below, decorated with source code comments.

JettyJmxExampleMain.java

```
package com.javacodegeeks.snippets.enterprise.jettyjmx;

import java.lang.management.ManagementFactory;

import org.eclipse.jetty.jmx.MBeanContainer;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.util.log.Log;
import org.eclipse.jetty.webapp.WebAppContext;

public class JettyJmxExampleMain {

        public static void main(String[] args) throws Exception {

                // 1. Creating the server on port 8080
                Server server = new Server(8080);

                // 2. Creating the WebAppContext for the created content
                WebAppContext ctx = new WebAppContext();
                ctx.setResourceBase("src/main/webapp");
                server.setHandler(ctx);

                // 3. CreatingManaged Managed Bean container
                MBeanContainer mbContainer = new MBeanContainer(ManagementFactory. ↩
                    getPlatformMBeanServer());

                // 4. Adding Managed Bean container to the server as an Event Listener  and ↩
                     Bean
                server.addEventListener(mbContainer);
                server.addBean(mbContainer);

                // 5. Adding Log
                server.addBean(Log.getLog());
                // 6. Starting the Server
                server.start();
                server.join();

        }
}
```

In the first steps (1 and 2), we initialize a Jetty Server with a Web Application context under `src/main/resources/webapp`. In this part, nothing is special in terms of JMX integration. The web application in this example consists of a trivial Servlet, details of which will be provided later.

In Step 3, we create our Managed Bean container. This container holds reference to the JMX Managed objects. In step 4, we attach this container to our Server. In the later steps (5 and 6), we add logging capability and start our server.

As mentioned above, the web application we deployed on our embedded Server is simple. It consists of a single servlet (`JCGServlet`) that increments a counter on each request. The counter is encapsulated in a singleton object. The content of the `web.xml`, `JCGServlet` and `CounterSingleton` are presented below:

web.xml

```
<web-app xmlns="https://java.sun.com/xml/ns/javaee" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://java.sun.com/xml/ns/javaee
              https://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
        version="3.0">

        <display-name>Jetty JMX Configuration Example</display-name>


        <servlet>
```

```
                <servlet-name>JCGServlet</servlet-name>
                <servlet-class>com.javacodegeeks.snippets.enterprise.jettyjmx.JCGServlet</ ↩
                    servlet-class>
        </servlet>

        <servlet-mapping>
                <servlet-name>JCGServlet</servlet-name>
                <url-pattern>/jcg/*</url-pattern>
        </servlet-mapping>


</web-app>
```

JCGServlet.java

```
package com.javacodegeeks.snippets.enterprise.jettyjmx;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServlet;


public class JCGServlet extends HttpServlet {


        @Override
        public void service(ServletRequest req, ServletResponse res) throws  ↩
            ServletException, IOException {

                CounterSingleton.getInstance().increment();

                res.getOutputStream().print("Application Specific Servlet Response");

        }

}
```

CounterSingleton.java

```
package com.javacodegeeks.snippets.enterprise.jettyjmx;

public class CounterSingleton {

        private static CounterSingleton instance = new CounterSingleton();

        private Integer counter = 0;

        private CounterSingleton() {
                counter = 0;
        }

        public static CounterSingleton getInstance() {
                return instance;
        }

        public synchronized void increment() {
                counter++;
        }
```

```
        public Integer getCounter() {
                return counter;
        }

        public synchronized void reset(){
                counter=0;
        }

}
```

When we start our application, our application is ready to be monitored and managed through JMX. We can verify that our web application and server are running by navigating to https://localhost:8080/jcg with our browser and seeing the response below:

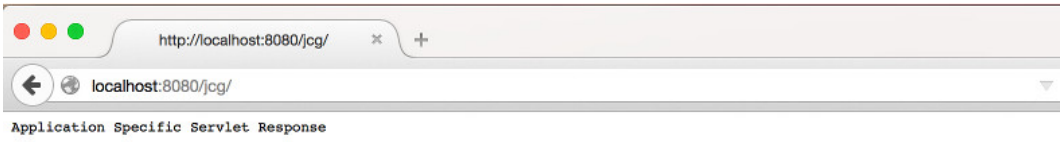

Figure 7.1: Sample Servlet Response

7.2.4 Monitoring with JConsole

We can monitor our JMX enabled embedded server using JConsole, which is available under JAVA_HOME of our system. When we launch JConsole, it shows as a list of available local processes as in the figure below:

