OSGi and GWT integration via Pax-web and Maven (v:0.01)

Sergio Blanco Diez   Ignacio Diaz de Sarralde Carvajal

January 13, 2010

Contents

1 Introduction **2**

2 GWT **3**

2.1 Toolkit overview . . . 3

2.1.1 Development overview . . . 3

2.1.2 Google Web Toolkit component overview . . . 4

2.2 Workflow . . . 4

2.2.1 Project creation . . . 4

2.2.2 Project structure . . . 5

2.2.3 “Hosted” or Development server . . . 6

2.2.4 Module definition; .gwt.xml files . . . 8

2.2.5 Coding the client; GWT Widgets . . . 9

2.2.6 Coding the server; Remote Procedure Calls . . . 12

2.3 Other features . . . 14

2.3.1 Internationalization . . . 14

2.3.2 Declarative User Interfaces . . . 14

3 OSGi and GWT integration **17**

3.1 Creating the GWT Project . . . 17

3.2 Things to consider in GWT Maven projects . . . 20

3.3 Enabling OSGi in the GWT project . . . 20

3.3.1 Turning the GWT project into a bundle . . . 20

3.3.2 Dependencies management . . . 25

3.3.3 Run using Pax Runner . . . 28

Chapter 1

Introduction

When developing complex OSGi applications it is usual to implement one or more web frontends, be them for end users or for consumption by other systems. OSGi provides a standard way to register servlets and resources under different URLs, but creating complex frontends can be difficult and time consuming. When the frontend is SOA based, solutions like Apache CXF help exposing internal services via web services or REST. But if the frontend must be a fairly complex web application in itself, complexity rises. It is usual in those cases to expose the system's functionality as web services or RESTful services and create a separate web application that consumes those services. This solution needs independent server software running, more configuration, more complex boot scripts...

Another solution is to enable war package deployment in OSGi. Just by deploying some bundles OSGi can work as a lightweight application container, enabling quick and easy integration of the frontend and the OSGi system. There are several web application frameworks in Java (Struts, OpenLaszlo, Spring, GWT, AppFuse, Eclipse RAP...), and even though the ideas of this document are usable for most of them (at least the ones that deploy as WARs in Apache Tomcat), this document will detail GWT 2.0 integration for it's complete, flexible and developer-friendly nature. To enable integration, Pax Web bundles will be used.

Sergio Blanco (sergio.blanco@deusto.es) Ignacio Diaz de Sarralde (isarralde@deusto.es)

Chapter 2

GWT

Google Web Toolkit[6] is a suite of tools devised for quick development of complex browser-based applications. Its goal is to enable the creation of optimized web applications without forcing the developer to understand the inner working of Javascript, AJAX, and browser specific quirks.

2.1 Toolkit overview

2.1.1 Development overview

What makes Google Web Toolkit development cycle unique is its Java to Javascript compiler. The developer codes both the presentation tier (the GUI) and the application tier (the inner workings of the application) in Java; and the compiler generates a fully web server-deployable application with fully optimized Javascript code.

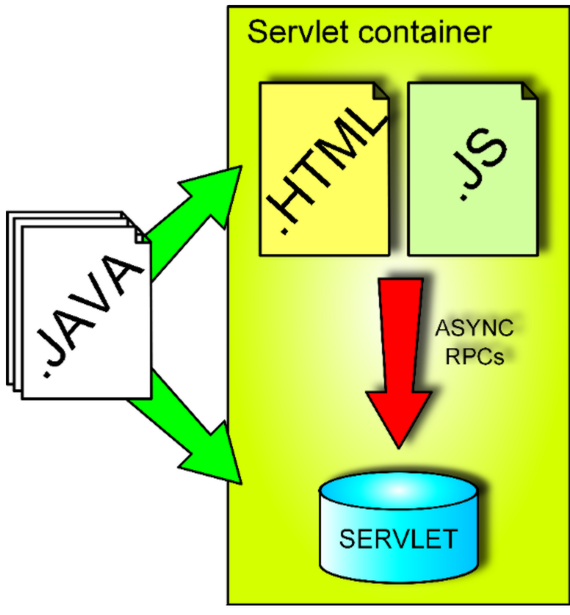

Figure 2.1: Google Web Toolkit compilation process.

Graphical User Interface design is done by composition of components called "widgets", not unlike traditional Swing Java desktop applications. This makes controlling events, input validation, and user interaction far easier than the traditional methods of web development.

2.1.2 Google Web Toolkit component overview

Google Web Toolkit includes a series of tools to allow for easier development of web applications:

Software Development Kit

The Google Web Toolkit SDK features a complete Java API Library; the Javascript compiler; and development or hosted mode server, complete with the Google Web Toolkit Developer Plugin for your web browser of choice.

Speed Tracer

A extension for the popular Chrome [4] web browser that allows for performance analysis in Google Web Toolkit-based applications.

Eclipse IDE plugin

A plugin for the Eclipse[3] Integrated Development Environment that adds specific support for Google Web Toolkit project creation, debugging, and deployment.

2.2 Workflow

Typical Google Web Toolkit (from now on, GWT) workflow follows the steps detailed here:

2.2.1 Project creation

GWT features a command line utility called `webAppCreator` that automatically generates all required files required to begin coding a new browser-based application development, in the shape of a simple but fully working web application. It also generates extra files such as Eclipse IDE[3] project files, Apache Ant[1] scripts and a complete .war file system for deployment including GWT libraries.

Application generation example

Let's create a new web application called "GWTDemo" by running the command

```
webAppCreator -out GWTDemo com.gwt.example
```

Listing 2.1: GWT project creation example

The `-out` parameter is both the application name and the folder to generate corresponding files. The second parameter establishes the root package of the application.

2.2.2 Project structure

Typical GWT projects should follow a fixed file and folder structure.

src:

Project source code in Java.

war:

Static project resources (images, audio, static files...) as well as compilation results.

test:

JUnit[7] test code.

Source code of Java classes should also follow the following packaging convention:

Root package:

Project module definitions (.gwt.xml), static resources to be loaded programatically.

Client package:

Client code. This will be compiled to Javascript.

Server package:

Server code. This is optional code that will be compiled to standard Java Servlets.

