

A vertical photograph of a waterfall in a dense, mossy forest. The water is white and turbulent as it falls, creating a misty spray at the bottom. The surrounding rocks and foliage are covered in vibrant green moss and ferns. The lighting is soft, filtering through the trees.

# Pretty GXT Themes

*The guide for making  
beautiful themes for Senc  
GXT*

# CREATING GXT THEMES

## WHAT IS GXT THEMES

### OVERVIEW

There are many different flavors of gxt themes. Sencha has gone to great lengths to lay ground work for making its look and feel extensible. The third version of GXT has completely rewritten the architecture of this. This was done in order to leverage more of the Appearance interfaces that GWT uses to render its components. Previous versions of GXT and all other GWT frameworks, such as SmartGWT rely entirely on CSS selectors and classes to control layout.

This worked good for small application, and even some less complex medium sized application. However, this failed horribly when trying to scale the applications user experience to more complex design. The end result of this design implementation created web applications that could not stray far from the look and feel of the core GXT components.

Front end designers soon found themselves hardcoding the look and feel into the presentation renderers (the interfaces used to render the components). This created designs that are baked into the project, and could not translate into other projects. Another negative side to this approach was creating monstrosously sized css sheets, and illogically formed image sprites.

Due to the tremendous growth and adoption of GXT, something had to be done. Sencha, the company that makes GXT, decided to rewrite there component rendering models. This overhaul utilized new interface layers that allowed developers to inject in Appearance classes. This technic was borrowed from how GWT renders its cell within datagrids, and has not yet been fully implemented into all GWT components.

GXT provided a work around to this, by implementing there own interface layer to handle injecting these Appearance classes. The appearance classes are used to define exact rendering of the components. This fixed the issue of developers baking in there designs into the presentation layer of there GXT web app. But this came at a cost.

The once popular framework, Tapestry, had this same issue. Tapestry touted the ability to inject reusable components into presentation templates, but at a cost. Your base classes needed to be abstract in order to allow flexibility to instantiate components without having to use a factory design. This leaves you with abstract class hell. Your once elegant object oriented code that used slick polymorphism and IoC dependency injection, now has become riddled with abstracted classes everywhere.

This completely defeats the point of creating interfaces for IoC, which undermines the entire methodology of this design pattern. Ultimately this decimated the popularity of this framework for scalable application. With the inception of JQuery and GWT, this decimated the popularity of this technology. These modern javascript rendering engines allowed engineers to develop large web apps that could were very extensible, while greatly increasing maintainability.

I see a similar problem arising by using the Appearance design model for theming. You are forced to inject the custom appearance class into the constructor of the component you are creating. And you will need an individual appearance class for each component you wish to style. This will make for not so elegant code. It is a necessity for

the presentation layer to be implicitly called by the applications controller for that view. Such as how the browser applies CSS to the dom structure of the webpage. It is bad practice to code CSS inline into your document, as it drastically increases complexity of your view. Additionally this makes your code hard to maintain and test.

injecting these appearance seem similar in nature to this problem. Take the case where you have the same component rendered in two different contextual places. And each context requires the design to be changed slightly, such as its position or additional content that needs to be overlayed on the component. You will need to create logic within the presenter to determine which appearance needs to be injected.

The only way to escape this using a bunch of if else statements or switch cases, but this technic destroys your dependency injection design pattern of GXT. To satisfy dependency injection design pattern you would need to create abstract classes which can have its rendering methods overloaded to specify the appearance to inject. This makes it unit testable, but it is a slippery slope and erodes your testability over time.

This problem has yet to be solved for complex GXT theme designs. Complex themes are basically any change that is not color or size associated. The layout of the component and overall flow of the design must not differ from the core GXT look and feel. The only way to accomplish such complex designs is to implement your own component set avoiding the additional interface layers that allow for appearance injection into all of the components.

So when you boil it all down, theming GXT destroys your ability to reuse components for different layouts without creating components that inject the appearances. This avoids having to make generic abstract classes that allow for reusability at the cost of almosy unmaintainable code. However the appearances are baked into the build process, and for complex layouts, a majority of the complex components will not be cross project compabitly.

#### DEVELOPMENT LIFECYCLE

- 1.) setup GWT dev environment
- 2.) create new GXT project, and implement first component defined by business priority. (The portal Layout)
- 3.) investigate and get experimental appearance injection working
- 4.) using mockup of portal design, modify experimental appearance in (3.) to this look and feel
- 5.) update maven builds to reflect any additional resources that need to be assembled for the module
- 6.) create additional appearance classes for all of target runtimes (browser, OS, and device)
- 7.) test in all of the target runtimes and browsers
- 8.) document procedure
- 9.) repeat from (4.) for next component

**//TODO: insert fancy flowchart for representing this lifecycle**

**//TODO: reformat this into an unordered list**

## DEVELOPMENT SETUP

### REQUIREMENTS

- Eclipse 4.2
- Eclipse GWT Plugin 4.2
- GXT 3.0.1
- GWT 2.5.1

### SETUP GWT IN ECLIPSE

Google Web Toolkit provides a set of tools that can simply be used with a text editor, the command line, and a browser. However, you may also use GWT with your favorite IDE. Google provides a plugin for Eclipse that makes development with Google Web Toolkit even easier.

### DOWNLOAD ECLIPSE

If you do not already have Eclipse, you may download it from the [Eclipse Website](#). We suggest downloading Eclipse 4.2 (Juno).

**//TODO: this should point to where the latest version can be found**

### INSTALL THE PLUGIN

Install the Google Plugin for Eclipse 3.7 by using the following update site:

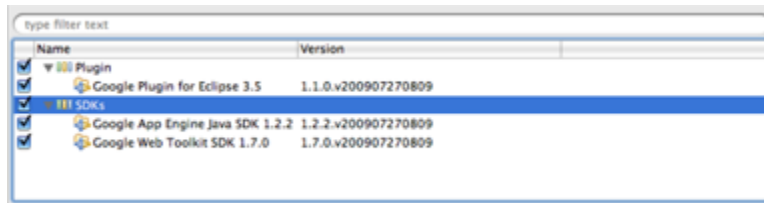
<http://dl.google.com/eclipse/plugin/3.7>

If you are using an earlier version of Eclipse, replace the 3.7 version number with your version (3.6 or 3.5). For detailed instructions on installing plugins in Eclipse, see instructions for [Eclipse 3.7](#), [Eclipse 3.6](#), or [Eclipse 3.5](#).

In the Install dialog, you will see an option to install the Plugin as well as the GWT and App Engine SDKs. Choosing the SDK options will install a GWT and/or App Engine SDK within your Eclipse plugin directory as a convenience.

**//TODO put in a tip box with more information on this**

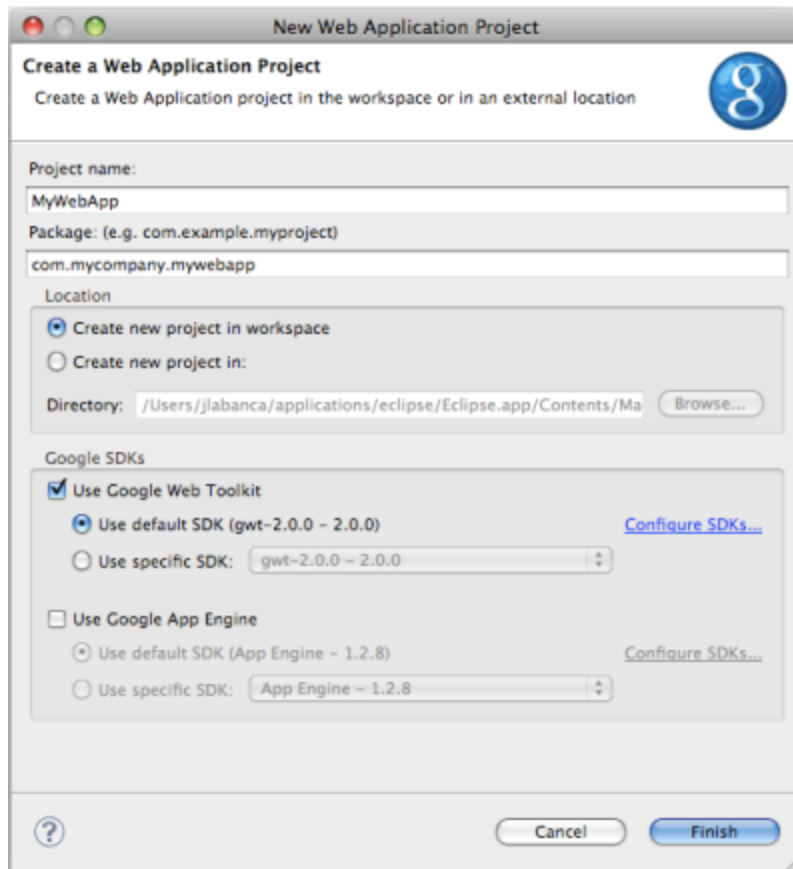
GWT release candidates are not bundled with The Google Plugin For Eclipse. If you're interested in using a GWT RC SDK, download and add it to your workspace as described [here](#).



## CREATE A WEB APPLICATION

To create a Web Application, select **File > New > Web Application Project** from the Eclipse menu.

In the **New Web Application Project** wizard, enter a name for your project and a java package name, e.g., `com.mycompany.mywebapp`. If you installed the Google App Engine SDK, the wizard gives you the option to use App Engine as well. For now, uncheck this option and click **Finish**.



Congratulations, you now have a Google Web Toolkit enabled web application. The plugin has created a boilerplate project in your workspace.

---

## RUN LOCALLY IN DEVELOPMENT MODE

Right-click on your web application project and select **Debug As > Web Application** from the popup menu.

This creates a **Web Application** launch configuration for you and launches it. The web application launch configuration will start a local web server and GWT development mode server.

You will find a Web Application view next to the console window. Inside you will find the URL for the development mode server. Paste this URL into Firefox, Internet Explorer, Chrome, or Safari. If this is your first time using that browser with the development mode server, it will prompt you to install the Google Web Toolkit Developer Plugin. Follow the instructions in the browser to install.

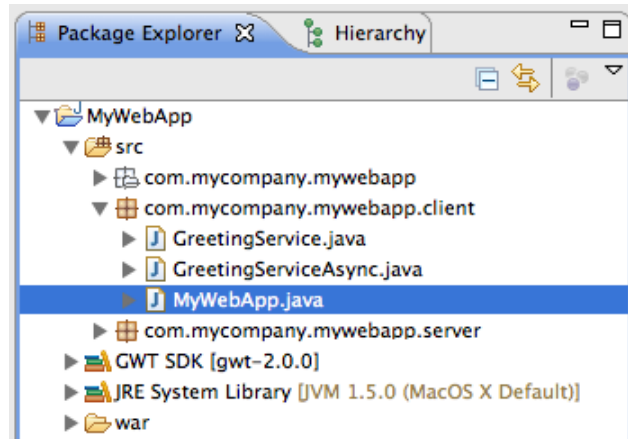


Once the browser plugin is installed, navigate to the URL again and the starter application will load in development mode.

---

## MAKE A FEW CHANGES

The source code for the starter application is in the `MyWebApp/src/` subdirectory, where `MyWebApp` is the name you gave to the project. You'll see two packages, `com.mycompany.mywebapp.client` and `com.mycompany.mywebapp.server`. Inside the client package is code that will eventually be compiled to JavaScript and run as client code in the browser. The java files in the server package will be run as Java bytecode on a server.



Look inside the `MyWebApp.java` file in the client package. Line 40 constructs the send button.

```
final Button sendButton = new Button("Send");
```

Change the text from "Send" to "Send to Server".

```
final Button sendButton = new Button("Send to Server");
```

Now, save the file and simply click "Refresh" back in your browser to see your change. The button should now say "Send to Server" instead of "Send".

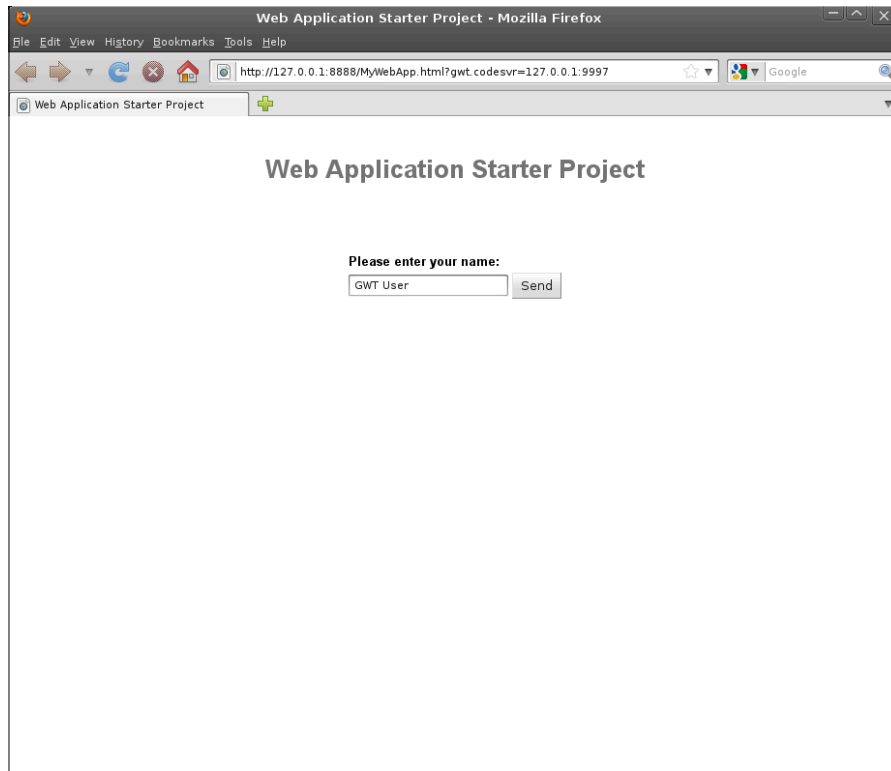
At this point, you can also set breakpoints, inspect variables and modify code as you would normally expect from a Java Eclipse debugging session.