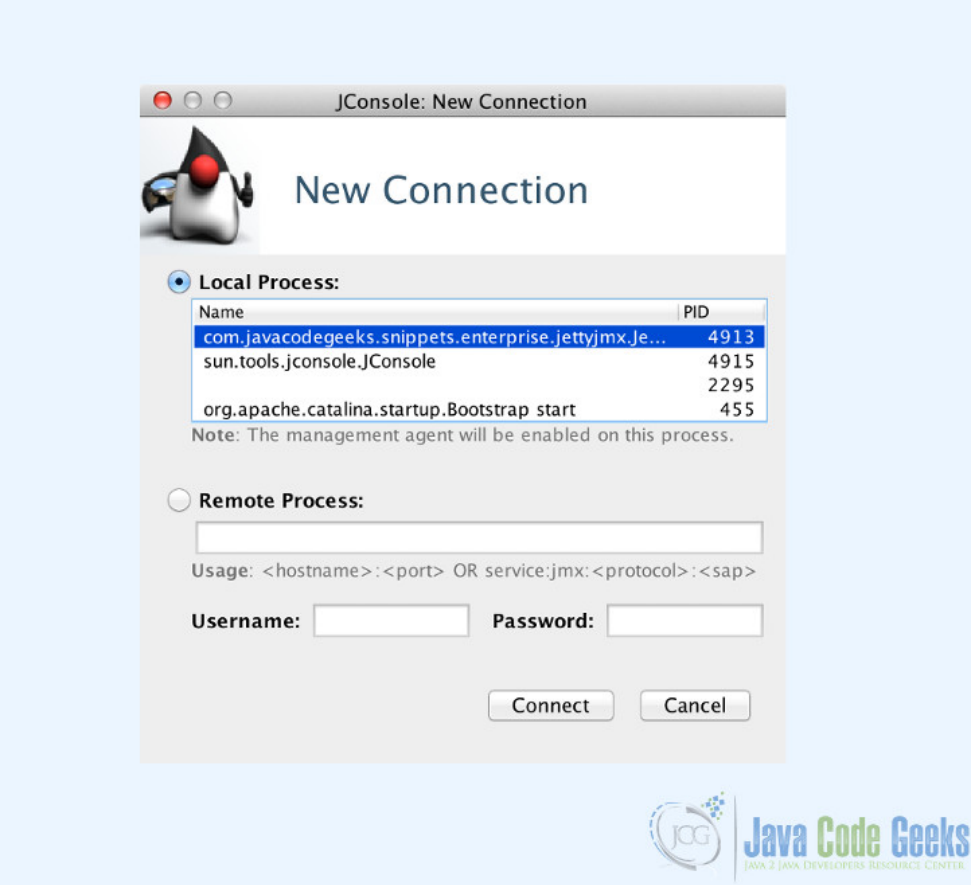

Figure 7.2: JConsole Initial Screen

Here our embedded server is listed with name of the main class. When we select this process and proceed, we can see various parameters (memory,CPU, thread utilization etc) related to our Jetty. The screen presents 6 tabs for JMX administration. When we select MBean tab, the available Managed Beans are listed in a tree, which can be viewed below:

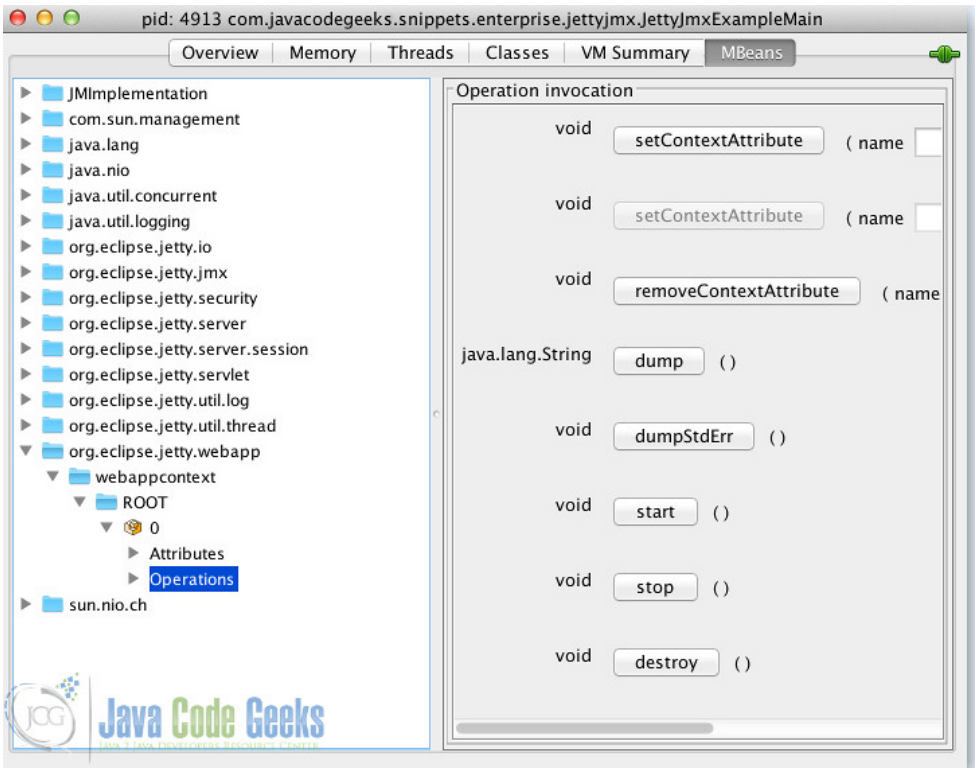

Figure 7.3: Managed Bean Tree

We can expand the tree `org.eclipse.jetty.webapp->webappcontext->ROOT->0`. This node shows a list parameters to be monitored under `Attributes` and a set of operations that can be invoked under `Operations` sections. Among these operations, we can stop the application invoking `stop()` method. When we call this operation, the webapp will immediately stop and will return 404 error when we try to access. We can restart our web application invoking the `start()` method.