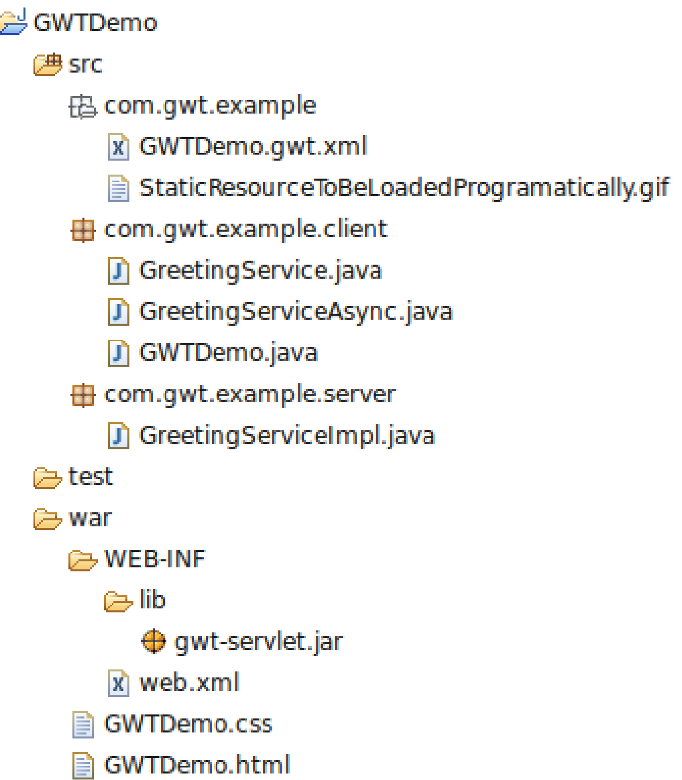

Figure 2.2: "GWTDemo" project structure.

2.2.3 "Hosted" or Development server

GWT includes a special web server designed for quick code testing; Combined with the Google Web Toolkit Developer Plugin for your web browser of choice, it also allows for advanced application debugging.

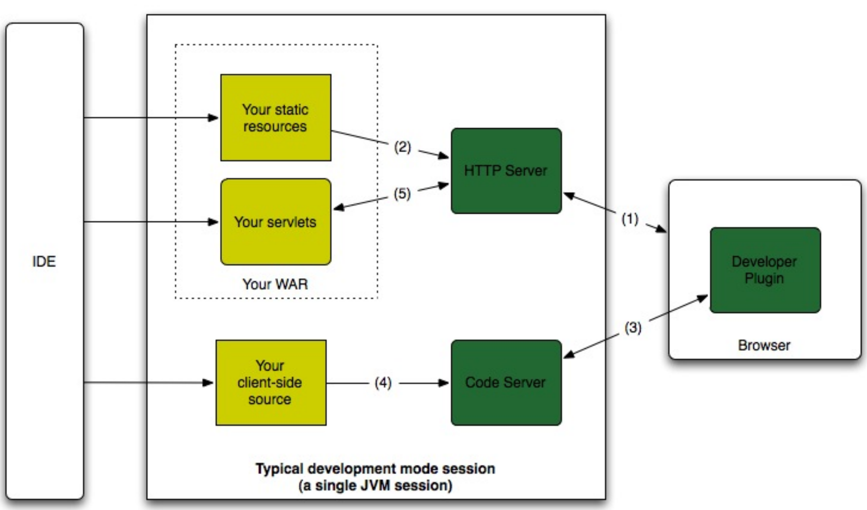

Figure 2.3: Typical development mode session.[6]

To run your application in the development server, you can either launch it using the Eclipse IDE[3] plugin or the `ant devmode` Apache Ant[1] task.

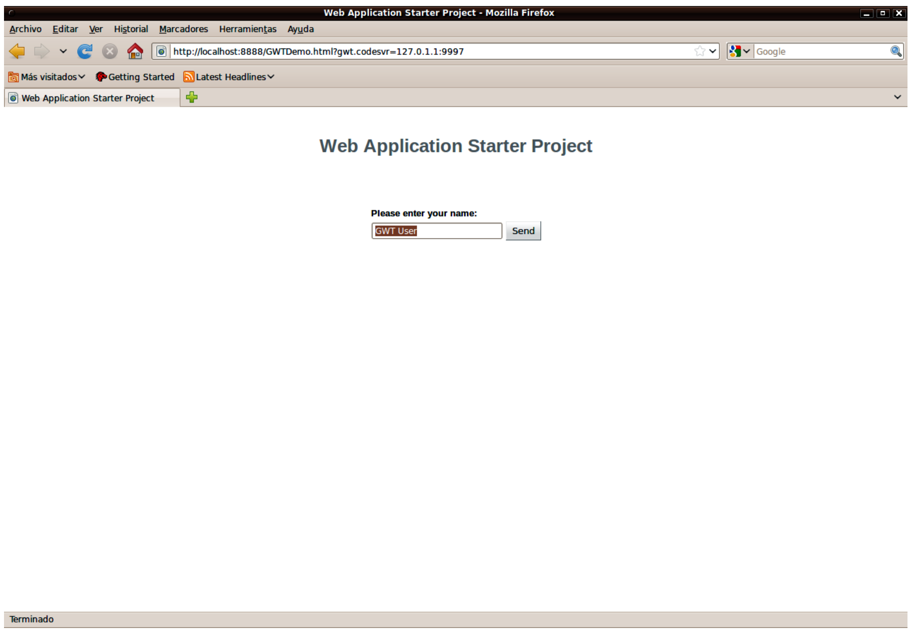

Figure 2.4: "GWTDemo" running in the Firefox[8] web browser in development mode

The hosted mode server shows all activity in the web application, including client-server remoting, exceptions, or user defined strings, in an easy to browse log.