---

## COMPILE AND RUN IN PRODUCTION MODE

To run the application as JavaScript in what GWT calls "production mode", compile the application by right-clicking the project and choosing **Google > GWT Compile**.

This command invokes the GWT compiler which generates a number of JavaScript and HTML files from the `MyWebApp` Java source code in the `MyWebApp/war/` subdirectory. To see the final application, open the file `MyWebApp/war/MyWebApp.html` in your web browser.



Congratulations! You've created your first web application using Google Web Toolkit. Since you've compiled the project, you're now running pure JavaScript and HTML that works in IE, Chrome, Firefox, Safari, and Opera. You could now deploy your application to production by serving the HTML and JavaScript files in your `MyWebApp/war/` directory from your web servers.

---

## DEPLOY TO APP ENGINE

Using the plugin, you can also easily deploy GWT projects to Google App Engine. If you installed the App Engine for Java SDK when you installed the plugin, you can now right-click on the project and App Engine "enable" it by choosing **Google > App Engine Settings**. Check the box marked **Use Google App Engine**. This will add the necessary configuration files to your project.

To deploy your project to App Engine, you first need to create an application ID from the [App Engine Administration Console](#).

Once you have an application ID, right-click on your project, and select **Google > App Engine Settings...** from the context menu. Enter your application ID into the **Application ID** text box. Click **OK**.

Right-click on your project and select **Google > Deploy to App Engine**. In the resulting **Deploy Project to Google App Engine** dialog, enter your Google Account email and password.

Click **Deploy**.



Congratulations! You now have a new web application built with Google Web Toolkit live on the web at <http://application-id.appspot.com/>.

## SETTING UP GXT

**//TODO: should properly include this into the bibliography**

<http://neiliscoding.blogspot.com/2012/05/getting-started-quick-setup-for-gxt-3.html>

## CREATE A USER DEFINED LIBRARY

Select Windows > Preferences and select Java > Build Path > User Libraries in the preference tree.

Click New and type "GXT" into the User library name field. Make sure your new library is selected and click "Add JARs". Navigate to the root folder of the Ext GWT download and add the "gxt.jar" file. Click OK. (Optional; If you prefer seeing jars in the build path rather than a user library, you can just click "Add External JARs" instead.)

## UPDATE GWT PROJECTS HOST PAGE

GXT 3.X requires a strict dtd. Add either of these as the first line in the host page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

or

```
<!DOCTYPE html>
```

Add a stylesheet link to reset.css and ensure it is the first referenced stylesheet:

```
<link rel="stylesheet" type="text/css" href="{module name}/reset.css" />
```

So in eclipse if you project is named "MyProject":

```
<link rel="stylesheet" type="text/css" href="MyProject /reset.css" />
```

Ext GWT Development Team Colin Alworth Says:

In your module file (ends in .gwt.xml) you might have a rename-to attribute - if so, that is the name of the module. This is the folder in which the modulename.cache.js file is created, along with other things like a hosted.html file, and eventually several <giant-hash-name>.cache.html files. If you don't specify a rename-to, it will default to the path to your module. So a module file at com/my/project/MyProject.gwt.xml will be com.my.project.MyProject, and the css file will be loaded like this:

```
<link rel="stylesheet" type="text/css" href="com.my.project.MyProject/reset.css" />
```

---

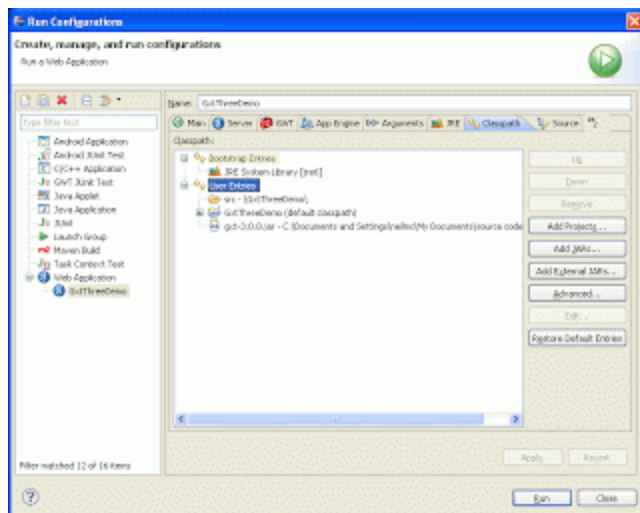
ADD GXT-3.X.X.JAR TO THE PROJECT DIRECTLY FROM THE DOWNLOADED ZIP.

1. Right click on project name in 'Package Explorer'.
2. Select 'Properties' from the content menu.
3. Select 'Java Build Path'.
4. Select 'Libraries' tab.
5. Add the gxt-3.X.X.jar either with 'Add JARs...' or 'Add External JARs...'.
6. (Optional) -- if you have created a User Library earlier on you can add the User Library you created instead

---

ADD GXT JAR TO LAUNCH CONFIGURATION.

1. Choose Run / Open Run Dialog.
2. Select your appropriate launch configuration under 'Java Application'.
3. Select the 'Classpath' tab.
4. Add the gxt-3.X.X.jar to the classpath under 'User Entries'.



---

UPDATE PROJECT'S MODULE XML FILE.

```
<inherits name='com.sencha.gxt.ui.GXT' />
```

This inherits all GXT modules. As an alternative, individual GXT modules can be inherited rather than inheriting all modules, but this is the easiest way to get started, and covers most of the default cases.

---

#### COMMENT OUT GWT THEME

Optional: If you used the GWT template code comment out the following to remove page margins;

```
<inherits name='com.google.gwt.user.theme.clean.Clean' />
```

---

#### MAVEN SETUP FOR PROJECT AND DEPENDENCY MANAGEMENT

If using Maven for project and dependency management, Sencha GXT artifacts are available in several ways. To get the commercially licensed releases, you will need to add the following repository section to your pom.xml:

```
<repository>

  <id>sencha-commercial-release</id>

  <name>Sencha commercial releases</name>

  <url>https://maven.sencha.com/repo/commercial-release/</url>

</repository>
```

For GPLv3 licensed jars, you may use maven central. There are several artifacts available:

- **gxt-release** - A zip of the release, similar to how GXT was released before 3.0
- **gxt** - The core component and data classes
- **gxt-charts** - The new drawing and charting API
- **gxt-legacy** - Classes to ease porting projects from earlier GXT versions
- **uibinder-bridge** - Optional support to allow complete construction of non Widget types

using <ui:with> in UiBinder XML. Will not be required when Google releases 2.5

## CONTINUOUS INTEGRATION

### OVERVIEW

**//TODO: need to add some text that overview how continuous integration makes this project development agile**

### GWT MAVEN PLUGIN

This plugin allows you to run commonly found build targets within the maven command line. This is useful for managing project dependencies. This also allows for easier integration of GWT modules. Writing GXT modules requires us to roll up our styles into a gwt module which is imported into the project which you wish to create a theme for.

### USING THE ARCHETYPE

Using the archetype is the simplest way to get a clean, ready for development GWT application. It is comparable to the application created by SDK webAppCreator with the "-maven" option, with some tweaks to nicely integrate with Google Eclipse and M2Eclipse plugins.

---

### CREATE PROJECT ON THE COMMAND LINE

Use it as you would any other Maven archetype to create a template/stub project.

```
mvn archetype:generate \  
  -DarchetypeGroupId=org.codehaus.mojo \  
  -DarchetypeArtifactId=gwt-maven-plugin \  
  -DarchetypeVersion=2.5.1
```

The generated project can then be imported as "existing project" into Eclipse, or if you don't like Eclipse you can use another IDE and run command-line maven to launch GWT host mode with "mvn gwt:run".

Note: don't run mvn `archetype:create` as the plugin uses archetype 2.0 descriptor.

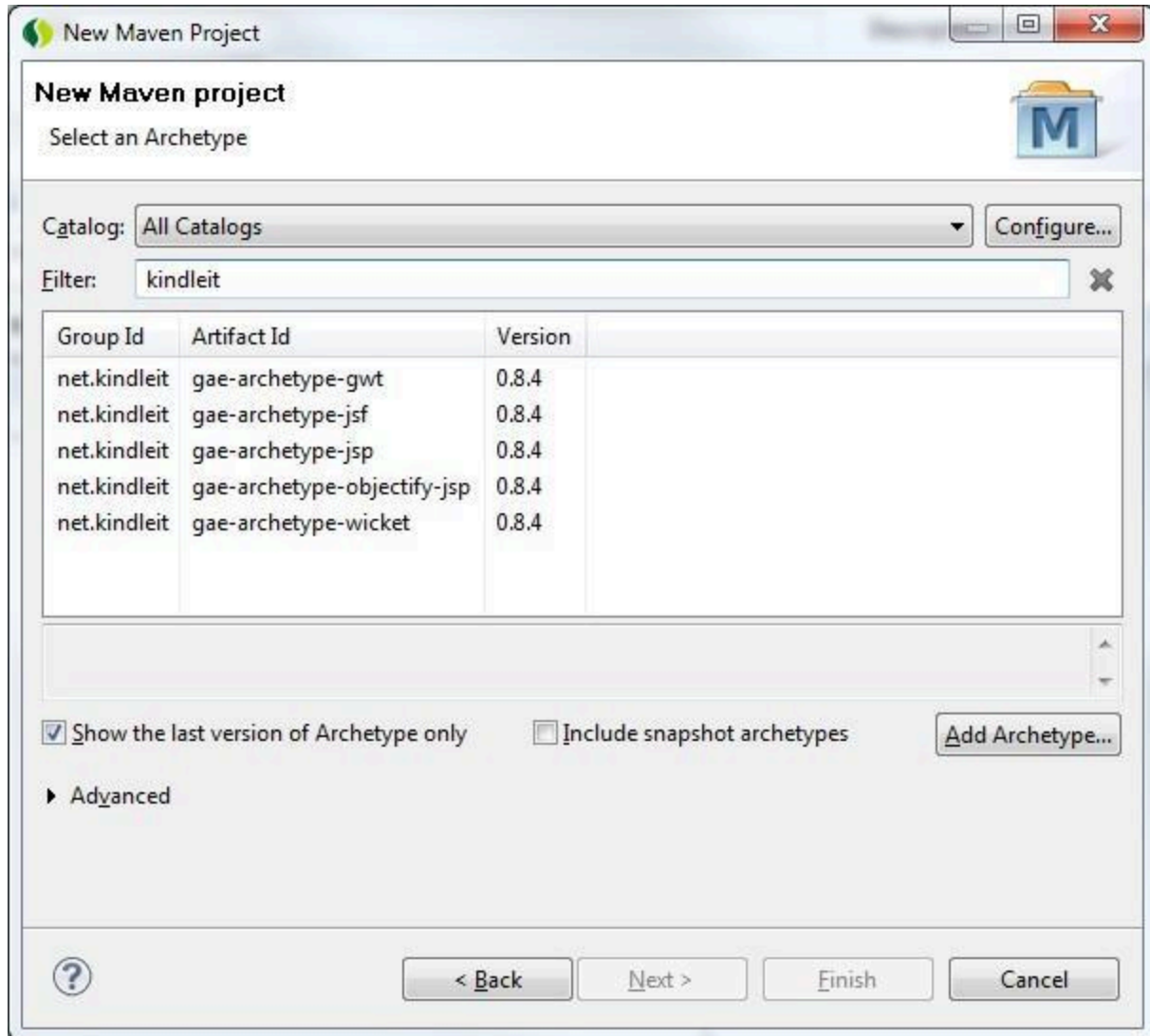
Running this from the command line sometimes will prompt you for more information to create it. Just take a look though the remote repo options and select the project settings according. This is also when you will setup your group and artifact setting for the project

---

### CREATING PROJECT USING ECLIPSE PLUGIN

First of all you have to have [m2eclipse plugin](#) installed to effectively use Maven in Eclipse. Once you have it:

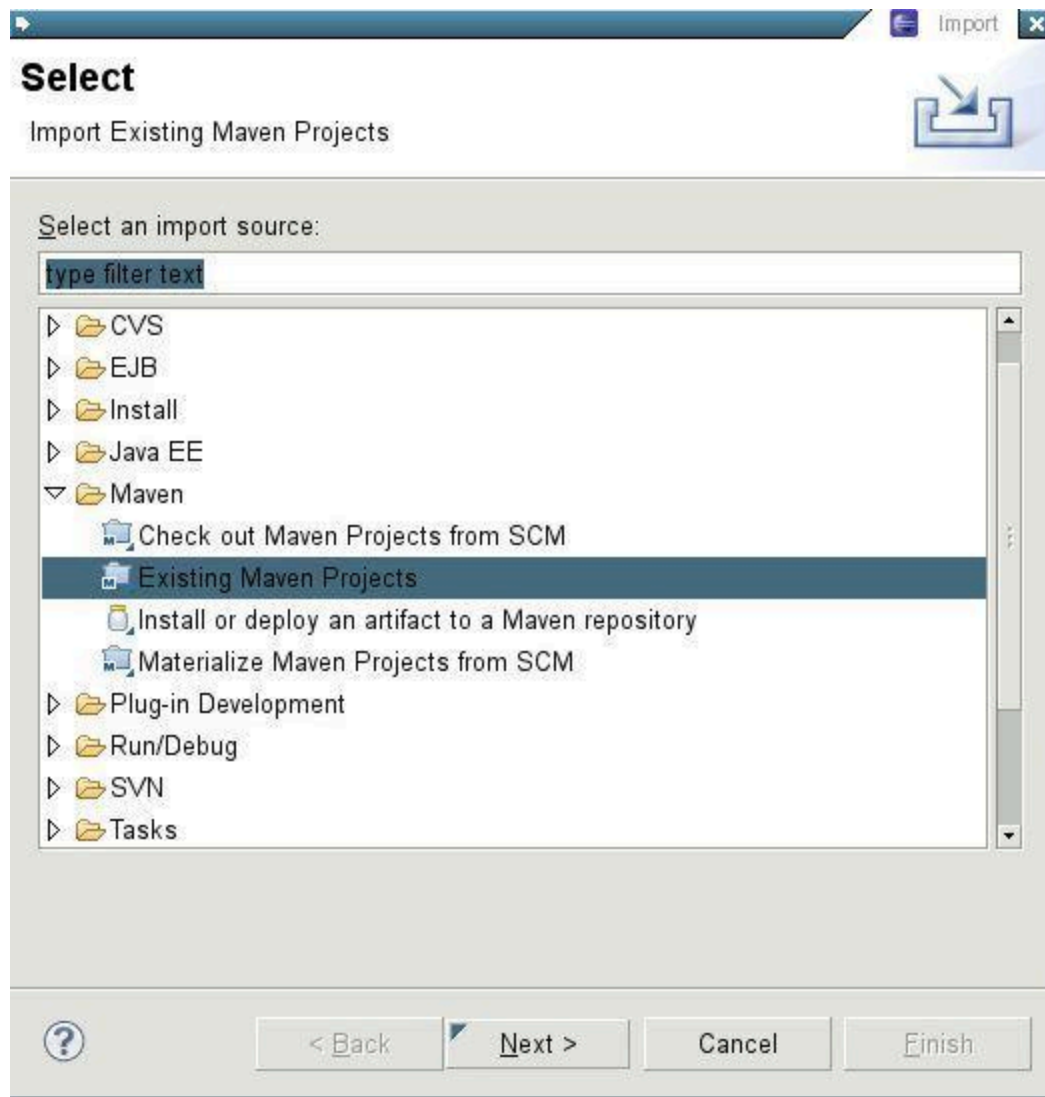
- Create a new Maven project with **File->New->Project...**
- Make sure "Create simple project.." checkbox is not checked
- Select appropriate archetype from kindleit
- Give your project an id and a group



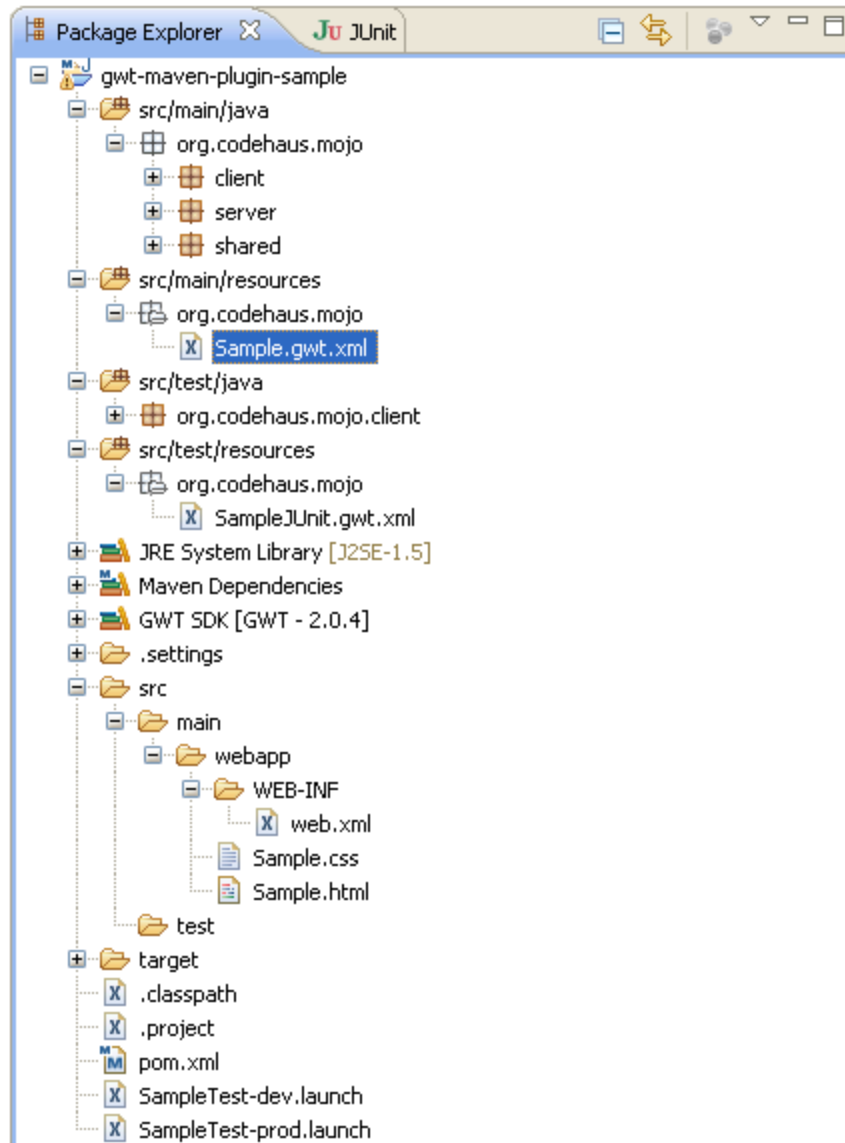
After that it will be generated with proper structure and dependencies

## IMPORT MAVEN PROJECT INTO ECLIPSE

Use **Import->Maven->Existing Maven Projects**



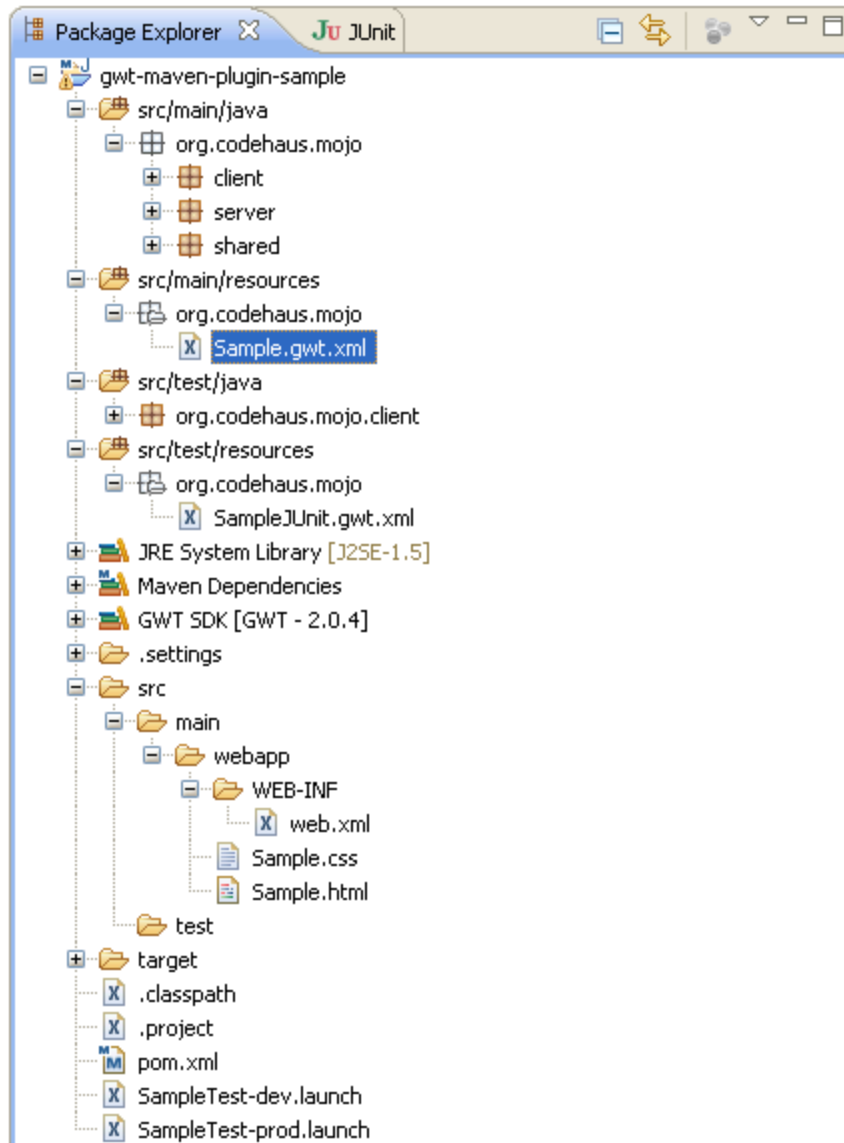
The generated project layout looks like following figure :



It uses M2Eclipse for classpath management and Google Eclipse Plugin for nice GWT support inside Eclipse.

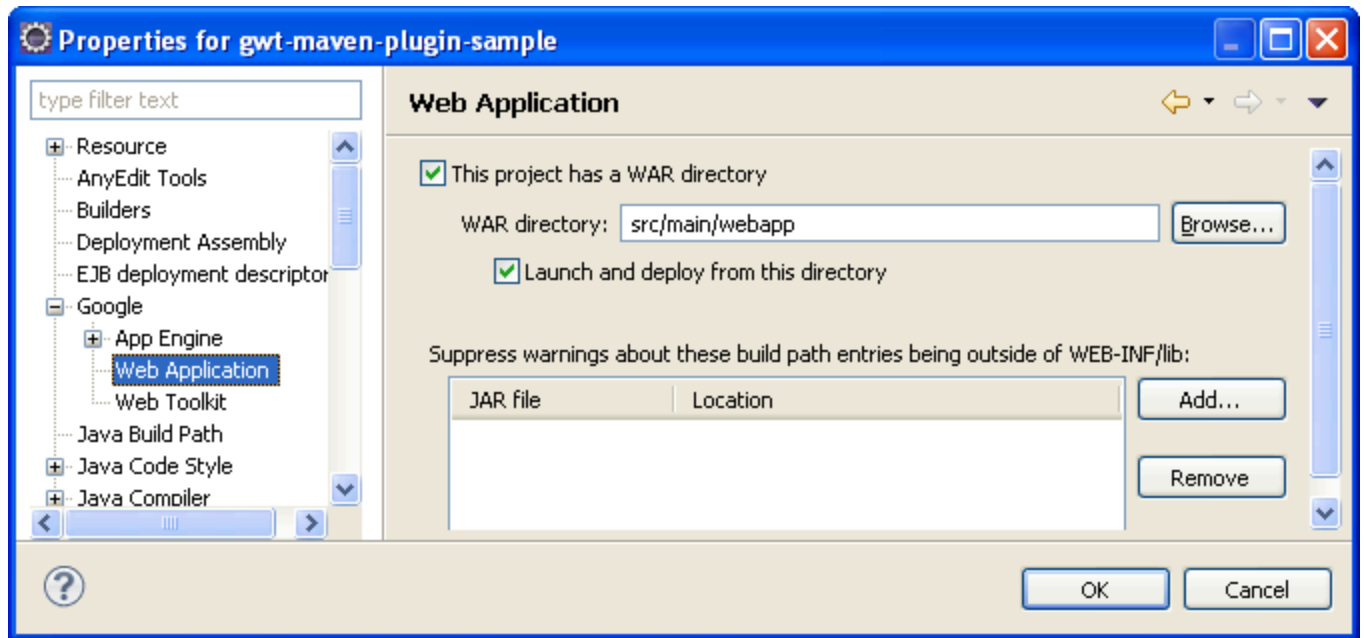
#### PROJECT ORGANIZATION

Compared to [GWT Documentation](#) on directory layout, the plugin follows Maven conventions:



Compared to the directory layout used by GWT, notice that the `/war` folder is replaced by the `src/main/webapp` folder, following [maven-war-plugin](#) conventions. The project structure generated by the `gwt-maven-plugin` archetype already includes the adequate Google Plugin for Eclipse configuration. If you manually migrate a GWT project to Maven, you will need to configure the Google Plugin for Eclipse to use this folder.