In addition to these, JConsole enables us various monitoring and administration options. Forcing a Garbage Collection or setting web application initialization parameters are among those options.

7.2.5 Jetty Managed Objects

As mentioned in the previous sections, Jetty enables us to create our Managed Beans using Jetty annotations. It is worth to mention three annotations here:

- `@ManagedObject`: This annotation is used for annotating managed object classes.
- `@ManagedAttribute`: This annotation denotes the getter fields that are listed under `Attributes` section,
- `@ManagedOperation`: This annotation denotes the methods to be listed under `Operations` section.

Here you can see an example Managed object named `JCGManagedObject`. This class simply returns our previously mentioned counter and provides an operation to reset the counter.

JCGManagedObject.java

```
package com.javacodegeeks.snippets.enterprise.jettyjmx;

import org.eclipse.jetty.util.annotation.ManagedAttribute;
import org.eclipse.jetty.util.annotation.ManagedObject;
import org.eclipse.jetty.util.annotation.ManagedOperation;
```

```
@ManagedObject("jcgManagedObject")
public class JCGManagedObject {


        @ManagedAttribute
        public Integer getCount() {
                return CounterSingleton.getInstance().getCounter();
         }


         @ManagedOperation
         public void reset() {
                CounterSingleton.getInstance().reset();
         }
}
```

Our managed bean can be wired to Jetty through adding the highlighted code below(Line 4) in the main:

```
// 4. Adding Managed Bean container to the server as an Event Listener and Bean
server.addEventListener(mbContainer);
server.addBean(mbContainer);
server.addBean(new JCGManagedObject());
```

Here we have created an instance of our managed object and added as a bean. When we restart our application and open JConsole, we can see our managed bean in the MBeans tab under `com.javacodegeeks.snippets.enterprise.jettyjmx->jcgmanagedobject->0`. Here we can see our counter, which is incremented at each request, as an attribute, and we can reset this counter invoking the `reset()` under the `Operations` section:

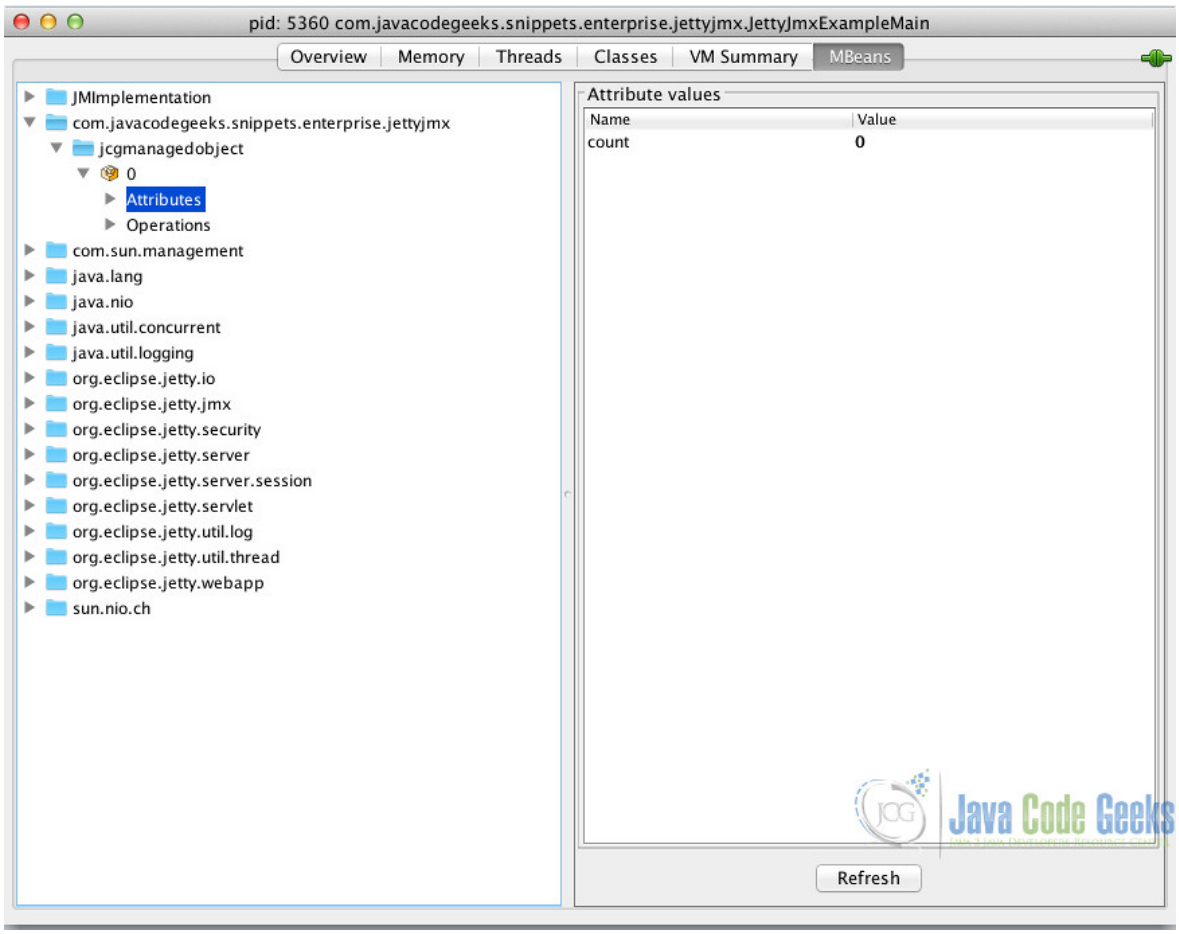

Figure 7.4: JCG Managed Object

7.3 JMX with Standalone Jetty

To this point, we have discussed how we can integrate JMX to embedded Jetty, now we are going to enable JMX for standalone mode. Jetty presents a modular architecture which also includes JMX integration a a module. Related configuration is stored under `JETTY_HOME/etc/jetty-jmx.xml`. This configuration is almost equal to our programmatic configuration in the embedded mode. All we have to do is enabling jmx module. The related steps are as simple as below:

- Open start.ini under JETTY_HOME
- Add this line: `--module=jmx-remote`
- Save and close the file.

When we run our standalone Jetty, Jetty will start with JMX enabled. We can access our server through JConsole and manage it as in the embedded mode.

7.4 Conclusion

Jetty provides powerful administration and monitoring capabilities through JMX. In this example, we have skimmed through Jetty JMX integration for embedded and standalone modes. In addition to this, we have created a Managed Object which is implemented in Jetty style.

Download

You can download the full source code of this example here: **Jetty JMX Example**

Chapter 8

Jetty OSGi Example

The OSGi specification defines a modularization and component model for Java applications. Jetty leverages OSGi support providing an infrastructure that enables developers to deploy Jetty and web applications inside an OSGi container. One can deploy traditional Java Web Applications or Context Handlers on Jetty within the OSGi container; in addition to this, OSGi bundles can be deployed as web applications.

In this example, we are going to show how we can deploy Web Applications on Jetty within an OSGi container. We are going to enable a Jetty Server on an OSGi container first, thereafter we are going to deploy a Servlet on our OSGi powered Jetty.

8.1 Environment and Prerequisites

In this example, we are going to use the following programming environment:

- Java 8 (Java 7 is also OK for this example)
- Eclipse for RCP and RAP Developers v4.5 (Mars)
- Equinox 3.10 OSGi implementation (v 3.9 is also fine) configured in Eclipse
- Jetty v9.2.11(We do not necessarily need Jetty installation, however having one will be handy)

At this point, we are not going to detail Equinox configuration in Eclipse, which would be beyond scope of this example. We assume that it is already configured.

8.2 Adding Jetty dependencies to OSGi Target

8.2.1 Jetty libraries

Jetty JAR files, which happen to exist under the `lib` folder of the Jetty installation contain appropriate manifest entries (`MANIFEST.MF`) for OSGi deployment. All we have to do is, to copy the necessary JARs under our OSGi target.

The necessary libraries are as follows:

- jetty-util
- jetty-http
- jetty-io
- jetty-security

- jetty-server
- jetty-servlet
- jetty-webapp
- jetty-deploy
- jetty-xml
- jetty-osgi-servlet-api

We have to place these libraries in a location that our OSGi container is aware of. We can wither copy to an existing location, or create a new location. In this example, we have copied to an existing OSGi target location.

8.2.2 jetty-osgi-boot Bundle

After copying Jetty dependencies, we have to add the jetty-osgi-boot bundle to the OSGi target. jetty-osgi-boot is the bundle that performs the initialization of the Jetty server. This bundle is not included in the Jetty installation, but can be easily obtained from Maven Central Repository.

Once we have downloaded the bundle, we should copy it to the OSGi target location.

8.2.3 Reloading OSGi Target

After we have copied the Jetty libs and boot bundle, we have to refresh our container in Eclipse. This can be performed following the steps below:

- Open Eclipse `Preferences` from the Menu
- Search for `Target` from the search box on the top left.
- Select your OSGi target
- Press `Reload`.

8.3 Running the Jetty Server on the OSGi container

jetty-osgi-boot bundle provides two options for the server initialization one of which must be configured:

- Setting jetty.home.bundle
- Setting jetty home

The first option stipulates that Jetty runs with the predefined XML files coming with the bundle JAR. The second option requires setting a Jetty home with the necessary configuration files. In this example, we will take the second option.

This can be accomplished as follows:

- Create a folder named `osgi-jetty-home` (You can name is as you wish.)
- Create the folder `osgi-jetty-home/etc`
- Include `jetty.xml, jetty-selector.xml and jetty-deploy.xml` files under `osgi-jetty-home/etc`. (Alternatively, you can copy from `jetty-osgi-boot` JAR or jetty installation)
- Add the following JVM parameter to run configuration of your OSGi container: `-Djetty.home=/path/to/your/osgi-jetty-home`

When you run the OSGi container, you will see that Jetty has started on port 8080. You can check via your browser navigating to https://localhost:8080.