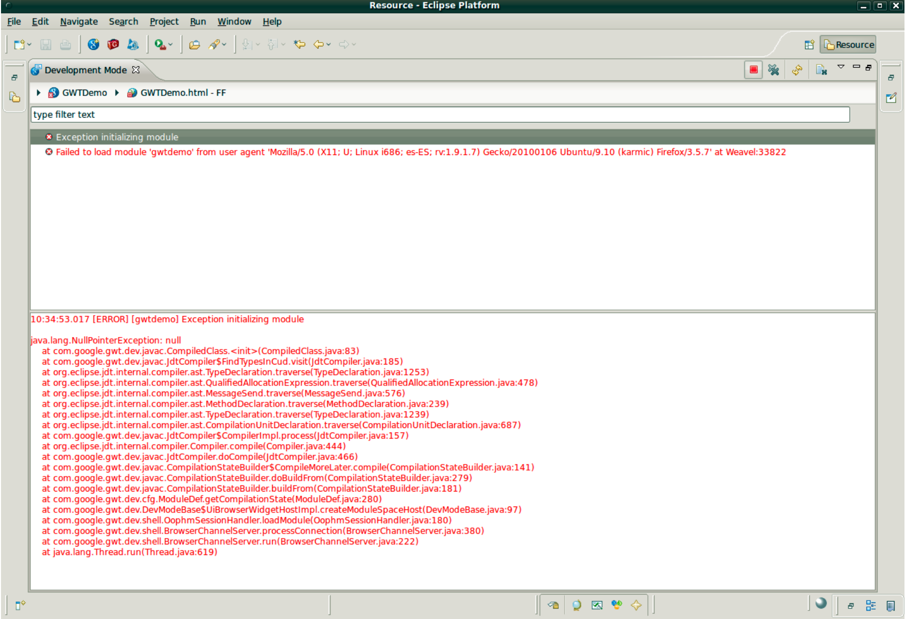

Figure 2.5: Crashing "GWTDemo" in the Eclipse IDE[3]

```
import com.google.gwt.core.client.GWT;
```

```
...

button.addClickHandler(new ClickHandler() {
   public void onClick(ClickEvent event) {
     GWT.log("User Pressed a button.", null); // Added debugging message
     if (label.getText().equals(""))
       label.setText("Hello World!");
     else
       label.setText("");
   }
 });
```

Listing 2.2: Adding a debugging message to the hosted mode log.

The development server also enables you to debug GWT applications with ease. Instead of the traditional web application debug approach, GWT allows you to work with your Java client code, run it step by step, and set breakpoints.

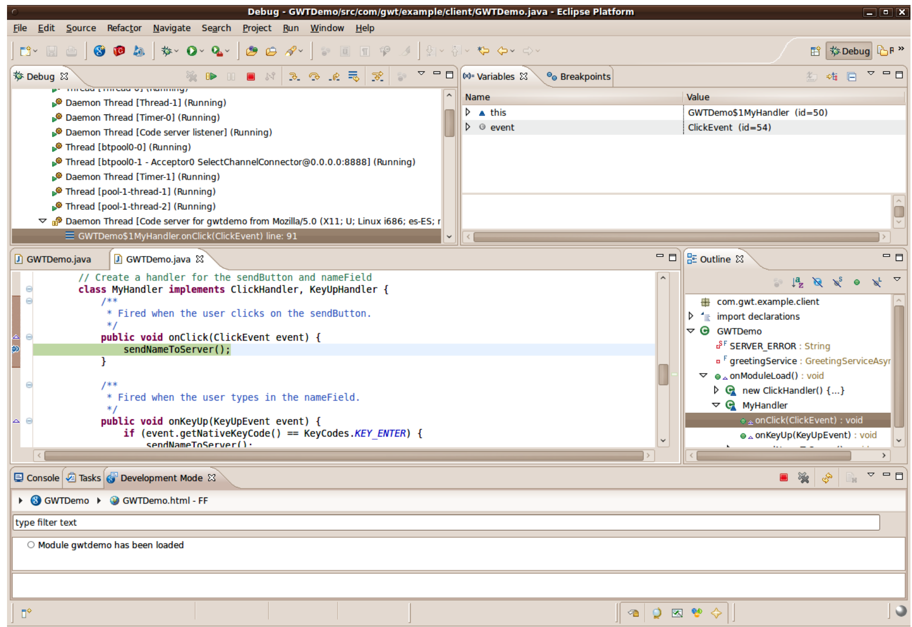

Figure 2.6: Debugging "GWTDemo" in the Eclipse IDE[3]

2.2.4 Module definition; .gwt.xml files

Each GWT application must include a GWT module definition `.gwt.xml` file in the root package. This file defines certain settings of the web application, including external modules imported, application mail class or entry point, path of the source to be compiled to Javascript.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='gwtdemo'>
  <!-- Inherit the core Web Toolkit stuff.                  -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet. You can change  -->
  <!-- the theme of your GWT application by uncommenting    -->
```

```
  <!-- any one of the following lines.                        -->
  <inherits name='com.google.gwt.user.theme.standard.Standard'/>
  <!-- <inherits name='com.google.gwt.user.theme.chrome.Chrome'/> -->
  <!-- <inherits name='com.google.gwt.user.theme.dark.Dark'/> -->

  <!-- Other module inherits                                   -->

  <!-- Specify the app entry point class.                      -->
  <entry-point class='com.gwt.example.client.GWTDemo'/>

  <!-- Specify the paths for translatable code                 -->
  <source path='client'/>

</module>
```

Listing 2.3: The GWTDemo module definition file

Additional modules might be added to the project using this file. A module is an extension of the client GWT framework, including additional widgets, utility classes, and so on. Modules must be packed in a .JAR file, contain both compiled classes AND their source code, as well as a `.gwt.xml` descriptor.

2.2.5 Coding the client; GWT Widgets

The presentation tier, or client, of a GWT application is coded in a subset of Java, using special classes provided by GWT called Widgets. These widgets are then translated into Javascript code, and embedded into container elements inside an HTML file. The element `id` property is used to identify which widget will be placed in each container element.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
    <link type="text/css" rel="stylesheet" href="GWTDemo.css">
    <title>Web Application Starter Project</title>
    <!-- Load the compiled Javascript module -->
    <script type="text/javascript" language="javascript" src="gwtdemo/gwtdemo.nocache.js">
        </script>
  </head>
  <!--Body may be empty, or may contain static elements.-->
  <body>
    <h1>Web Application Starter Project</h1>
    <table align="center">
      <tr>
        <td colspan="2" style="font-weight:bold;">Please enter your name:</td>
      </tr>
      <tr>
    <!--widgets will be embedded into these table cells using their id-->
        <td id="nameFieldContainer"></td>
        <td id="sendButtonContainer"></td>
      </tr>
    </table>
  </body>
</html>
```

Listing 2.4: GWTDemo default HTML file.

For example, one might use the `body` element to have a full-page application running, or smaller containers like `div` or `td` to mix them with classic HTML design. Widgets can also be grouped using special container widgets called Panels, or with composite classes.