## POM CONFIGURATION

In order to use gwt-maven-plugin, you will need to configure it using the `plugins` section of your POM.

You also need to include the GWT dependencies in your POM, and use the adequate gwt-maven-plugin version. The plugin will check this version and warn if you try to use inconsistent releases.

```
<dependencies>

  <dependency>

    <groupId>com.google.gwt</groupId>

    <artifactId>gwt-servlet</artifactId>

    <version>2.5.1</version>

    <scope>runtime</scope>

  </dependency>

  <dependency>

    <groupId>com.google.gwt</groupId>

    <artifactId>gwt-user</artifactId>

    <version>2.5.1</version>
```

```

    <scope>provided</scope>

  </dependency>

</dependencies>

<build>

  <plugins>

    <plugin>

      <groupId>org.codehaus.mojo</groupId>

      <artifactId>gwt-maven-plugin</artifactId>

      <version>2.5.1</version>

      <executions>

        <execution>

          <goals>

            <goal>compile</goal>

            <goal>generateAsync</goal>

            <goal>test</goal>

          </goals>

        </execution>

      </executions>

    </plugin>

  </plugins>

</build>

```

Note : Don't define `gwt-dev` as project dependency : this JAR only contains gwt SDK tools and has many common libraries packaged that may conflict with the ones defined for your project, resulting in uncomprehensible `NoSuchMethodError`s. The gwt-maven-plugin will automatically resolve the required dependency and add it to classpath when launching GWT tools.

Large Maven projects often are divided into sub-projects. This section describe the maven configuration needed to use such layout on GWT projects with gwt-maven-plugin. If you're not familiar with multi-module layout, please read first the [related maven documentation](#).

**NOTE** that GWT also has a notion of [module](#). Both Maven and GWT use the term *module* to define units of modularization. To a degree both concepts go hand in hand, as gwt-modules define boundaries at which Maven-modules might be cut. To not confuse these two terms though, for the rest of this section we will use the term **module**, if we talk about **GWT**-modules, in contrast to the term **project**, if we talk about **Maven**-modules.

First, we will setup a basic Maven project structure consisting of two sub-projects: one containing domain code and another one containing the actual GWT application. Like other web application, a common pattern is to separate GUI functionality from domain functionality (among others) :

```
parent/                                     (aggregating parent project)
|- pom.xml
|
|- domain/                                 (domain code, etc.; packaging: JAR)
|   |- pom.xml
|   \- src/main/java/
|       \- org/codehaus/mojo/domain
|           \- User.java
|
\-- webapp/                               (GUI code; packaging: WAR)
    |- pom.xml
    \- src/
        \- main/java/
            \- org/codehaus/mojo/ui/
                \- Hello.gwt.xml
                \- client/Hello.java
        \- main/webapp/
            \- WEB-INF/web.xml
```

To convert the domain project to a valid GWT module, we add a module descriptor `Domain.gwt.xml` to the `domain` project that we can extend from our webapp `Hello` module.

```
| - domain/
|   | - pom.xml
|   \ - src/main/java/
|       \ - org/codehaus/mojo/domain/
|           \ - User.java
|   \ - src/main/resources/
|       \ - org/codehaus/mojo/
|           \ - Domain.gwt.xml (Additional gwt.xml module file)
<module>
    <inherits name="com.google.gwt.user.User"/>
    <source path="domain"/>
</module>
```

The domain project is not yet a valid GWT module : GWT compiler requires Java source files.

First option is to use the [gwt:resources](#) or [gwt:source-jar](#) goals to attach java sources and resources in the generated jar. The side effect is that this jar will include java sources, that may not be what you want to do if you use it elsewhere, typically in the webapp servlets.

Second option is to configure Maven to [package a source-jar](#) for domain project. In the `webapp` project, configure gwt-maven-plugin to use it as additional source for the domain dependency :

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>gwt-maven-plugin</artifactId>
    <version>2.5.1</version>
    <executions>
        ...
    </executions>
    <configuration>
```

```
<compileSourcesArtifacts>

    <artifact>org.codehaus.mojo:domain</artifact>

</compileSourcesArtifacts>

</configuration>

</plugin>
```

## COMPILE GWT APPLICATION

You can use the following configuration in your pom.xml to run the GWT compiler when the project is built. By default, the [compile](#) goal is configured to be executed during the "prepare-package" phase to run as late as possible.

```
<project>

[... ]

<build>

    <plugins>

        [...]

        <plugin>

            <groupId>org.codehaus.mojo</groupId>

            <artifactId>gwt-maven-plugin</artifactId>

            <version>2.5.1</version>

            <executions>

                <execution>

                    <configuration>

                        <module>com.mycompany.gwt.Module</module>

                    </configuration>

                    <goals>

                        <goal>compile</goal>

                    </goals>

                </execution>

            </executions>

        </plugin>

    </plugins>

</build>

</project>
```

```
        </executions>

        </plugin>

        [...]

    </plugins>

</build>

[...]
```

---

```
</project>
```

---

## CONFIGURE GWT MODULES

The `module` paramter can be used to define a single module in your application. You can also configure compilation for multiple modules by nesting them inside a "*modules*" element. If none is set, the plugin will automatically scan project source and resources directories for ".gwt.xml" module files.

You can also force the plugin to compile a module from command line by setting the `gwt.module` system property.

---

## TWEAK THE COMPILER OUTPUT

By default, the GWT compiler is run with WARN logging. If you have compilation issues, you may want it to be more verbose. Simply add a command line option :

```
-Dgwt.logLevel=[LOGLEVEL]
```

Where LOGLEVEL can be ERROR, WARN, INFO, TRACE, DEBUG, SPAM, or ALL

The compiler style is set to its default value (`OBFUSCATED`) to generate compact javascript. You can override this for debugging purpose of the generated javascript by running with command line option :

```
-Dgwt.style=[PRETTY|DETAILED]
```

The compiler will output the generated javascript in the project output folder (`${project.build.directory}/${project.build.finalName}`). For a WAR project, this matches the exploded web application root. You can also configure the plugin to compile in `${basedir}/src/main/webapp` that may better match using lightweight development process based on to the "[inplace](#)" mode of the war plugin. To enable this, just set the `inplace` parameter to true.

---

## COMPILATION PROCESS FAILING

You may get compilation errors due to `OutOfMemoryException` or `StackOverflowException`. The compilation and permutation process used by GWTCompiler is a high memory consumer, with many recursive steps. You can get rid of those errors by setting the JVM parameters used to create the child process where GWT compilation occurs :

```
<project>

[... ]

<build>

  <plugins>

    [...]

    <plugin>

      <groupId>org.codehaus.mojo</groupId>

      <artifactId>gwt-maven-plugin</artifactId>

      <version>2.5.1</version>

      <executions>

        <execution>

          <configuration>

            <extraJvmArgs>-Xmx512M -Xss1024k</extraJvmArgs>

          </configuration>

          <goals>

            <goal>compile</goal>

          </goals>

        </execution>

      </executions>

    </plugin>

    [...]

  </plugins>

</build>
```

```
[...]  
</project>
```

## COMPILER OUTPUT DIRECTORY

The compile goal is used to run the GWTCompiler and generate the JavaScript application. This mojo can switch between to modes : *standard* or *inplace*.

**Standard** uses simple integration of GWTCompiler in the maven build process and will output in the project build directory where the maven-war-plugin is expected to create the exploded web-application before packaging it as a WAR.

**Inplace** use the web application source directory `src/main/webapp` as output folder. to match the [war:inplace](#) goal. This one setup an exploded WAR structure in the source folder for rapid JEE development without the time-consuming package/deploy/restart cycle.

Using this folder is also very usefull for those of us that run a server using [tomcat7:run](#) or [jetty:run](#) goals. Those plugins don't require any packaging to launch the webapp, and handle nicely Maven dependencies and classes without requirement for a `WEB-INF/lib` and `WEB-INF/classes`. With this default GWTCompiler output directory, the application can be run as is with no packaging requirement.

## TESTING GWT CODE WITH MAVEN

One special aspect of gwt-maven-plugin to be familiar with is that it runs its own special `test` goal in order to support `GWTTestCase` and `GWTTestSuite` derived GWT tests. If you're not familiar with GWT testing support you should first read the related [documentation](#).

We do not consider GWTTestCase to be unit test as they require the whole GWT Module to run. For this reason, the `test` goal is bound by default to `integration-test` phase. But this is only our point of view and you may want to run such test during the standard `test` phase, and maybe use a profile and naming convention to execute such tests on demand.

Another option is to [use the MVP design pattern](#) to keep your code and UI components separated. In such case, GWT [MockUtilities](#) will help you disable Deferred Binding (that requires a full GWT stack) and use mock components.

## USING SUREFIRE

It's a long story as to why a dedicated `gwt:test` is needed, but in few words the regular Maven Surefire testing plugin requires some complex setup to work fine with GWTTestSuites. If you really want to do so, here is a sample configuration :

```
<plugin>  
  
  <artifactId>maven-surefire-plugin</artifactId>
```



```

<version>2.6</version>

<configuration>

  <additionalClasspathElements>

<additionalClasspathElement>${project.build.sourceDirectory}</additionalClasspathElement>

<additionalClasspathElement>${project.build.testSourceDirectory}</additionalClasspathElement>

  </additionalClasspathElements>

  <useManifestOnlyJar>false</useManifestOnlyJar>

  <forkMode>always</forkMode>

  <systemProperties>

    <property>

      <name>gwt.args</name>

      <value>-out ${webAppDirectory}</value>

    </property>

  </systemProperties>

</configuration>

</plugin>

```

---

## USING GWT-MAVEN-PLUGIN TEST GOAL

The gwt-maven-plugin testing support is **not** intended to be run standalone, rather it is bound to the Maven `integration-test` phase. To get `gwt:test` to run, you should include the `test` goal in your plugin configuration executions, and you should invoke `mvn verify` (or `mvn install`).

```

<project>

  [...]

  <build>

    <plugins>

      [...]

```

```

<plugin>

  <groupId>org.codehaus.mojo</groupId>

  <artifactId>gwt-maven-plugin</artifactId>

  <version>2.5.1</version>

  <executions>

    <execution>

      <goals>

        <goal>test</goal>

      </goals>

    </execution>

  </executions>

</plugin>

[...]

</plugins>

</build>

[...]

</project>

```

---

## SEPARATE GWT TEST TESTS FROM STANDARD UNIT TESTS

Because you will likely want to run **both** Surefire and gwt-maven-plugin based tests (for regular server side JUnit tests with Surefire, and for client model and controller tests with GWT) you need to distinguish these tests from each other. This is done using a naming convention.

You can configure the Surefire plugin (responsible for running tests during maven build) to skip the GwtTests using some naming patterns :

```

<plugin>

  <artifactId>maven-surefire-plugin</artifactId>

  <version>2.6</version>

  <configuration>

```

```

    <excludes>

        <exclude>**/*GwtTest.java</exclude>

    </excludes>

</configuration>

</plugin>

```

A simpler way to separate classic and GWT tests is to name latests `GwtTestSomething.java`. As surefire looks for tests that are named `SomethingTest.java` by default, they will be ignored during `test` phase.

By default, the gwt-maven-plugin uses `GwtTest*.java` as inclusion pattern so that such testst will **not** match the standard Surefire pattern. Using this convention you don't have to change your configuration.

To configure the plugin to use another naming convention, set the `includes` plugin parameter.

```

<plugin>

    <groupId>org.codehaus.mojo</groupId>

    <artifactId>gwt-maven-plugin</artifactId>

    <version>2.5.1</version>

    <configuration>

        <includes>**/CustomPattern*.java</includes>

    </configuration>

    <executions>

        <execution>

            <goals>

                <goal>test</goal>

            </goals>

        </execution>

    </executions>

</plugin>

```

The plugin uses the standard Surefire testrunner, and contributes to execution report. If you use the surefire-report-plugin, you will see the GwtTests result included in generated project site.

---

## ABOUT GWTTESTSUITE

The [Official GWT documentation](#) on TestSuites suggests to grouping your GWT tests in a suite by extending GwtTestSuite. Maven will not be able to run such a Suite. The snippet below is what will allow both the Eclipse JUnit Runner and Maven to run the test cases.