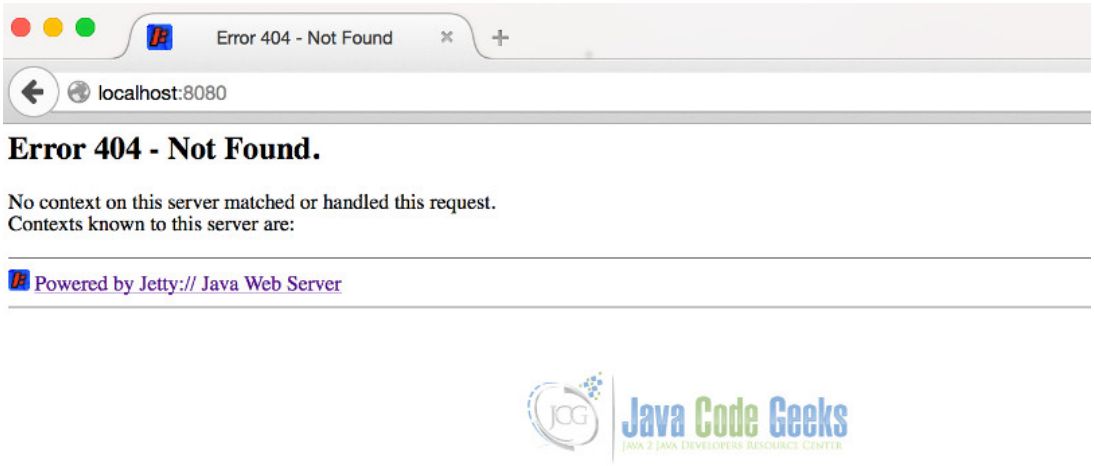

Figure 8.1: OSGi powered Jetty

8.4 Deploying a Servlet on the OSGi Jetty

In this part, we will show how to deploy a simple servlet on the Jetty which runs in our OSGi container. The example can be extended to include web apps, resource handlers or other configuration.

In this example we are going to create a simple OSGi bundle, in the activator of which, we will configure a Servlet and register its handler as an OSGi component.

8.4.1 Creating the Eclipse Project

We start with creating the Eclipse project. The steps needed are as follows:

- Click on `File-->New->Plug-in Project`.
- Type project name as `jetty-osgi-example`.
- Select `an OSGi framework` as the target platform.
- Press `Next`.
- Check the option: "Generate an Activator...".
- Press `Finish`.