Certain Widgets might implement Handlers in order to react to user interaction. In a sense, GUI design with GWT doesn't differ very much from traditional Java GUI design with Swing or similar APIs.

The client might also implement some or all the application tier, from simple validation and error checking to obtaining and processing data from external sources, such as RSS feeds. A subset of the java.lang API can be compiled to Javascript.

```
package com.gwt.example.client;

import com.google.gwt.core.client.*;
import com.google.gwt.event.dom.client.*;
import com.google.gwt.user.client.ui.*;
import com.google.gwt.user.client.rpc.AsyncCallback;

//EntryPoint defines the onModuleLoad() method, which is to be run at the initial load of
     the web application.
public class GWTDemo implements EntryPoint {

   private static final String SERVER_ERROR = "An error occurred while "
          + "attempting to contact the server. Please check your network "
          + "connection and try again.";
   //Proxy to make Remote Procedure Calls to a service running in a servlet.
   private final GreetingServiceAsync greetingService = GWT
          .create(GreetingService.class);

   //onModuleLoad() method.
   public void onModuleLoad() {
      //Creation of a Button widget
      final Button sendButton = new Button("Send");
      //Creation of a Textbox widget
      final TextBox nameField = new TextBox();
      //Setting the "Text" property of the textbox
      nameField.setText("GWT User");
      //We can add style names to widgets
      sendButton.addStyleName("sendButton");
      //Add the nameField and sendButton to the RootPanel, a special container in the
          HTML page where the    //widgets will be embedded.
      //RootPanel.get() will return the entire body element
      RootPanel.get("nameFieldContainer").add(nameField);
      RootPanel.get("sendButtonContainer").add(sendButton);
      //Focus the cursor on the name field when the app loads
      nameField.setFocus(true);
      nameField.selectAll();
      //Create a popup dialog box
      final DialogBox dialogBox = new DialogBox();
      dialogBox.setText("Remote Procedure Call");
      dialogBox.setAnimationEnabled(true);
      final Button closeButton = new Button("Close");
      // We can set the id of a widget by accessing its Element
      closeButton.getElement().setId("closeButton");
      final Label textToServerLabel = new Label();
      final HTML serverResponseLabel = new HTML();
      //Widgets can be combined and arranged by use of panels.
      VerticalPanel dialogVPanel = new VerticalPanel();
```

```
        dialogVPanel.addStyleName("dialogVPanel");
        dialogVPanel.add(new HTML("<b>Sending name to the server:</b>"));
        dialogVPanel.add(textToServerLabel);
        dialogVPanel.add(new HTML("<br><b>Server replies:</b>"));
        dialogVPanel.add(serverResponseLabel);
        dialogVPanel.setHorizontalAlignment(VerticalPanel.ALIGN_RIGHT);
        dialogVPanel.add(closeButton);
        dialogBox.setWidget(dialogVPanel);
        //Adding a ClickHandler to close the DialogBox
        closeButton.addClickHandler(new ClickHandler() {
            public void onClick(ClickEvent event) {
                dialogBox.hide();
                sendButton.setEnabled(true);
                sendButton.setFocus(true);
            }
        });
        //Defining a new class to add a ClickHandler and KeyUpHandler for the sendButton
             and nameField.
        class MyHandler implements ClickHandler, KeyUpHandler {

            //Fired when the user clicks on the sendButton.
            Public void onClick(ClickEvent event) {
                sendNameToServer();
            }
            //Fired when the user types in the nameField.
            public void onKeyUp(KeyUpEvent event) {
                if (event.getNativeKeyCode() == KeyCodes.KEY_ENTER) {
                    sendNameToServer();
                }
            }
            //Send the name from the nameField to the server and wait for a response via an
                 Asynchronous RPC
            private void sendNameToServer() {
                sendButton.setEnabled(false);
                String textToServer = nameField.getText();
                textToServerLabel.setText(textToServer);
                serverResponseLabel.setText("");
                greetingService.greetServer(textToServer,
                //Creation of an AsyncCallback
                    new AsyncCallback<String>() {
                        //If the RPC fails, either due to a network error or because of a
                             User defined exception, the onFailure() function is run.
                        public void onFailure(Throwable caught) {
                            // Show the RPC error message to the user
                            dialogBox.setText("Remote Procedure Call - Failure");
                            serverResponseLabel.addStyleName("serverResponseLabelError");
                            serverResponseLabel.setHTML(SERVER_ERROR);
                            dialogBox.center();
                            closeButton.setFocus(true);
                        }
                        //If the RPC succeeds, the on Success() function is run with the
                             return from the RPC call as parameter..
                        public void onSuccess(String result) {
                            dialogBox.setText("Remote Procedure Call");
                            serverResponseLabel.removeStyleName("serverResponseLabelError");
                            serverResponseLabel.setHTML(result);
                            dialogBox.center();
                            closeButton.setFocus(true);
```

```
                    }
                });
            }
        }
        // Add the handler to send the name to the server
        MyHandler handler = new MyHandler();
        sendButton.addClickHandler(handler);
        nameField.addKeyUpHandler(handler);
    }
}
```

Listing 2.5: GWTDemo code. Widgets are embedded into elements defined in the HTML file.

2.2.6 Coding the server; Remote Procedure Calls

The optional server code is compiled into standard Java Servlets deployable in any web application container such as Apache Tomcat [2]. Servlets might use the full Java implementation and APIs, making them ideal for operations like database access, complex calculations, web service interoperation, and many more.

The approach of this server code is that of a service that can be invoked with asynchronous calls from the application client. This approach makes them a really flexible alternative for client-server communication.

```
package com.gwt.example.server;

import com.gwt.example.client.GreetingService;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;

/**
 * The server side implementation of the RPC service.
 */
@SuppressWarnings("serial")
public class GreetingServiceImpl extends RemoteServiceServlet implements
        GreetingService {

    public String greetServer(String input) {
        String serverInfo = getServletContext().getServerInfo();
        String userAgent = getThreadLocalRequest().getHeader("User-Agent");
        return "Hello, " + input + "!<br><br>I am running " + serverInfo
                + ".<br><br>It looks like you are using:<br>" + userAgent;
    }
}
```

Listing 2.6: GWTDemo “GreetingService” service implementation.