```
public class MyGwtTestSuite extends TestCase /*note this is TestCase and not
TestSuite */

{

    public static Test suite()

    {

        GwtTestSuite suite = new GwtTestSuite( "All Gwt Tests go in here" );

        suite.addTestSuite( GwtTest1.class );

        return suite;

    }

}
```

---

## USE GWTTESTSUITE PADAWAN

GWTTestCase derived tests are slow. This is because the JUnitShell has to load the module for each test (create the shell, hook into it, etc). GWTTestSuite mitigates this by grouping all the tests that are for the same module (those that return the same value for getModuleName) together and running them via the same shell instance.

This is a **big** time saver, and GWTTestSuite is easy to use, so using it is a good idea. For this reason, the default value of the test inclusion pattern is `**/Gwt*Suite.java`.

We recommend to name your test suite `GwtTestSuite.java` so that the test filter picks it up, but name the actual tests with a convention that Surefire will ignore by default - something that **does not** start with `GwtTest`, and does **not** start or end with `Test`. For example `MyClassTestGwt.java`. This way, gwt-maven-plugin picks up the `Suite`, and runs it, but does not also run individual tests (and Surefire does not pick it up either)

---

## TESTING MODES

GWTTestCase uses HTMLUnit to run your code. [HTMLUnit](#) doesn't provide a full browser with GUI components, but is 100% Java based so it doesn't require a native process and can run in your favorite debugger. Please note this is only emulation, always validate test results with a real browser.

By default, tests run in DevMode. As an option, you can run your code compiled to JavaScript instead of using the dev mode bridge between Java and JavaScript, using the `productionMode` parameter. GWT Team suggest that you run your tests in both modes to detect [differences between Java and JavaScript](#). You may configure your continuous integration server with `-Dgwt.test.prod=true` system property to check this.

#### TESTING WITH "REAL" BROWSERS

Running headless is usefull, but when it fails you'll need to see what happen and use your own browser. In such case, use the manual mode using the `mode` parameter (or `gwt.test.mode` system property) to "manual". GWT will then print a URL to access from your browser.

You may also want to test your code on various browser, not just the hosted mode embeded one. You then need a remote browser available for testing. See [GWT documentation](#) for more explanations.

Selenium mode can be used for running your test suite on real browsers using Selenium RC. Set the `mode` parameter (or `gwt.test.mode` system property) to "selenium". Don't forget to configure the `selenium` parameter with the host running the Selenium RC browser:

```
<plugin>

  <groupId>org.codehaus.mojo</groupId>

  <artifactId>gwt-maven-plugin</artifactId>

  <version>2.5.1</version>

  <configuration>

    <selenium>myhost:4444/*firefox"</selenium>

  </configuration>

</plugin>
```

Selenium may not fit your needs, then GWT comes with a custom RemoteBrowserServer. You can lauch such a server using `gwt:browser` goal, by default it will run Internet Explorer on a Windows box as "ie". You can then run your testsuite on this remote browser, by setting the the `mode` parameter (or `gwt.test.mode` system property) to "remoteweb". You'll have to configure the `remoteweb` parameter to declare your remoteBrowserServer:

```
<plugin>

  <groupId>org.codehaus.mojo</groupId>

  <artifactId>gwt-maven-plugin</artifactId>

  <version>2.5.1</version>

  <configuration>
```

```
<remoteweb>rmi://myhost/ie</remoteweb>

</configuration>

</plugin>
```

## CONFIGURING DevMode

Another aspect of gwt-maven-plugin to be familiar with it can help you configure the embedded Servlet container GWT uses in DevMode. The standard **src/main/webapp** webapp folder is used by gwt-maven-plugin to run the dev mode server (Jetty).

## USING ECLIPSE

The Google Plugin for Eclipse automatically handles launching DevMode with a simple right click on your module **gwt.xml** file "run as > web application".

## USING COMMAND LINE

You can use `mvn gwt:run` to launch DevMode from command line. In such case, the plugin will compile your classes and prepare the exploded webapp structure. DevMode is then launched on your webapp.

## COMFORTABLE DEBUGGING WITH GWT, MAVEN AND ECLIPSE

Needed tools:

- Eclipse
- maven
- m2eclipse plugin
- Google Plugin for Eclipse

## COMPLEX STRUCTURE OF MULTIMODULE MAVEN PROJECT

The tutorial will use sample structure to describe setup of whole project in Eclipse with possibility to automatically propagate change in source files of dependent modules to final GUI. The sample structure is prepared for extending with new use cases.

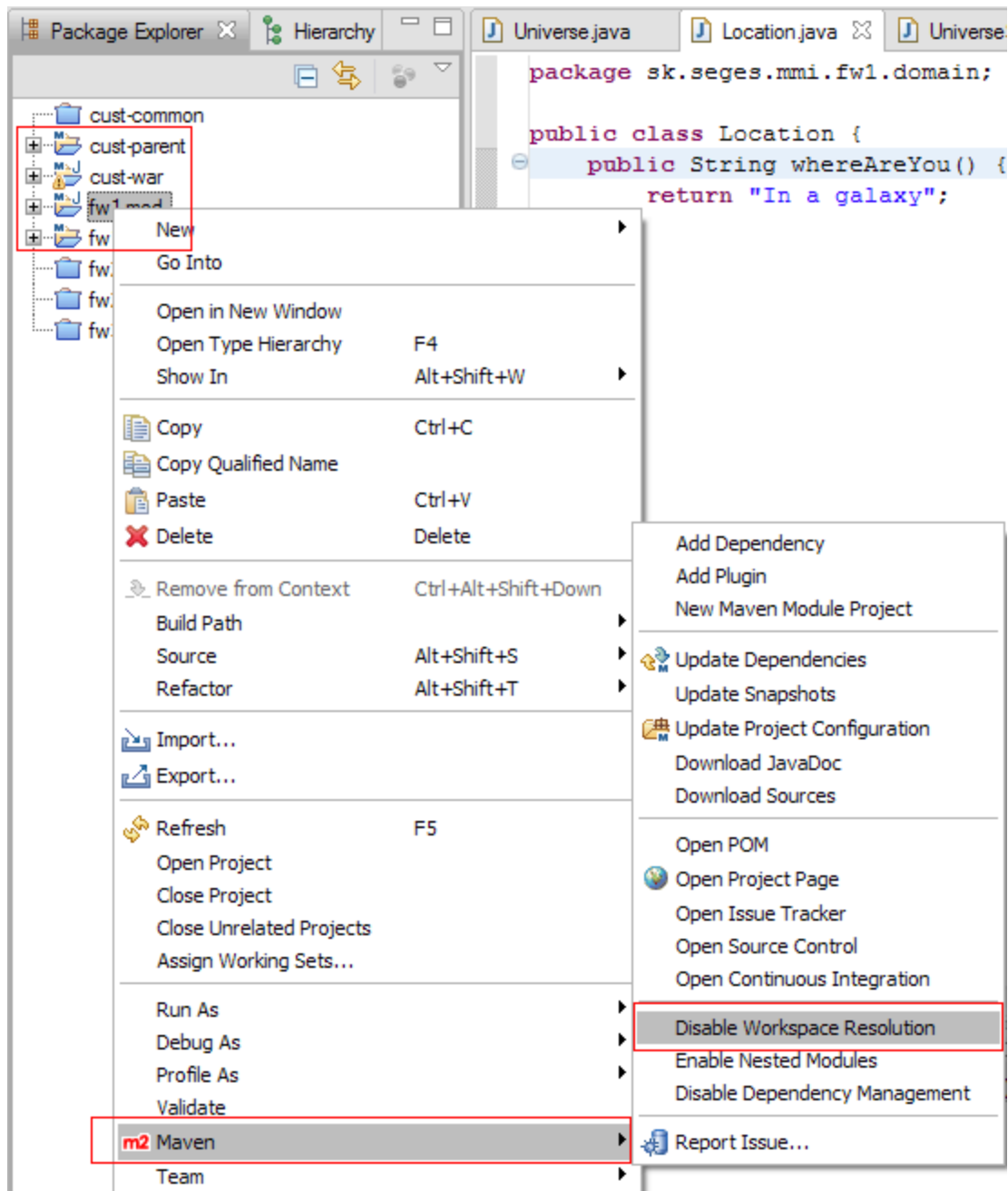
Let's consider simplified scenario of one framework module and one customer's web portal.

- Framework module (fw1-mod) can be used by different portals in combination with different modules
- Framework module is developed by us and it is reflecting common behaviour used in these portals
- Customer's portal (cust-war) has also it's own shared module (cust-common) of common customer's behaviour
- Framework module is more generic, customer shared module is specific only to that customer

- framework module is located in `java/fw1/fw1-mod`
- customer's portal is located in `web/cust-portal/cust-war`

This step is common for both alternatives - import projects into Eclipse using m2eclipse facilities.

- File -> Import -> Maven -> Existing Maven Projects
- import fw1-mod, fw1-parent, cust-parent and cust-war
- ensure that Workspace resolution is enabled



---

## GENERATING LAUNCH CONFIGURATION

Steps needed to generate one of the launch configurations are as follows:

- `mvn gwt:eclipse`
- there will be a launch configuration generated e.g. `sk.seges.mmi.cust.Universe.launch`
- Just search the launch configuration in Eclipse and run it. All dependencies are resolved automatically and the GPE is copying a change transparently. If you take a look on the classpath GPE has it will be the same as the one you would setup for Java App. launch configuration.

You can now do the following:

- run dev mode with the launcher
- modify e.g. a string in Location class in fw1-mod
- press `Ctrl+S` in Eclipse
- press Refresh button in your browser
- voila! change appears

---

## PRODUCTIVITY TIP FOR MULTI-PROJECT SETUP

---

## INTRODUCTION

Consider the following project setup

- api (jar) - GWT library
- gui (war) - GWT gui with endpoint that uses the library.

With this layout, any change to the api (gwt library) must be repackaged as a JAR and the hosted mode must be restarted to see the change in the hosted browser.

The following tip explains how to use the build-helper-maven-plugin to improve productivity and hack the multi-project wall between modules.

---

## BUILD HELPER

[Build-helper-maven-plugin](#) allow you to setup additional source folders for your project. The idea here is to declare the api source folder to make it "visible" from the war project / hosted mode browser.

If you add a source path with the build-helper-maven-plugin directly in the gui's pom you will possibly have problems because of 2 issues.

- At least my IDE (Netbeans) cannot have two open projects that share the same source path. The api module will loose its src/java in the user interface, and the gui will get one ekstra "generated sources" path, this is quite annoying.



- Because there is no guarantee on how the developer will checkout the code, the gui's pom cannot guess where the api's `src/main/java` is on the disk.

---

## SOLUTION

The solution to those two issues is to create a profile in your pom which you'd only activate when you run the `gwt:run` target:

```
<profile>

  <id>dev</id>

  <build>

    <plugins>

      <plugin>

        <groupId>org.codehaus.mojo</groupId>

        <artifactId>build-helper-maven-plugin</artifactId>

        <version>1.4</version>

        <executions>

          <execution>

            <id>add-source</id>

            <phase>generate-sources</phase>

            <goals>

              <goal>add-source</goal>

            </goals>

            <configuration>

              <sources>

                <source>../api/src/main/java</source>

              </sources>

            </configuration>

          </execution>
```

```

    <execution>

      <id>add-resource</id>

      <phase>generate-sources</phase>

      <goals>

        <goal>add-resource</goal>

      </goals>

      <configuration>

        <resources>

          <resource>

            <directory>../api/src/main/resources</directory>

            <targetPath>resources</targetPath>

          </resource>

        </resources>

      </configuration>

    </execution>

  </executions>

</plugin>

</plugins>

</build>

</profile>

```

You can then test in development mode and edit files in multiple projects by running:

[mvn gwt:run -Pdev](#)

In Netbeans it is possible to save such a run target in the user interface.

#### DEV MODE WITH APPENGINE LAUNCHER

With version 2.1.1, you can now start debug mode with AppEngineLauncher. Simple and default configuration. This one will download and extract appengine sdk. The appengine-sdk version used is defined with a mojo property [appEngineVersion](#). You can change in the cli with -Dgwt.appEngineVersion= :

```

<project>

  [...]

  <build>

    <plugins>

      [...]

      <plugin>

        <groupId>org.codehaus.mojo</groupId>

        <artifactId>gwt-maven-plugin</artifactId>

        <version>2.5.1</version>

        <configuration>

<server>com.google.appengine.tools.development.gwt.AppEngineLauncher</server>

          </configuration>

        </plugin>

      [...]

    </plugins>

  </build>

  [...]

</project>

```

You can use an already locally installed appengine-sdk. The recommended way is to use a property and to set the value in your settings.xml

```

<project>

  [...]

  <build>

    <plugins>

      [...]

      <plugin>

```

```

    <groupId>org.codehaus.mojo</groupId>

    <artifactId>gwt-maven-plugin</artifactId>

    <version>2.5.1</version>

    <configuration>

        <appEngineHome>${appEngineSdk}</appEngineHome>

    </configuration>

</plugin>

[...]

</plugins>

</build>

[...]