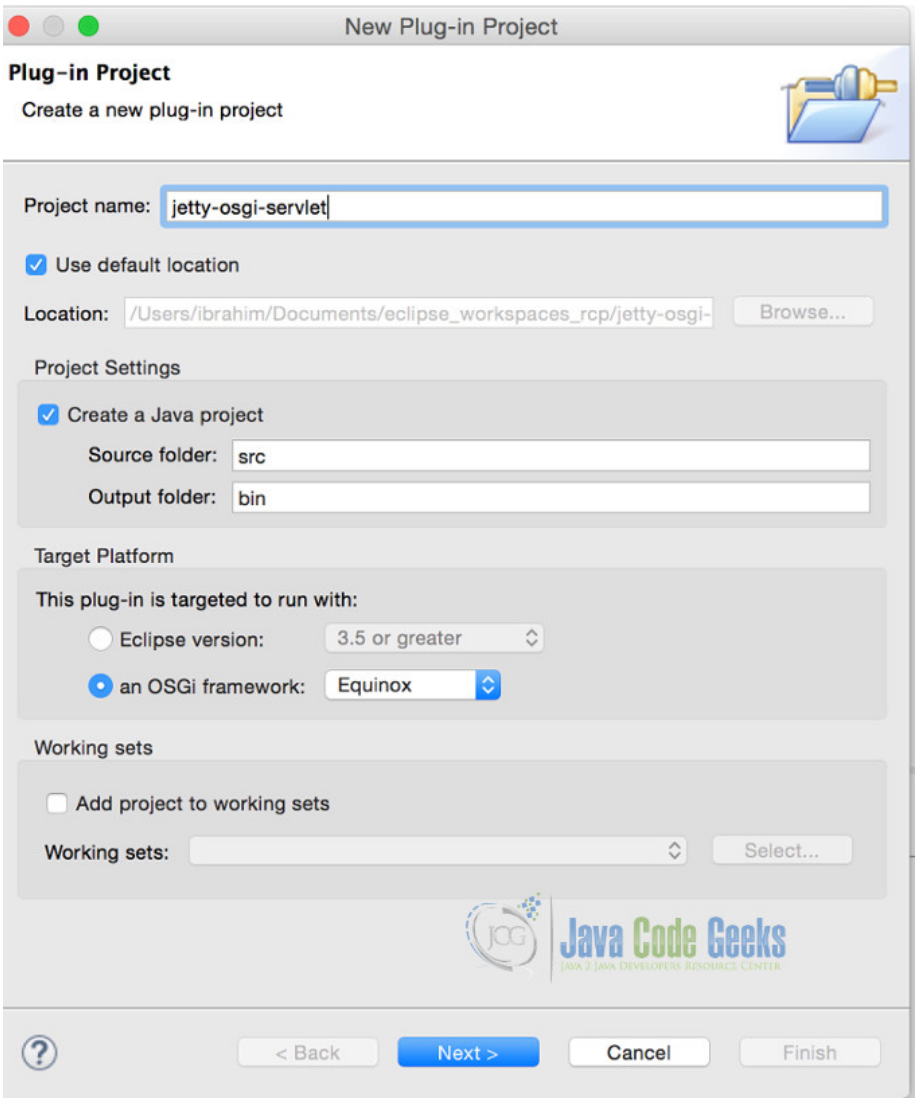

Figure 8.2: Creating the Eclipse project

8.4.2 Adding Required Plugins

After we have created our project, we have to add Jetty dependencies as Required Plugins in the `MANIFEST.MF` file. We can do it through Eclipse as follows:

- Open `META-INF/MANIFEST.MF` file with Eclipse Editor
- On the `Dependencies`, click on “Add” button on the `Required Plug-ins`" section.
- Type `Jetty` in the search box and add all the Jetty plugins that are available in the OSGi container.
- Press `OK`.

Now, the Jetty dependencies are ready. Our `MANIFEST.MF` file looks like:

MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Jetty-osgi-example
```

```
Bundle-SymbolicName: jetty-osgi-example
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: jetty_osgi_example.Activator
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Import-Package: org.osgi.framework;version="1.3.0"
Bundle-ActivationPolicy: lazy
Require-Bundle: org.eclipse.jetty.server;bundle-version="9.2.11",
 org.eclipse.jetty.osgi-servlet-api;bundle-version="3.1.0",
 org.eclipse.jetty.servlet;bundle-version="9.2.11",
 org.eclipse.jetty.deploy;bundle-version="9.2.11",
 org.eclipse.jetty.http;bundle-version="9.2.11",
 org.eclipse.jetty.io;bundle-version="9.2.11",
 org.eclipse.jetty.osgi.boot;bundle-version="9.2.11",
 org.eclipse.jetty.security;bundle-version="9.2.11",
 org.eclipse.jetty.util;bundle-version="9.2.11",
 org.eclipse.jetty.webapp;bundle-version="9.2.11",
 org.eclipse.jetty.xml;bundle-version="9.2.11"
```

8.4.3 Wiring our Servlet to OSGI and Jetty

After setting the dependencies, we are going deploy a simple Servlet on our OSGi powered Jetty. Our Servlet is named as `JcgServlet` and very simple as follows:

JcgServlet.java

```
package jetty_osgi_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 JcgServlet extends HttpServlet{

        @Override
        protected void service(HttpServletRequest req, HttpServletResponse resp) throws  ↩
            ServletException, IOException {
                resp.getWriter().println("Hello JCG, Hello OSGi");
        }

        public JcgServlet() {
                super();
        }
}
```

Now we are going to wire this Servlet to our Jetty. As you remember, while creating the Eclipse Project, we had checked the option `Generate an Activator...`. This selection creates a class `jetty_osgi_example.Activator`. In this class, we can register our components to OSGi once the bundle is activated. Now we are going to register our Servlet Handler, so that it will be available to the Jetty.

We are going to implement the `activate()` method of the Activator. Below you can see the `Activator` class decorated with source code comments.

Activator.java

```
package jetty_osgi_example;

import java.util.Hashtable;
```

```
import org.eclipse.jetty.server.handler.ContextHandler;
import org.eclipse.jetty.servlet.ServletContextHandler;
import org.eclipse.jetty.servlet.ServletHandler;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

        private static BundleContext context;

        static BundleContext getContext() {
                return context;
        }


        public void start(BundleContext bundleContext) throws Exception {
                Activator.context = bundleContext;

                //1. We create a Servlet Handler
                ServletHandler handler = new ServletHandler();

                //2. We register our Servlet and its URL mapping
                handler.addServletWithMapping(JcgServlet.class, "/*");

                //3. We are creating a Servlet Context handler
                ServletContextHandler ch= new ServletContextHandler();

                //4. We are defining the context path
                ch.setContextPath("/jcgservletpath");

                //5. We are attaching our servlet handler
                ch.setServletHandler(handler);

                //6. We are creating an empty Hashtable as the properties
                Hashtable props = new Hashtable();

                // 7. Here we register the ServletContextHandler as the OSGi service
                bundleContext.registerService(ContextHandler.class.getName(), ch, props);

                System.out.println("Registration Complete");
        }


        public void stop(BundleContext bundleContext) throws Exception {
                Activator.context = null;
        }