Service interfaces definition

For the client to be able to perform remote procedure calls to the server, at least two interfaces must be defined.

The first one is a standard Java interface, not unlike the ones used for a standard remote procedure call.

```
package com.gwt.example.client;
```

```java
import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;

/**
 * The client side stub for the RPC service.
 */
@RemoteServiceRelativePath("greet")
public interface GreetingService extends RemoteService {
    String greetServer(String name);
}
```

Listing 2.7: GWTDemo “GreetingService” service interface.

Notice the annotation `@RemoteServiceRelativePath()`. This defines the Servlet binding, and must match the one defined in the `web.xml` for the service to work.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

  <!-- Servlets -->
  <servlet>
    <servlet-name>greetServlet</servlet-name>
    <servlet-class>com.gwt.example.server.GreetingServiceImpl</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>greetServlet</servlet-name>
    <url-pattern>/gwtdemo/greet</url-pattern>
  </servlet-mapping>

  <!-- Default page to serve -->
  <welcome-file-list>
    <welcome-file>GWTDemo.html</welcome-file>
  </welcome-file-list>

</web-app>
```

Listing 2.8: GWTDemo `web.xml` file.

The second one is an asynchronous interface. Some development tools allow for automatic async interface generation.

```java
package com.gwt.example.client;

import com.google.gwt.user.client.rpc.AsyncCallback;

/**
 * The async counterpart of <code>GreetingService</code>.
 */
public interface GreetingServiceAsync {
    void greetServer(String input, AsyncCallback<String> callback);
}
```

Listing 2.9: GWTDemo “GreetingService” service asynchronous interface.

Performing an asynchronous Remote Procedure Call is quite trivial once all interfaces have been defined.

```
//Obtain a reference to the service
GreetingServiceAsync greetingService = GWT.create(GreetingService.class);
//Create a new AsyncCallback outlining the original function's return type.
AsyncCallback acb = new AsyncCallback<String>() {
    //If the RPC fails, either due to a network error or because of a User defined
        exception, the onFailure() function is run.
    public void onFailure(Throwable caught) {
        // Show the RPC error message to the user, process the thrown exceptions, error
            management...
    }
    //If the RPC succeeds, the on Success() function is run with the return from the RPC
        call as parameter..
    public void onSuccess(String result) {
        //Do something with the obtained value.
    }
});
//Actual RPC call
greetingService.greetServer("Hello World",acb);
```

Listing 2.10: GWTDemo Asynchronous RPC.

Server modules

Since server code is compiled to plain old Java Servlets inside a .WAR file, external modules imported by server classes can be standard Java libraries packed in .JAR files, without the need for source code or descriptor inclusion.

Of course, GWT modules can also be loaded from the server code, since they also feature the compiled class files.

2.3 Other features

2.3.1 Internationalization

Google Web Toolkit features a set of tools[5] to help the developer create locale-aware web applications and libraries. Several techniques exist to give a multilingual approach to strings, typed values or classes.

2.3.2 Declarative User Interfaces

From Google Web Toolkit version 2 onwards, developers can create user interfaces declaratively in XML files[?], without the hassle of manually compositing widgets by code. It also allows for better collaborative work between programmers and UI designers.

Let's look at a simple example[9]

```
CheckBox rubyCheck = new CheckBox("Ruby");
rubyCheck.setFormValue("ruby");
boxes.add(rubyCheck);
CheckBox pythonCheck = new CheckBox("Python");
pythonCheck.setFormValue("python");
```

```
boxes.add(pythonCheck);
CheckBox javaCheck = new CheckBox("Java");
javaCheck.setFormValue("java");
boxes.add(javaCheck);
CheckBox phpCheck = new CheckBox("PHP");
phpCheck.setFormValue("php");
boxes.add(phpCheck);
// set up panel
Panel panel = new VerticalPanel();
panel.add(new Label("Choose a language:"));
```

Listing 2.11: Traditional UI widget composition using java code

Now, if we were to create the same UI using UiBinder, we would have to define two different files.

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
    xmlns:g='urn:import:com.google.gwt.user.client.ui'>
    <g:VerticalPanel>
      <g:Label>Choose a language:</g:Label>
      <g:CheckBox ui:field="rubyCheck" formValue="ruby">Ruby</g:CheckBox>
      <g:CheckBox ui:field="pythonCheck" formValue="python">Python</g:CheckBox>
      <g:CheckBox ui:field="javaCheck" formValue="java">Java</g:CheckBox>
      <g:CheckBox ui:field="phpCheck" formValue="php">PHP</g:CheckBox>
      <g:Button ui:field="button">Submit</g:Button>
    </g:VerticalPanel>
</ui:UiBinder>
```

Listing 2.12: UiBinder XML file

An XML file describing the widget layout and the widget identifiers...

```
public class LanguageList extends Composite {
  interface MyUiBinder extends UiBinder<Widget, LanguageList> {}
  private static MyUiBinder uiBinder = GWT.create(MyUiBinder.class);
  @UiField CheckBox rubyCheck;
  @UiField CheckBox pythonCheck;
  @UiField CheckBox javaCheck;
  @UiField CheckBox phpCheck;
  @UiField Button button;
  public LanguageList() {
      // bind XML file of same name of this class to this class
      initWidget(uiBinder.createAndBindUi(this));
      final ArrayList<CheckBox> boxes = new ArrayList<CheckBox>();
      boxes.add(rubyCheck);
      boxes.add(pythonCheck);
      boxes.add(javaCheck);
      boxes.add(phpCheck);
  }
```

Listing 2.13: Loading the UiBinder XML from Java code

... as well as a Java class that binds the XML declarative UI to our widget.

Bibliography

[1] Apache Ant Project [online]. URL: http://ant.apache.org/.

[2] Apache Tomcat Project [online]. URL: http://tomcat.apache.org/.

[3] Eclipse IDE Project [online]. URL: http://www.eclipse.org/.

[4] Google Chrome web browser [online]. URL: http://www.google.com/chrome?hl=en.

[5] Google Web Toolkit Developer Guide: Internationalization [online]. URL: http://code.google.com/intl/en/webtoolkit/doc/latest/DevGuideI18n.html%.

[6] Google Web Toolkit project homepage [online]. URL: http://code.google.com/intl/en-EN/webtoolkit/.

[7] JUnit Testing Framework [online]. URL: http://www.junit.org/.