</project>

```

## USING THE GOOGLE ECLIPSE PLUGIN

The [Google Plugin for Eclipse](#) is a nice integration of GWT inside Eclipse to make development easier. It can be used to launch the DevMode with a simple right click and provides several wizards and tools, and optionally the GWT Designer.

## PROJECT LAYOUT

Your maven project will end something like this. Please note the `Module.gwt.xml` module descriptor located in `src/main/java` directory :

```

pom.xml

|_src
  |_main
    |_java
      |_ com/mycompany/gwt/Module.gwt.xml
      |_ com/mycompany/gwt/client
      |_ |_ ModuleEntryPoint.java

```

```
|_resources

|_webapp

| index.html

|_WEB-INF

|_web.wml
```

---

## MAVEN CONFIGURATION

Your Maven configuration will be something like this :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                              http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany</groupId>

  <artifactId>webapp</artifactId>

  <packaging>war</packaging>

  <version>1.0-SNAPSHOT</version>

  <dependencies>

    <dependency>

      <groupId>com.google.gwt</groupId>

      <artifactId>gwt-user</artifactId>

      <scope>provided</scope>

    </dependency>

    <dependency>

      <groupId>com.google.gwt</groupId>

      <artifactId>gwt-servlet</artifactId>
```

```
<scope>runtime</scope>

</dependency>

</dependencies>

<build>

  <plugins>

    <plugin>

      <groupId>org.codehaus.mojo</groupId>

      <artifactId>gwt-maven-plugin</artifactId>

      <version>2.5.1</version>

      <executions>

        <execution>

          <goals>

            <goal>compile</goal>

          </goals>

        </execution>

      </executions>

      <configuration>

        <runTarget>index.html</runTarget>

      </configuration>

    </plugin>

  </plugins>

</build>

</project>
```

With this setup, you can start your GWT module with a single right-click in your Eclipse IDE with *Run as : Web application*.

You can the edit your java code and just hit refresh to see changes applied in the browser.

---

## ECLIPSE CONFIGURATION

Import your maven project into eclipse using m2eclipse import wizard (or your preferred tooling). The Google Plugin for Eclipse should automatically enable the GWT nature on the project. If not, manually enable it from project preferences by setting the `Use Google Web Toolkit` checkbox (the GWT SDK should be picked from your POM).

---

## MULTIPROJECT SETUP

Big projects may want to split the client-side application in modules, typically using Maven support for multiprojects. The project layout will become something like this (maybe with some more classes) :

```
pom.xml // reactor project

|_domain // shared with other (GWT or Java) projects
|   |_src
|       |_main
|           |_java
|               |_ com/mycompany/domain
|                   |_ User.java
|                       |_resources
|                           |_ com/mycompany/Domain.gwt.xml
|_webapp
|   |_src
|       |_main
|           |_java
|               |_ com/mycompany/gwt/Module.gwt.xml // inherits Domain.gwt.xml
|                   |_ com/mycompany/gwt/client
|                       |_ ModuleEntryPoint.java
...

```

When using Eclipse-Maven integration like the m2eclipse plugin, other maven projects open in the workspace will be automatically resolved as projects (instead of JARs). When the referenced project is well configured (\*) as a

GWT module project, changes to java sources will be available in DevMode with a simple refresh with no requirement to repackage the modules.



(\*) A "*well configured GWT module project*" is expected to have Java sources copied as resources in the project build outputDirectory (using gwt:resource goal) and a dedicated gwt.xml module file to define the required inherits.

The gwt-maven-plugin [src/it/reactor](#) project can be reviewed as a demonstrating sample of this setup.



## GWT LIBRARY MODULES

A GWT library can be used to package classes for mutualization and/or modularization. A GWT library is just a java archive (JAR) containing both classes and java sources for later GWT compilation and a gwt.xml module descriptor.

GWT projects can be organized in a variety of ways. However, particular conventions are encouraged to make it easy to identify which code is intended to run on the client browser, the server, or both.

This section describes the fundamentals of project organization with GWT as well as the recommended conventions.

## HTML HOST PAGES

GWT modules are stored on a web server as a set of JavaScript and related files. In order to run the module, it must be loaded from a web page of some sort. Any HTML page can include a GWT application via a `SCRIPT` tag. This HTML page is referred to as a *host page* from the GWT application's point of view. A typical HTML host page for an application written with GWT from scratch might not include any visible HTML body content at all. The example below shows how to embed a GWT application that will use the entire browser window.

```
<html>
<head>

  <!-- Properties can be specified to influence deferred binding -->
  <meta name='gwt:property' content='locale=en_UK'>

  <!-- Stylesheets are optional, but useful -->
  <link rel="stylesheet" href="Calendar.css">

  <!-- Titles are optional, but useful -->
  <title>Calendar App</title>

</head>
<body>

  <!-- This script tag is what actually loads the GWT module. The -->
  <!-- 'nocache.js' file (also called a "selection script") is -->
  <!-- produced by the GWT compiler in the module output directory -->
  <!-- or generated automatically in development mode. -->
  <script language="javascript" src="calendar/calendar.nocache.js"></script>

  <!-- Include a history iframe to enable full GWT history support -->
  <!-- (the id must be exactly as shown) -->
  <iframe src="javascript:\"" id="__gwt_historyFrame" style="width:0;height:0;border:0"></iframe>

</body>
</html>
```

Note that the body of the page contains only a `SCRIPT` tag and an `IFRAME` tag. It is left to the GWT application to then fill in all the visual content.

But GWT was designed to make it easy to add GWT functionality to existing web applications with only minor changes. It is possible to allow the GWT module to selectively insert widgets into specific places in an HTML page. To accomplish this, use the `id` attribute in your HTML tags to specify a unique identifier that your GWT code will use to attach widgets to that HTML element. For example:

```
<body>
  <!-- ... other sample HTML omitted -->
  <table align=center>
    <tr>
      <td id="slot1"></td>
      <td id="slot2"></td>
    </tr>
  </table>
</body>
```

Notice that the `td` tags include an `id` attribute associated with them. This attribute is accessible through the `DOM` class. You can easily attach widgets using the method `RootPanel.get()`. For example:

```
final Button button = new Button("Click me");
final Label label = new Label();

...

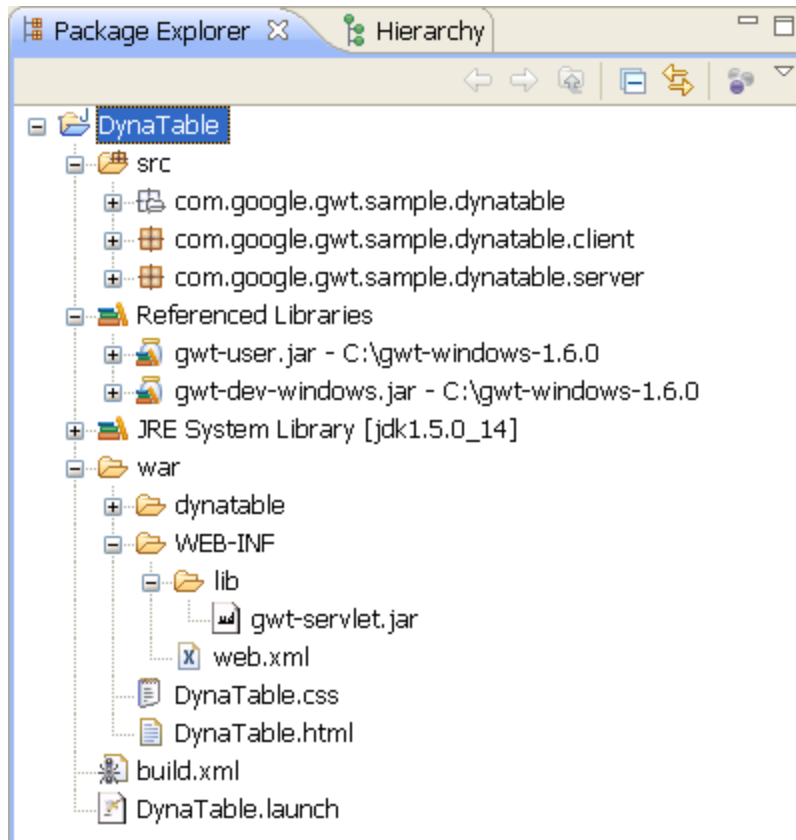
RootPanel.get("slot1").add(button);
RootPanel.get("slot2").add(label);
```

In this manner, GWT functionality can be added as just a part of an existing page, and changing the application layout can be done in plain HTML. The `118N` sample uses this technique heavily.

A host HTML page does not have to be static content. It could also be generated by a servlet, or by a JSP page.

## STANDARD DIRECTORY AND PACKAGE LAYOUT

GWT projects are overlaid onto Java packages such that most of the configuration can be inferred from the classpath and the [module definitions](#).



---

## GUIDELINES

If you are not using the Command-line tools to generate your project files and directories, here are some guidelines to keep in mind when organizing your code and creating Java packages.

1. Under the main project directory create the following directories:
2. `src` folder - contains production Java source
3. `war` folder - your web app; contains static resources as well as compiled output
4. `test` folder - (optional) JUnit test code would go here
5. Within the `src` package, create a project root package and a client package.
6. If you have server-side code, also create a server package to differentiate between the client-side code (which is translated into JavaScript) from the server-side code (which is not).
7. Within the project root package, place one or more module definitions.
8. In the `war` directory, place any static resources (such as the host page, style sheets, or images).
9. Within the client and server packages, you are free to organize your code into any subpackages you require.

---

## EXAMPLE: GWT STANDARD PACKAGE LAYOUT

For example, all the files for the "DynaTable" sample are organized in a main project directory also called "DynaTable".

- Java source files are in the directory: `DynaTable/src/com/google/gwt/sample/dynatable`
- The module is defined in the XML file: `DynaTable/src/com/google/gwt/sample/dynatable/DynaTable.gwt.xml`
- The project root package is: `com.google.gwt.sample.dynatable`
- The logical module name is: `com.google.gwt.sample.dynatable.DynaTable`

#### THE SRC DIRECTORY

The src directory contains an application's Java source files, the module definition, and external resource files.

Package	File	Purpose
<code>com.google.gwt.sample.dynatable</code>		The project root package contains module XML files.
<code>com.google.gwt.sample.dynatable</code>	<code>DynaTable.gwt.xml</code>	Your application module. Inherits <code>com.google.gwt.user.User</code> and adds an entry point class, <code>com.google.gwt.sample.dynatable.client.DynaTable</code> .
<code>com.google.gwt.sample.dynatable</code>		Static resources that are loaded programmatically by GWT code. Files in the public directory are copied into the same directory as the GWT compiler output.
<code>com.google.gwt.sample.dynatable</code>	<code>logo.gif</code>	An image file available to the application code. You might load this file programmatically using this URL: <code>GWT.getModuleBaseURL() + "logo.gif"</code> .
<code>com.google.gwt.sample.dynatable.client</code>		Client-side source files and subpackages.
<code>com.google.gwt.sample.dynatable.client</code>	<code>DynaTable.java</code>	Client-side Java source for the entry-point class.

<code>com.google.gwt.sample.dynatable.client</code>	<code>SchoolCalendarService.java</code>	An RPC service interface.
<code>com.google.gwt.sample.dynatable.server</code>		Server-side code and subpackages.
<code>com.google.gwt.sample.dynatable.server</code>	<code>SchoolCalendarServiceImpl.java</code>	Server-side Java source that implements the logic of the service.

## THE WAR DIRECTORY

The war directory is the deployment image of your web application. It is in the standard expanded war format recognized by a variety of Java web servers, including Tomcat, Jetty, and other J2EE servlet containers. It contains a variety of resources:

- Static content you provide, such as the host HTML page
- GWT compiled output
- Java class files and jar files for server-side code
- A `web.xml` file that configures your web app and any servlets

A detailed description of the war format is beyond the scope of this document, but here are the basic pieces you will want to know about:

Directory	File	Purpose
<code>DynaTable/war/</code>	<code>DynaTable.html</code>	A host HTML page that loads the DynaTable app.
<code>DynaTable/war/</code>	<code>DynaTable.css</code>	A static style sheet that styles the DynaTable app.
<code>DynaTable/war/dynatable/</code>		The DynaTable module directory where the GWT compiler writes output and files on the public path are copied. NOTE: by default this directory would be the long, fully-qualified module name <code>com.google.gwt.sample.dynatable.DynaTable</code> . However, in our GWT module XML file we used the <code>rename-to="dynatable"</code> attribute to shorten it to a nice name.
<code>DynaTable/war/dynatable/</code>	<code>dynatable.nocache.js</code>	The "selection script" for DynaTable. This is the script that must be loaded from the host HTML to load the GWT module into the page.

DynaTable/war/WEB-INF		All non-public resources live here, see the servlet specification for more detail.
DynaTable/war/WEB-INF	web.xml	Configures your web app and any servlets.
DynaTable/war/WEB-INF/classes		Java compiled class files live here to implement server side functionality. If you're using an IDE set the output directory to this folder.
DynaTable/war/WEB-INF/lib		Any library dependencies your server code needs goes here.
DynaTable/war/WEB-INF/lib	gwt-servlet.jar	If you have any servlets using GWT RPC, you will need to place a copy of gwt-servlet.jar here.

#### THE TEST DIRECTORY

The test directory contains the source files for any JUnit tests.