}
```

In the activator, we have first created a `ServletHandler` and registered our `Servlet` along with a mapping. Thereafter we appended it to a `ServletContextHandler` with a context path. Lastly we have registered our `ServletContextHandler` as an OSGi component. Now our Jetty Server will find our `ServletContextHandler` as its context handler.

Please note that, the the components are resolved by name, therefore the component name `ContextHandler.class.getName()` shouldn't be replaced with an arbitrary name.

After we implemented our bundle, we can run our OSGi container. when we try to access https://localhost:8080/jcgservletpath/, we will see that our request is handled by our Servlet with the following response:

Figure 8.3: Servlet Response

Now we have a Servlet running on the Jetty within an OSGI container. This example can be tried with different Handler and web application configurations. Jetty promises full support for OSGi.

8.5 Conclusion

Jetty provides full support for OSGi containers in order to leverage modularity. In this example, we have deployed Jetty in an OSGi container thereafter we have deployed a Servlet on this Jetty, in which we have defined our `ServletContextHandler` as an OSGi service.

Download

You can download the full source code of this example here : **Jetty OSGi Example**

Chapter 9

Jetty JSP Example

JSP (JavaServer Pages) which is core part of Java EE, enables developers to create dynamic web content based on the Java Servlet technology. In this example, we are going to enable Jetty for JSP. We will start with Embedded mode of Jetty. We are going to initialize our embedded Jetty to run JSP pages. Thereafter we will continue with standalone mode and shortly mention the JSP configuration in standalone mode.

Jetty supports two JSP Engine implementations: Apache Jasper and Glassfish Jasper. Starting from Jetty version 9.2, the default and favored implementation is Apache Jasper. In this example we are going to use this one; however we will show how we can switch to Glassfish implementation in the standalone mode.

At this point, we have to mention that, this example should not be considered as a JSP tutorial but a demonstration of JSP on Jetty container.

9.1 Environment

In the example, following environment will be used:

- Java 8 (Java 7 is also OK)
- Maven 3.x.y
- Eclipse Luna(as the IDE)
- Jetty v9.2.11 (In Embedded Jetty example, we will add Jetty libraries through Maven)

9.2 JSP with Embedded Jetty

9.2.1 Structure of the Example

In this example, we are going enable JSP in an Embedded Jetty. We are going to implement a very simple JSP page which will demonstrate JSP and JSTL capabilities. We are going to package this application as a WAR file; so we will be able to drop and run it in a standalone Jetty.

9.2.2 Creating the Maven Project in Eclipse

We will create the Maven project in Eclipse, applying the steps below:

- Go to File → New →Other → Maven Project

- Tick Create a simple project and press “Next”.
- Enter groupId as : com.javacodegeeks.snippets.enterprise
- Enter artifactId as : jetty-jsp-example
- Select packaging as “war”.
- Press “Finish”.

After creating our project, we are going to add following dependencies to our `pom.xml`:

- org.eclipse.jetty:jetty-server
- org.eclipse.jetty:jetty-webapp
- org.eclipse.jetty:jetty-annotations
- org.eclipse.jetty:apache-jsp
- jstl:jstl

The first dependency `(jetty-server)` is the core Jetty dependency. `jetty-webapp` is needed for creating Jetty web application context. `jetty-annotations` dependency can be viewed as a utility, which makes JSP initialization easier. `apache-jsp` dependency is the Apache implementation of JSP and finally `jstl` is the JSP standard tag library(version 1.2).

After adding the necessary dependencies, our pom.xml looks like:

```
<dependencies>
        <!--Jetty dependencies start here -->
        <dependency>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-server</artifactId>
                <version>9.2.11.v20150529</version>
        </dependency>

        <dependency>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-webapp</artifactId>
                <version>9.2.11.v20150529</version>

        </dependency>
        <dependency>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-annotations</artifactId>
                <version>9.2.11.v20150529</version>
        </dependency>
        <!-- Jetty Dependencies end here -->

        <!--Jetty Apache JSP dependency  -->
        <dependency>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>apache-jsp</artifactId>
                <version>9.2.11.v20150529</version>

        </dependency>

        <!-- JSTL Dependency -->

        <dependency>
                <groupId>jstl</groupId>
                <artifactId>jstl</artifactId>
                <version>1.2</version>
        </dependency>
</dependencies>
```

9.2.3 Configuring the Web Application

As mentioned above, we are going to configure a very simple JSP application which will demonstrate both JSP and JSTL capabilities. The steps needed are described below:

- Create the folder `src/main/webapp` under your project directory(if not exists).
- Create `WEB-INF` directory under `src/main/webapp` (if not exists).
- Create `web.xml` under `src/main/webapp/WEB-INF`.
- Create `example.jsp` under `src/main/webapp`.

The content of the web.xml to enable JSP can be viewed below:

```
<web-app xmlns="https://java.sun.com/xml/ns/javaee" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://java.sun.com/xml/ns/javaee https://java.sun.com/xml/ns/ ↩
            javaee/web-app_2_5.xsd"
        version="2.5">
        <display-name>JSP Example Application</display-name>

        <servlet id="jsp">
                <servlet-name>uu</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>*.jspf</url-pattern>
                <url-pattern>*.jspx</url-pattern>
                <url-pattern>*.xsp</url-pattern>
                <url-pattern>*.JSP</url-pattern>
                <url-pattern>*.JSPF</url-pattern>
                <url-pattern>*.JSPX</url-pattern>
                <url-pattern>*.XSP</url-pattern>
        </servlet-mapping>
</web-app>
```

`example.jsp` is a simple JSP file which shows current date and outputs a literal text which is a JSTL expression. The content of the JSP file is as follows:

example.jsp