[8] Mozilla Firefox web browser [online]. URL: http://www.mozilla-europe.org/en/firefox/.

[9] RubyGeek:GWT 2.0: UiBinder, first look at simple example [online]. URL: http://www.rubygeek.com/2009/10/31/gwt-20-uibinder-first-look-at-simpl%e-example/.

Chapter 3

OSGi and GWT integration

To integrate OSGi and GWT it would be ideal to be able to deploy a WAR package as it's done with Apache Tomcat. OSGi Http Service doesn't support the latest Servlet specification, and even if it did, one would have to strip out the Servlets and resources from the WAR and register them using the Http Service. Fortunately, Pax Web[17] already does that.

Pax web is an OSGi R4 Http Service and Web Container[13] implementation that extends the OSGi Http Service with newer Servlet support, filters, listeners, JSP support, automatic WAR deployment...

This chapter will detail how to use Pax web to enable OSGi and GWT integration, using Maven as project manager to ease workflow and make continuous integration easy. It is recommended before reading this chapter to read the "Continuous integration in OSGi projects using Maven" document as it explains Maven essentials and OSGi project management and the contents of this chapter just fit with those concepts.

3.1 Creating the GWT Project

The GWT project (or projects) will integrate with other OSGi bundles, so a typical scenario is a multimodule Maven project created with Pax Construct. That will be the setup used for the examples of this chapter, but it should be noted that the only important setup is correct project dependencies in the Maven POMs.

To create and manage the project it is a good idea to use the Google Web Toolkit Maven Plugin[14]. The plugin provides goals to manage all the lifecycle of GWT projects properly and even provides a Maven archetype to create a project template (which is not the same as the classic GWT template but a bare minimum template). Let's create the project inside the parent project directory.

```
mvn archetype:generate -DarchetypeGroupId=org.codehaus.mojo -DarchetypeArtifactId=gwt-
    maven-plugin -DarchetypeVersion=1.1 -DgroupId=es.deusto.morelab.project -DartifactId=
    GWTsubproject
```

Listing 3.1: GWT project creation example (Using Maven)

Let's take a look at the generated pom.xml

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

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
    maven.apache.org/maven-v4_0_0.xsd">
  <!--
    GWT-Maven archetype generated POM
  -->
  <modelVersion>4.0.0</modelVersion>
  <parent>
      <artifactId>ExampleProject</artifactId>
      <groupId>es.deusto.morelab.example</groupId>
      <version>1.0-SNAPSHOT</version>
    </parent>
  <groupId>es.deusto.morelab.example</groupId>
  <artifactId>GWTProject</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>gwt-maven-archetype-project</name>

  <properties>

      <!-- convenience to define GWT version in one place -->
      <gwt.version>1.6.4</gwt.version>

      <!--  tell the compiler we can use 1.5 -->
      <maven.compiler.source>1.5</maven.compiler.source>
      <maven.compiler.target>1.5</maven.compiler.target>

  </properties>

  <dependencies>

      <!--  GWT dependencies (from central repo) -->
    <dependency>
      <groupId>com.google.gwt</groupId>
      <artifactId>gwt-servlet</artifactId>
      <version>${gwt.version}</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>com.google.gwt</groupId>
      <artifactId>gwt-user</artifactId>
      <version>${gwt.version}</version>
      <scope>provided</scope>
    </dependency>

    <!-- test -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.4</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <outputDirectory>war/WEB-INF/classes</outputDirectory>
    <plugins>
      <plugin>
```

```
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>gwt-maven-plugin</artifactId>
      <version>1.1-SNAPSHOT</version>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>generateAsync</goal>
            <goal>test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <!--
        If you want to use the target/web.xml file mergewebxml produces,
        tell the war plugin to use it.
        Also, exclude what you want from the final artifact here.
          <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-war-plugin</artifactId>
              <configuration>
                  <webXml>target/web.xml</webXml>
                  <warSourceExcludes>.gwt-tmp/**</warSourceExcludes>
              </configuration>
          </plugin>
          -->

    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.0.2</version>
        <configuration>
          <source>${maven.compiler.source}</source>
          <target>${maven.compiler.target}</target>
        </configuration>
    </plugin>
  </plugins>
</build>

</project>
```

Listing 3.2: Generated GWT Project POM

According to the POM, the project will build as a WAR package. It will download and use the GWT version set in the `gwt.version` tag, so it is a good idea to set it to 2.0.0 or any other version that can be found in the central Maven repository[10] or any manually added repository. GWT dependencies have already been set and even JUnit has been configured for automated test running.

The build process is also configured to run some goals from gwt-maven-plugin. The `compile` and `test` goals are all right, but the `generateAsync` goal may be deleted and run manually if desired, as it attempts to generate the asynchronous interfaces automatically from the basic interfaces defined but in some projects those have already been written.

The `src` directory follows the Maven convention and has a `main` and a `test` directory. The `main` directory has 3 directories inside: `java` for sources, `resources` for web resources (HTML, CSS, custom Javascript…) and `webapp` for the `web.xml` configuration file and resources related to the WAR package.

3.2 Things to consider in GWT Maven projects

The usual GWT workflow is started by creating the GWT project using the provided Google tools. This created project is a example with some server and client side code and it is already configured to make both available. The GWT Maven example, on the contrary, only includes client side code and therefore doesn't have any Servlet registered in `web.xml`. When GWT server side code is created, the developer must remember to modify the `web.xml` file. This may be solved in the future with a custom archetype.

Should the need for the GWT debug server arise, it is neccesary to configure the Google Web Toolkit Maven Plugin to set the `runTarget` element to the initial HTML view (`module/module.html`)

3.3 Enabling OSGi in the GWT project

To enable OSGi integration the GWT project must be somehow transformed into a bundle. For the project to be considered a bundle it needs the following elements:

- A Bundle Activator
- An OSGi-compliant `MANIFEST.MF` file
- Compile-time OSGi dependencies

One may believe it is also needed to deploy the project as a JAR package instead of a WAR package, but that is not neccesary given the three previous elements are correctly included.

3.3.1 Turning the GWT project into a bundle

OSGi dependencies

The first step should be configuring the Maven project to import OSGi libraries on build time. For that, let's just add two dependencies to the POM.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
    maven.apache.org/maven-v4_0_0.xsd">
  <!--
    GWT-Maven archetype generated POM
  -->
  <modelVersion>4.0.0</modelVersion>
  <parent>
      <artifactId>ExampleProject</artifactId>
      <groupId>es.deusto.morelab.example</groupId>
      <version>1.0-SNAPSHOT</version>
    </parent>
  <groupId>es.deusto.morelab.example</groupId>
  <artifactId>GWTProject</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>gwt-maven-archetype-project</name>

  <properties>
```

```
    <!-- convenience to define GWT version in one place -->
    <gwt.version>2.0.0</gwt.version>

    <!--  tell the compiler we can use 1.5 -->
    <maven.compiler.source>1.5</maven.compiler.source>
    <maven.compiler.target>1.5</maven.compiler.target>

</properties>

<dependencies>

    <!--  GWT dependencies (from central repo) -->
  <dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-servlet</artifactId>
    <version>${gwt.version}</version>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-user</artifactId>
    <version>${gwt.version}</version>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>org.osgi</groupId>
    <artifactId>osgi_R4_core</artifactId>
    <version>1.0</version>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.osgi</groupId>
    <artifactId>osgi_R4_compendium</artifactId>
    <optional>true</optional>
    <version>1.0</version>
  </dependency>

  <!-- test -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.4</version>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <outputDirectory>war/WEB-INF/classes</outputDirectory>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>gwt-maven-plugin</artifactId>
      <version>1.1-SNAPSHOT</version>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
```

```xml
            <goal>generateAsync</goal>
            <goal>test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <!--
        If you want to use the target/web.xml file mergewebxml produces,
        tell the war plugin to use it.
        Also, exclude what you want from the final artifact here.
          <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-war-plugin</artifactId>
              <configuration>
                  <webXml>target/web.xml</webXml>
                  <warSourceExcludes>.gwt-tmp/**</warSourceExcludes>
              </configuration>
          </plugin>
          -->

    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.0.2</version>
        <configuration>
          <source>${maven.compiler.source}</source>
          <target>${maven.compiler.target}</target>
        </configuration>
    </plugin>
   </plugins>
  </build>

</project>
```