Package	File	Purpose
com.google.gwt.sample.dynatable.client		Client-side test files and subpackages.
com.google.gwt.sample.dynatable.client	DynaTableTest.java	Test cases for the entry-point class.
com.google.gwt.sample.dynatable.server		Server-side test files and subpackages.

com.google.gwt.sample.dynatable.server	SchoolCalendarServiceImplTest.java	Test cases for server classes.
--	------------------------------------	--------------------------------

## MODULES: UNITS OF CONFIGURATION

Individual units of GWT configuration are called *modules*. A module bundles together all the configuration settings that your GWT project needs:

- inherited modules
- an entry point application class name; these are optional, although any module referred to in HTML must have at least one entry-point class specified
- source path entries
- public path entries
- deferred binding rules, including property providers and class generators

Modules are [defined in XML](#) and placed into your [project's package hierarchy](#). Modules may appear in any package in your classpath, although it is strongly recommended that they appear in the root package of a [standard project layout](#).

## ENTRY-POINT CLASSES

A module entry-point is any class that is assignable to [EntryPoint](#) and that can be constructed without parameters. When a module is loaded, every entry point class is instantiated and its [EntryPoint.onModuleLoad\(\)](#) method gets called.

## SOURCE PATH

Modules can specify which subpackages contain translatable *source*, causing the named package and its subpackages to be added to the *source path*. Only files found on the source path are candidates to be translated into JavaScript, making it possible to mix [client-side](#) and [server-side](#) code together in the same classpath without conflict. When module inherit other modules, their source paths are combined so that each module will have access to the translatable source it requires.

The default source path is the *client* subpackage underneath where the [Module XML File](#) is stored.

## PUBLIC PATH

Modules can specify which subpackages are *public*, causing the named package and its subpackages to be added to the *public path*. The public path is the place in your project where static resources referenced by your GWT module, such as CSS or images, are stored. When you compile your application into JavaScript, all the files that can be found on your public path are copied to the module's output directory. When referencing public resources in client code (for example, setting the URL of an `Image` widget, you should construct the URL like

this: `GWT.getModuleBaseURL() + "resourceName.png"`. When referencing public resources from a [Module XML File](#), just use the relative path within the public folder, the module's base URL will be prepended automatically. When a module inherits other modules, their public paths are combined so that each module will have access to the static resources it expects.

The default public path is the *public* subdirectory underneath where the [Module XML File](#) is stored.

#### DEFINING A MODULE: FORMAT OF MODULE XML FILES

Modules are defined in XML files with a file extension of *.gwt.xml*. Module XML files should reside in your project's root package.

If you are using the [standard project structure](#), your module XML can be as simple as the following example:

```
<module rename-to="dynatable">
  <inherits name="com.google.gwt.user.User" />
  <entry-point class="com.google.gwt.sample.dynatable.client.DynaTable" />
</module>
```

#### LOADING MODULES

Module XML files are found on the Java classpath. Modules are always referred to by their logical names. The logical name of a module is of the form *pkg1.pkg2.ModuleName* (although any number of packages may be present). The logical name includes neither the actual file system path nor the file extension.

For example, if the module XML file has a file name of...

```
~/src/com/example/cal/Calendar.gwt.xml
```

...then the logical name of the module is:

```
com.example.cal.Calendar
```

#### RENAMING MODULES

The `<module>` element supports an optional attribute `rename-to` that causes the compiler to behave as though the module had a different name than the long, fully-qualified name. Renaming a module has two primary use cases:

- to have a shorter module name that doesn't reflect the actual package structure, this is the most typical and recommended use case
- to create a "working module" to speed up development time by restricting the number of permutations



- `com.foo.WorkingModule.gwt.xml`:

```
<module rename-to="com.foo.MyModule">
  <inherits name="com.foo.MyModule" />
  <set-property name="user.agent" value="ie6" />
  <set-property name="locale" value="default" />
</module>
```

When `WorkingModule.gwt.xml` is compiled, the compiler will produce only an `ie6` variant using the default locale; this will speed up development compilations. The output from the `WorkingModule.gwt.xml` will be a drop-in replacement for `MyModule.gwt.xml` because the compiler will generate the output using the alternate name. (Of course, if `com.foo.MyModule` was itself renamed, you would just copy its `rename-to` attribute.)

---

#### DIVIDING CODE INTO MULTIPLE MODULES

Creating a second module doesn't necessarily mean that that module must define an entry point. Typically, you create a new module when you want to package up a library of GWT code that you want to reuse in other GWT projects. An example of this is the Google API Library for Google Web Toolkit ([GALGWT](#)), specifically the Gears for GWT API binding. If you download the library and take a look at the `gwt-google-apis/com/google/gwt/gears` you'll find the `Gears.gwt.xml` file for the module which doesn't define an entry point. However, any GWT project that would like to use Gears for GWT will have to inherit the `Gears.gwt.xml` module. For example, a module named "Foo" might want to use GALGWT, so in `Foo.gwt.xml` an `<inherits>` entry would be needed:

```
<module>
...
  <inherits name='com.google.gwt.gears.Gears' />
```

---

#### LOADING MULTIPLE MODULES IN AN HTML HOST PAGE

If you have multiple GWT modules in your application, there are two ways to approach loading them.

1. Compile each module separately and include each module with a separate `<script>` tag in your [HTML host page](#).
2. Create a top level module XML definition that includes all the modules you want to include. Compile the top level module to create a single set of JavaScript output.

The first approach may seem the easiest and most obvious. However, the second approach will lead to much better end-user performance. The problem with loading multiple modules is that each module has to be downloaded separately by the end-user's browser. In addition, each module will contain redundant copies of GWT library code and could possibly conflict with each other during event handling. The second approach is strongly recommended.

---

#### CONTROLLING COMPILER OUTPUT

The GWT compiler separates the act of compiling and packaging its output with the Linker subsystem. It is responsible for the final packaging of the JavaScript code and providing a pluggable bootstrap mechanism for any particular deployment scenario.

- `<define-linker name="short_name" class="fully_qualified_class_name" />` : Register a new Linker instance with the compiler. The `name` attribute must be a valid Java identifier and is used to identify the Linker in `<add-linker>` tags. It is permissible to redefine an already-defined Linker by declaring a new `<define-linker>` tag with the same name. Linkers are divided into three categories, PRE, POST, and PRIMARY. Exactly one primary linker is run for a compilation. Pre-linkers are run in lexical order before the primary linker, and post-linkers are run in reverse lexical order after the primary linker.
- `<add-linker name="linker_name" />` : Specify a Linker to use when generating the output from the compiler. The `name` property is a previously-defined Linker name. This tag is additive for pre- and post-linkers; only the last primary linker will be run.

Several linkers are provided by `Core.gwt.xml`, which is automatically inherited by `User.gwt.xml`.

- **std** : The standard iframe-based bootstrap deployment model.
- **xs** : The cross-site deployment model.
- **sso** : This Linker will produce a monolithic JavaScript file. It may be used only when there is a single distinct compilation result.

From `Core.gwt.xml`:

```
<module>
  <define-linker name="std" class="com.google.gwt.dev.linker.IFrameLinker" />
  <define-linker name="sso" class="com.google.gwt.dev.linker.SingleScriptLinker" />
  <define-linker name="xs" class="com.google.gwt.dev.linker.XSLinker" />
  <add-linker name="std" />
</module>
```

Changing the desired linker in `MyModule.gwt.xml`:

```
<module>
  <inherits name="com.google.gwt.core.Core" />
  <add-linker name="xs" />
</module>
```

---

## OVERRIDING ONE PACKAGE IMPLEMENTATION WITH ANOTHER

The `<super-source>` tag instructs the compiler to "re-root" a source path. This is useful for cases where you want to re-use an existing Java API for a GWT project, but the original source is not available or not translatable. A common reason for this is to emulate part of the JRE not implemented by GWT.

For example, suppose you want to implement the `UUID` class provided by the JRE under `java.util`. Assume your project's module file is `com/example/myproject/MyProject.gwt.xml`. Place the source for the `UUID` class into `com/example/myproject/jre/java/util/UUID.java`. Then add a line to `MyProject.gwt.xml`:

```
<super-source path="jre" />
```

This tells the compiler to add all subfolders of `com/example/myproject/jre/` to the [source path](#), but to strip off the path prefix up to and including `jre`. As a result, `com/google/myproject/gwt/jre/java/util/UUID.java` will be visible to the compiler as `java/util/UUID.java`, which is the intended result.

The GWT project uses this technique internally for the JRE emulation classes provided with GWT. One caveat specific to overriding JRE classes in this way is that they will never actually be used in development mode. In development mode, the native JRE classes always supercede classes compiled from source.

The `<super-source>` element supports [pattern-based filtering](#) to allow fine-grained control over which resources get copied into the output directory during a GWT compile.

---

## XML ELEMENT REFERENCE

This section documents the most commonly used elements in the module XML file.

- `<inherits name="logical-module-name" />` : Inherits all the settings from the specified module as if the contents of the inherited module's XML were copied verbatim. Any number of modules can be inherited in this manner. See also [this advice about deciding which modules to inherit](#).
- `<entry-point class="classname" />` : Specifies an [entry point](#) class. Any number of entry-point classes can be added, including those from inherited modules. Entry points are all compiled into a single codebase. They are called sequentially in the order in which they appear in the module file. So when the `onModuleLoad()` of your first entry point finishes, the next entry point is called immediately.
- `<source path="path" />` : Each occurrence of the `<source>` tag adds a package to the [source path](#) by combining the package in which the module XML is found with the specified path to a subpackage. Any Java source file appearing in this subpackage or any of its subpackages is assumed to be translatable. The `<source>` element supports [pattern-based filtering](#) to allow fine-grained control over which resources get copied into the output directory during a GWT compile.

*If no `<source>` element is defined in a module XML file, the client subpackage is implicitly added to the source path as if `<source path="client" />` had been found in the XML. This default helps keep module XML compact for standard project layouts.*

**//TODO: This should use the grey box note instead**

- `<public path="path" />` : Each occurrence of the `<public>` tag adds a package to the [public path](#) by combining the package in which the module XML is found with the specified path to identify the root of a public path entry. Any file appearing in this package or any of its subpackages will be treated as a publicly-accessible resource. The `<public>` element supports [pattern-based filtering](#) to allow fine-grained control over which resources get copied into the output directory during a GWT compile.

*If no `<public>` element is defined in a module XML file, the public subpackage is implicitly added to the public path as if `<public path="public">` had been found in the XML. This default helps keep module XML compact for standard project layouts.*

**//TODO: This should use the grey box note instead**

- `<servlet path="url-path" class="classname" />` : For RPC, this element loads a servlet class mounted at the specified URL path. The URL path should be absolute and have the form of a directory (for example, `/calendar`). Your client code then specifies this URL mapping by annotating the service interface with the [@RemoteServiceRelativePath](#) attribute. Any number of servlets may be loaded in this manner, including those from inherited modules.

*The `<servlet>` element applies only to GWT's embedded server server-side debugging feature.*

*NOTE: as of GWT 1.6, this tag does no longer loads servlets in development mode, instead you must configure a `WEB-INF/web.xml` in your war directory to load any servlets needed.*

**//TODO: This should use the grey box note instead**

- `<script src="js-url" />` : Automatically injects the external JavaScript file located at the location specified by `src`. See [automatic resource inclusion](#) for details. If the specified URL is not absolute, the resource will be loaded from the module's base URL (in other words, it would most likely be a public resource).
- `<stylesheet src="css-url" />` : Automatically injects the external CSS file located at the location specified by `src`. See [automatic resource inclusion](#) for details. If the specified URL is not absolute, the resource will be loaded from the module's base URL (in other words, it would most likely be a public resource).
- `<extend-property name="client-property-name" values="comma-separated-values" />` : Extends the set of values for an existing client property. Any number of values may be added in this manner, and client property values accumulate through inherited modules. You will likely only find this useful for [specifying locales in internationalization](#).

---

## ELEMENTS FOR DEFERRED BINDING

The following elements are used for defining [deferred binding](#) rules. Deferred binding is not commonly used in user projects.

- `<replace-with class="replacement_class_name">` : A directive to use deferred binding with replacement.
- `<generate-with class="generator_class_name">` : A directive to use deferred binding using a [Generator](#)
- `<define-property name="property_name" values="property_values">` : Define a property and allowable values (comma-separated identifiers). This element is typically used to generate a value that will be evaluated by a rule using a `<when . . .>` element.
- `<set-property name="property_name" value="property_value">` : Set the value of a previously-defined property (see `<define property>` above). This element is typically used to generate a value that will be evaluated by a rule using a `<when . . .>` element. Note that `set-property` and `property-provider` on the same value will overwrite each other. The last definition encountered is the one that is used.
- `<property-provider name="property_name">` : Define a JavaScript fragment that will return the value for the named property at runtime. This element is typically used to generate a value that will be evaluated in a `<when . . .>` element. To see examples of `<property-provider>` definitions in action, see the files `I18N.gwt.xml` and `UserAgent.gwt.xml` in the GWT source code. Note that `set-property` and `property-provider` on the same value will overwrite each other. The last definition encountered is the one that is used.