```
<%@page import="java.util.ArrayList"%>

<html>
<head>
```

```
<title>Java Code Geeks Snippets - Sample JSP Page</title>
<meta>
<%@ taglib uri="https://java.sun.com/jsp/jstl/core" prefix="c"%>
</meta>
</head>

<body>
        <c:out value="Jetty JSP Example"></c:out>
        <br />
        Current date is: <%=new java.util.Date()%>
</body>
</html>
```

9.2.4 Enabling JSP programmatically

In this part, we are going to start an embedded Jetty server with the simple web application that we have configured in the previous section and thereafter we will enable JSP for our server. In order to keep things simple, we are going to implement our Jetty Server through our Main class of the project. You can see the `JettyJspExampleMain` class below, decorated with source code comments.

JettyJspExampleMain.java

```
package com.javacodegeeks.snippets.enterprise.jettyjsp;

import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.webapp.WebAppContext;

public class JettyJspExampleMain {

        public static void main(String[] args) throws Exception {

                // 1. Creating the server on port 8080
                Server server = new Server(8080);

                // 2. Creating the WebAppContext for the created content
                WebAppContext ctx = new WebAppContext();
                ctx.setResourceBase("src/main/webapp");
                ctx.setContextPath("/jetty-jsp-example");

                //3. Including the JSTL jars for the webapp.
                ctx.setAttribute("org.eclipse.jetty.server.webapp. ↩
                    ContainerIncludeJarPattern",".*/[^/]*jstl.*\\.jar$");

                //4. Enabling the Annotation based configuration
                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.EnvConfiguration", "org.eclipse.jetty.plus.webapp. ↩
            PlusConfiguration");
        classlist.addBefore("org.eclipse.jetty.webapp.JettyWebXmlConfiguration", "org. ↩
            eclipse.jetty.annotations.AnnotationConfiguration");

        //5. Setting the handler and starting the Server
                server.setHandler(ctx);
                server.start();
                server.join();

        }
}
```

- First we initialize an embedded Server on port 8080.
- Then we initialize the web application context.
- In Step 3, we include jstl jar for our web application. If we skip this step, we will not be able to use JSTL tags in our JSP pages.
- In step 4, we enable annotation based configuration for our server. This part of the code looks a bit like magical snippet, which seems irrelevant with JSP configuration; however these three lines is the most crucial part for JSP configuration. When annotation configuration is enabled, JSP implementation is automatically discovered and injected to the server. Otherwise we would have to implement it manually.
- Step 5 includes the snippets for setting the context handler and starting the server.

9.2.5 Running the Application

When we run the application, our embedded server will start on port 8080. If we try to access `https://localhost:8080/jetty-jsp-example/example.jsp` we can see our simple JSP page:

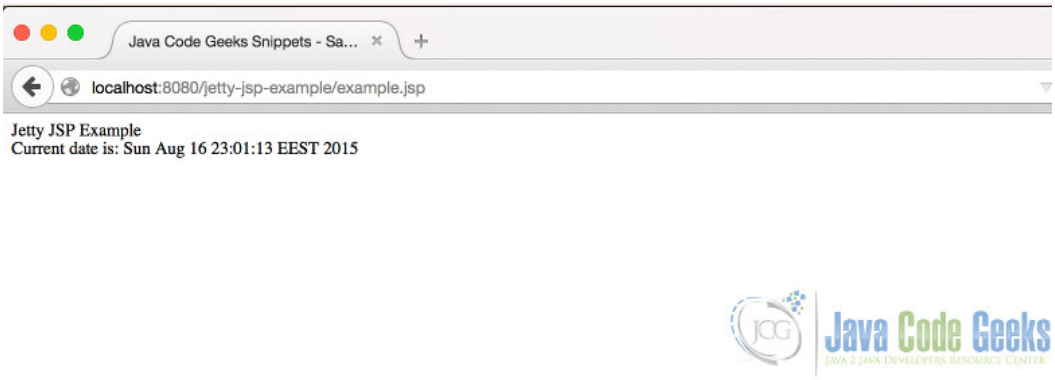

Figure 9.1: Output of example.jsp

In our webpage JSP, "Jetty JSP Example" text comes from a JSTL expression whereas current date is an outcome of a core JSP expression.

9.3 JSP in Standalone Jetty

In the previous sections, we have discussed how to enable JSP on an Embedded Jetty. In the standalone mode, it is very easy to run JSP. In standalone mode, JSP is enabled by default. All we have to do is, dropping the JSP web application WAR in the webapps directory of Jetty.

Jetty has a `jsp` module which is enabled by default. You can disable it via the `start.ini` file under `JETTY_HOME` removing the following line:

```
--module=jsp
```

`start.ini` file has a line that sets `Apache` as the default JSP implementation:

```
jsp-impl=apache
```

If we want to use Glassfish implementation for some reason, then we have to alter this line to:

```
jsp-impl=glassfish
```

9.4 Conclusion

In this example, we have discussed how we can configure Jetty for JSP. We have first demonstrated configuration for Embedded Jetty with a simple JSP application, thereafter we have briefly mentioned how JSP is configured for the standalone mode.

Download

You can download the full source code of this example here: **jetty-jsp-example**