Listing 3.3: GWT Project POM with generated OSGi dependencies

Creating a Bundle Activator

The Bundle Activator to create will be just a standard one, implementing the BundleActivator interface.

Creating a valid MANIFEST.MF

There are two options to create the `MANIFEST.MF` file:

- Dynamic generation on build time using the Felix Maven Bundle Plugin[12]
- Manually create the file and configure Maven to package it in the WAR package

For this example the second option was chosen, but information on how to use the Felix Maven Bundle Plugin may be found in it's homepage[12].

First the `MANIFEST.MF` file must be created. To package the `MANIFEST.MF` file in the WAR package `maven-war-plugin` must be configured to use the manually created MANIFEST.MF file as the WAR manifest. To do so, a plugin configuration entry must be created.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
    maven.apache.org/maven-v4_0_0.xsd">
  <!--
    GWT-Maven archetype generated POM
  -->
  <modelVersion>4.0.0</modelVersion>
  <parent>
       <artifactId>ExampleProject</artifactId>
       <groupId>es.deusto.morelab.example</groupId>
       <version>1.0-SNAPSHOT</version>
    </parent>
  <groupId>es.deusto.morelab.example</groupId>
  <artifactId>GWTProject</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>gwt-maven-archetype-project</name>

  <properties>

     <!-- convenience to define GWT version in one place -->
     <gwt.version>2.0.0</gwt.version>

     <!-- tell the compiler we can use 1.5 -->
     <maven.compiler.source>1.5</maven.compiler.source>
     <maven.compiler.target>1.5</maven.compiler.target>

  </properties>

  <dependencies>

     <!-- GWT dependencies (from central repo) -->
    <dependency>
      <groupId>com.google.gwt</groupId>
      <artifactId>gwt-servlet</artifactId>
      <version>${gwt.version}</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>com.google.gwt</groupId>
      <artifactId>gwt-user</artifactId>
      <version>${gwt.version}</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.osgi</groupId>
      <artifactId>osgi_R4_core</artifactId>
      <version>1.0</version>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.osgi</groupId>
      <artifactId>osgi_R4_compendium</artifactId>
      <optional>true</optional>
      <version>1.0</version>
    </dependency>
```

```
  <!-- test -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.4</version>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <outputDirectory>war/WEB-INF/classes</outputDirectory>
  <plugins>
    <plugin>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
          <archive>
              <manifestFile>src/main/webapp/WEB-INF/MANIFEST.MF</manifestFile>
          </archive>
  </configuration>
    </plugin>

    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>gwt-maven-plugin</artifactId>
      <version>1.1-SNAPSHOT</version>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>generateAsync</goal>
            <goal>test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <!--
        If you want to use the target/web.xml file mergewebxml produces,
        tell the war plugin to use it.
        Also, exclude what you want from the final artifact here.
          <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-war-plugin</artifactId>
              <configuration>
                  <webXml>target/web.xml</webXml>
                  <warSourceExcludes>.gwt-tmp/**</warSourceExcludes>
              </configuration>
          </plugin>
          -->

    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.0.2</version>
        <configuration>
          <source>${maven.compiler.source}</source>
          <target>${maven.compiler.target}</target>
        </configuration>
```

```
    </plugin>
  </plugins>
</build>

</project>
```

Listing 3.4: GWT Project POM with MANIFEST.MF packaging

In this example the file was placed under `src/main/webapp/WEB-INF/`, but it may be placed anywhere under the project root.

3.3.2 Dependencies management

In order for the OSGified GWT project to consume other bundle's services, it needs to declare dependency upon those bundles to have access to their interfaces during build time (and obviously must import some packages in it's `MANIFEST.MF` file). The same happens when other bundles want to consume services provided by the OSGified GWT project.

Dependence of other bundles upon the GWT project

The problem with WAR packages is that they are not like JARs dependency-wise. Classes are distributed in different ways and then the dependency mechanisms can't import them. Fortunately, Maven can generate and install a JAR file containing the `.class` files. To achieve this, another change must be introduced to the `maven-war-plugin` configuration of the OSGified GWT project.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
    maven.apache.org/maven-v4_0_0.xsd">
  <!--
    GWT-Maven archetype generated POM
  -->
  <modelVersion>4.0.0</modelVersion>
  <parent>
        <artifactId>ExampleProject</artifactId>
        <groupId>es.deusto.morelab.example</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
  <groupId>es.deusto.morelab.example</groupId>
  <artifactId>GWTProject</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>gwt-maven-archetype-project</name>

  <properties>

      <!-- convenience to define GWT version in one place -->
      <gwt.version>2.0.0</gwt.version>

      <!-- tell the compiler we can use 1.5 -->
      <maven.compiler.source>1.5</maven.compiler.source>
      <maven.compiler.target>1.5</maven.compiler.target>

  </properties>

  <dependencies>
```

```
    <!-- GWT dependencies (from central repo) -->
  <dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-servlet</artifactId>
    <version>${gwt.version}</version>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-user</artifactId>
    <version>${gwt.version}</version>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>org.osgi</groupId>
    <artifactId>osgi_R4_core</artifactId>
    <version>1.0</version>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.osgi</groupId>
    <artifactId>osgi_R4_compendium</artifactId>
    <optional>true</optional>
    <version>1.0</version>
  </dependency>

  <!-- test -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.4</version>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <outputDirectory>war/WEB-INF/classes</outputDirectory>
  <plugins>
    <plugin>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
          <archive>
              <manifestFile>src/main/webapp/WEB-INF/MANIFEST.MF</manifestFile>
          </archive>
      <attachClasses>true</attachClasses>
  </configuration>
    </plugin>

    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>gwt-maven-plugin</artifactId>
      <version>1.1-SNAPSHOT</version>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
```

```
            <goal>generateAsync</goal>
            <goal>test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <!--
        If you want to use the target/web.xml file mergewebxml produces,
        tell the war plugin to use it.
        Also, exclude what you want from the final artifact here.
          <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-war-plugin</artifactId>
              <configuration>
                  <webXml>target/web.xml</webXml>
                  <warSourceExcludes>.gwt-tmp/**</warSourceExcludes>
              </configuration>
          </plugin>
          -->

    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.0.2</version>
        <configuration>
          <source>${maven.compiler.source}</source>
          <target>${maven.compiler.target}</target>
        </configuration>
    </plugin>
  </plugins>
</build>

</project>
```

Listing 3.5: GWT Project POM with additional classes JAR generation

To depend upon this additional JAR file, the dependency entry of other bundles should use the `classes` classifier and the `provided` scope, the first to use the JAR file and the second to not merge the classes in the bundle and assume they will be provided by someone else in runtime.

```
<dependency>
    <groupId>es.deusto.morelab.agurezain</groupId>
    <artifactId>GWTAPP</artifactId>
    <version>1.0-SNAPSHOT</version>
    <classifier>classes</classifier>
    <scope>provided</scope>
</dependency>
```

Listing 3.6: Dependency entry to the GWT project

Dependence upon other bundles

Dependence upon other bundles is configured normally via POM dependencies. Bundles are of type (packaging) `bundle`, so it may be neccesary to indicate so. Bundle manifests must reflect the imported and exported packages too. Follow the instructions of “Continuous integration in OSGi projects using Maven”.

3.3.3 Run using Pax Runner

A usual strategy to ensure easy project startup is to copy the Pax Runner[16] JAR to the parent project directory and create a script that will use Pax Runner with a composite provision file[15]. The composite provision file describes what bundles should be acquired and how, and the script can do some configuration on it's own, like boot delegation[11] configuration (needed to use database drivers under OSGi, for example).

The following two snippets show a composite provision file that deploys some bundles via a profile, other bundles via HTTP URLs, other bundles via Maven, including a GWT WAR bundle and a Bash script that uses Pax Runner with the previous composite provision configuration file and boot delegation for all packages.

```
#Libraries and system bundles
scan-bundle:mvn:org.eclipse.osgi/services/3.1.200-v20070605@1
scan-bundle:mvn:org.eclipse.equinox/event/1.0.100-v20070516@1
scan-composite:mvn:org.ops4j.pax.runner.profiles/war/0.7.2/composite
scan-bundle:http://www.eclipse.org/downloads/download.php?file=/equinox/drops/R
    -3.5.1-200909170800/javax.servlet_2.5.0.v200806031605.jar&url=http://eclipsemirror.
    yoxos.com/eclipse.org/equinox/drops/R-3.5.1-200909170800/javax.servlet_2.5.0.
    v200806031605.jar&mirror_id=200@1
scan-bundle:http://www.eclipse.org/downloads/download.php?file=/equinox/drops/R
    -3.5.1-200909170800/org.eclipse.equinox.http.servlet_1.0.200.v20090520-1800.jar&url=
    http://d2u376ub0heus3.cloudfront.net/equinox/drops/R-3.5.1-200909170800/org.eclipse.
    equinox.http.servlet_1.0.200.v20090520-1800.jar&mirror_id=1032@1

#Example Project bundles
scan-bundle:war:mvn:es.deusto.morelab.example/GWTProject/1.0-SNAPSHOT/war@update
scan-bundle:mvn:es.deusto.morelab.example/OSGiBundle/1.0-SNAPSHOT@update
```

Listing 3.7: Composite provision configuration

```
#!/bin/sh
#
# Script to run Pax Runner, which starts OSGi frameworks with applications.

java -cp .:pax-runner-1.3.0.jar org.ops4j.pax.runner.Run --platform=equinox scan-
    composite:file:provision.conf --bootdelegation=*
```

Listing 3.8: Runner script

Bibliography

[10] Central Maven Repository [online]. URL: http://repo1.maven.org/maven2/.

[11] Equinox boot delegation information [online]. URL: http://wiki.eclipse.org/Equinox_Boot_Delegation.

[12] Felix Maven Bundle Plugin homepage [online]. URL: http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html%.

[13] Introduction to the OSGi Web Container [online]. URL: http://blog.springsource.com/2009/05/27/introduction-to-the-osgi-web-c%ontainer/.

[14] Maven GWT Plugin homepage [online]. URL: http://mojo.codehaus.org/gwt-maven-plugin/.

[15] Pax Runner composite provisioning documentation [online]. URL: http://paxrunner.ops4j.org/display/paxrunner/Composite+provisioning.

[16] Pax Runner project homepage [online]. URL: http://paxrunner.ops4j.org/space/Pax+Runner.

[17] Pax web project homepage [online]. URL: http://wiki.ops4j.org/display/paxweb/Pax+Web.