## DEFINING CONDITIONS

The `<replace-with-class>` and `<generate-with-class>` elements can take a `<when...>` child element that defines when this rule should be used, much like the `WHERE` predicate of an SQL query. The three different types of predicates are:

- `<when-property-is name="property_name" value="value" />` : Deferred binding predicate that is true when a named property has a given value.
- `<when-type-assignable class="class_name" />` : Deferred binding predicate that is true for types in the type system that are assignable to the specified type.
- `<when-type-is class="class_name" />` : Deferred binding predicate that is true for exactly one type in the type system.

Several different predicates can be combined into an expression. Surround your `<when...>` elements using the following nesting elements begin/end tags:

- `<all> when_expressions </all>` : Predicate that ANDs all child conditions.
- `<any> when_expressions </any>` : Predicate that ORs all child conditions.
- `<none> when_expressions </none>` : Predicate that NANDs all child conditions.

## DEFERRED BINDING EXAMPLE

As an example module XML file that makes use of deferred binding rules, here is a module XML file from the GWT source code, `Focus.gwt.xml`:

```
<module>
<inherits name="com.google.gwt.core.Core" />
<inherits name="com.google.gwt.user.UserAgent" />

<!-- old Mozilla, and Opera need a different implementation -->
<replace-with class="com.google.gwt.user.client.ui.impl.FocusImplOld">
  <when-type-is class="com.google.gwt.user.client.ui.impl.FocusImpl" />
  <any>
    <when-property-is name="user.agent" value="gecko" />
    <when-property-is name="user.agent" value="opera" />
  </any>
</replace-with>

<!-- Safari needs a different hidden input -->
<replace-with class="com.google.gwt.user.client.ui.impl.FocusImplSafari">
  <when-type-is class="com.google.gwt.user.client.ui.impl.FocusImpl" />
  <when-property-is name="user.agent" value="safari" />
</replace-with>

<!-- IE's implementation traps exceptions on invalid setFocus() -->
<replace-with class="com.google.gwt.user.client.ui.impl.FocusImplIE6">
<when-type-is class="com.google.gwt.user.client.ui.impl.FocusImpl" />
  <any>
    <when-property-is name="user.agent" value="ie6" />
  </any>
</replace-with>
```

```
</replace-with>
</module>
```

## HOW DO I KNOW WHICH GWT MODULES I NEED TO INHERIT?

GWT libraries are organized into modules. The standard modules contain big pieces of functionality designed to work independently of each other. By selecting only the modules you need for your project (for example the JSON module rather than the XML module), you minimize complexity and reduce compilation time.

Generally, you want to inherit at least the User module. The User module contains all the core GWT functionality, including the EntryPoint class. The User module also contains reusable UI components (widgets and panels) and support for the History feature, Internationalization, DOM programming, and more.

## STANDARD MODULES GWT 1.5

//TODO: this needs to be updated with the most current information for 2.5

Module	Logical Name	Module Definition	Contents
User	com.google.gwt.user.User	User.gwt.xml	Core GWT functionality
HTTP	com.google.gwt.http.HTTP	HTTP.gwt.xml	Low-level HTTP communications library
JSON	com.google.gwt.json.JSON	JSON.gwt.xml	JSON creation and parsing
JUnit	com.google.gwt.junit.JUnit	JUnit.gwt.xml	JUnit testing framework integration
XML	com.google.gwt.xml.XML	XML.gwt.xml	XML document creation and parsing

GWT 1.5 also provides several *theme* modules which contain default styles for widgets and panels. You can specify one theme in your project's module XML file to use as a starting point for styling your application, but you are not required to use any of them.

## THEMES

//TODO: add some information about GWT themes here

Module	Logical Name	Module Definition	Contents
Chrome	com.google.gwt.user.theme.chrome.Chrome	Chrome.gwt.xml	Style sheet
Dark	com.google.gwt.user.theme.dark.Dark	Dark.gwt.xml	Style sheet
Standard	com.google.gwt.user.theme.standard.Standard	Standard.gwt.xml	Style sheet

## How To

To inherit a module, edit your project's module XML file and specify the logical name of the module you want to inherit in the `<inherits>` tag.

```
<inherits name="com.google.gwt.junit.JUnit"/>
```

**Note:** Modules are always referred to by their logical names. The logical name of a module is of the form *pkg1.pkg2.ModuleName* (although any number of packages may be present). The logical name includes neither the actual file system path nor the file extension.

## AUTOMATIC RESOURCE INCLUSION

Modules can contain references to external JavaScript and CSS files, causing them to be automatically loaded when the module itself is loaded. This can be handy if your module is intended to be used as a reusable component because your module will not have to rely on the HTML host page to specify an external JavaScript file or stylesheet.

## INCLUDING EXTERNAL JAVASCRIPT

Script inclusion is a convenient way to automatically associate external JavaScript files with your module. Use the following syntax to cause an external JavaScript file to be loaded into the [host page](#) before your module entry point is called.

```
<script src="_js-url_" />
```

The script is loaded into the namespace of the [host page](#) as if you had included it explicitly using the HTML `<script>` element. The script will be loaded before your [onModuleLoad\(\)](#) is called.

*Versions of GWT prior to 1.4 required a script-ready function to determine when an included script was loaded. This is no longer required; all included scripts will be loaded when your application starts, in the order in which they are declared.*

**//TODO this needs to go into a grey box**

---

## INCLUDING EXTERNAL STYLESHEETS

Stylesheet inclusion is a convenient way to automatically associate external CSS files with your module. Use the following syntax to cause a CSS file to be automatically attached to the [host page](#).

```
<stylesheet src="_css-url_" />
```

You can add any number of stylesheets this way, and the order of inclusion into the page reflects the order in which the elements appear in your module XML.

---

## RELATIVE VS. ABSOLUTE URL

If an absolute URL is specified in the `src` attribute, that URL will be used verbatim. However, if a non-absolute URL is used (for example, "foo.css"), the module's base URL is prepended to the resource name. This is identical to constructing an absolute URL using `GWT.getModuleBaseURL() + "foo.css"` in client code. This is useful when the target resource is from the module's public path.

---

## INCLUSION AND MODULE INHERITANCE

Module inheritance makes resource inclusion particularly convenient. If you wish to create a reusable library that relies upon particular stylesheets or JavaScript files, you can be sure that clients of your library have everything they need automatically by inheriting from your module.

**//TODO: need to add more information from [Module XML Format](#)**

---

## FILTERING PUBLIC AND SOURCE PACKAGES

The [module XML format's](#) `<public>`, `<source>` and `<super-source>` elements supports certain attributes and nested elements to allow pattern-based inclusion and exclusion in the [public path](#). These elements follow the same rules as [Ant's FileSet](#) element. Please see the [documentation](#) for `FileSet` for a general overview. These elements do not support the full `FileSet` semantics. Only the following attributes and nested elements are currently supported:

- The `includes` attribute
- The `excludes` attribute
- The `defaultexcludes` attribute



- The `casesensitive` attribute
- Nested `include` tags
- Nested `exclude` tags

Other attributes and nested elements are not supported.

#### IMPORTANT

The default value of `defaultexcludes` is `true`. By default, the patterns listed [here](#) are excluded.

#### THE BOOTSTRAP SEQUENCE

Consider the following HTML page that loads a GWT module:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <body onload='alert("w00t!")'>
    <img src='bigImageZero.jpg'></img>
    <script source='externalScriptZero.js'></script>
    <img src='bigImageOne.jpg'></img>
    <img src='reallyBigImageTwo.jpg'></img>
    <script src='myApp/myApp.nocache.js'></script>
    <script src='externalScriptOne.js'></script>
  </body>
</html>
```

The following principles are needed to understand the sequence of operations that will occur in this page:

- `<script>` tags always block evaluation of the page until the script is fetched and evaluated.
- `<img>` tags do **not** block page evaluation.
- Most browsers will allow a maximum of two simultaneous connections for fetching resources.
- The `body.onload()` event will only fire once **all** external resources are fetched, including images and frames.
- The GWT selection script (i.e. `myApp/myApp.nocache.js`) will be fetched and evaluated like a normal script tag, but the compiled script will be fetched **asynchronously**.
- Once the GWT selection script has started, its `onModuleLoad()` can be called at any point after the outer document has been parsed.

Applying these principles to the above example, we obtain the following sequence:

1. The HTML document is fetched and parsing begins.
2. Begin fetching `bigImageZero.jpg`.
3. Begin fetching `externalScriptZero.js`.
4. `bigImageZero.jpg` completes (let's assume). Parsing is blocked until `externalScriptZero.js` is done fetching and evaluating.
5. `externalScriptZero.js` completes.
6. Begin fetching `bigImageOne.jpg` and `reallyBigImageTwo.jpg` simultaneously.

7. `bigImageOne.jpg` completes (let's assume again). `myApp/myApp.nocache.js` begins fetching and evaluating.
8. `myApp/myApp.nocache.js` completes, and the compiled script (`<hashname>.cache.html`) begins fetching in a hidden `IFRAME` (this is non-blocking).
9. `<hashname>.cache.html` completes. `onModuleLoad()` is not called yet, as we're still waiting on `externalScriptOne.js` to complete before the document is considered 'ready'.
10. `externalScriptOne.js` completes. The document is ready, so `onModuleLoad()` fires.
11. `reallyBigImageTwo.jpg` completes.
12. `body.onload()` fires, in this case showing an `alert()` box.

This is a bit complex, but the point is to show exactly when various resources are fetched, and when `onModuleLoad()` will be called. The most important things to remember are that

- You want to put the GWT selection script as early as possible within the body, so that it begins fetching the compiled script before other scripts (because it won't block any other script requests).
- If you are going to be fetching external images and scripts, you want to manage your two connections carefully.
- `<img>` tags are not guaranteed to be done loading when `onModuleLoad()` is called.
- `<script>` tags **are** guaranteed to be done loading when `onModuleLoad()` is called.

---

#### A NOTE ON MULTIPLE GWT MODULES AND ENTRYPOINTS

If you have multiple `EntryPoints` (the interface that defines `onModuleLoad()`) within a module, they will all be called in sequence as soon as that module (and the outer document) is ready.

If you are loading multiple GWT modules within the same page, each module's `EntryPoint` will be called as soon as both that module and the outer document is ready. Two modules' `EntryPoints` are not guaranteed to fire at the same time, or in the same order in which their selection scripts were specified in the host page.

#### USING MAVEN FOR BUILDING MODULES

**//TODO: insert some text about how maven goes about doing this**

---

#### PACKAGING

The only distinction with a standard JAR project is the mix of sources and classes in the output folder. A simple way to achieve this is to add a dedicated `resource` in the POM :

```
<resources>

  <resource>

    <directory>src/main/java</directory>

    <includes>
```

```

        <include>**/*.java</include>

        <include>**/*.gwt.xml</include>

    </includes>

</resource>

</resources>

```

Another option is to let the plugin detect the required source to be included, based on the module descriptor. The benefit is that the plugin will not include java files that are not declared as GWT source by the module descriptor, avoiding end-user to reference your internal classes (usually for library that include both client and server side components).

```

<plugin>

  <groupId>org.codehaus.mojo</groupId>

  <artifactId>gwt-maven-plugin</artifactId>

  <version>2.5.1</version>

  <executions>

    <execution>

      <goals>

        <goal>resources</goal>

      </goals>

    </execution>

  </executions>

</plugin>

```

---

## USING GENERAL PURPOSE JARs AS GWT LIBRARY

Many users want to use common usage libraries as GWT modules. Some would like to reuse the server-side business classes on client side, using a shared Maven module. The requirement to include sources in the JAR can then be annoying : including sur JAR in the webapp means the source code will be distributed with the application.

gwt-maven-plugin provides a convenient \*hack\* to work around this restriction using [Maven convention for source jars](#). The "compileSourcesArtifacts" parameter can be used to select a subset of project dependencies (using "groupId:artifactId" syntax). The plugin will download (or get from your local repository) the artifact sources and add them to the compiler Classpath.

```

<plugin>

  <groupId>org.codehaus.mojo</groupId>

  <artifactId>gwt-maven-plugin</artifactId>

  <version>2.5.1</version>

  <compileSourcesArtifacts>

    <compileSourcesArtifact>com.myCompany:domain</compileSourcesArtifact>

  </compileSourcesArtifacts>

</plugin>

```

The only missing fragment to allow using this library from your GWT code is to create a "gwt.xml" module file. Create file "com/mycompany/Domain.gwt.xml" to match package "com.mycompany.domain" to be included as GWT code resource :

```

<module>

  <inherits name="com.google.gwt.user.User" />

  <source path="domain" />

</module>

```

You can include this file in the "domain" JAR (as it will reasonably "pollute" the artifact)

```

|_ domain

|   |_ src/main/java/com/mycompany/domain/SomeUseFullClass.java

|   |_ src/main/resources/com/mycompany/Domain.gwt.xml

|_ webapp

    |_ src/main/resources/com/mycompany/web/MyApp.gwt.xml

```

... or create one for your webapp by creating a phantom package to only contain this file.

```

|_ domain

|   |_ src/main/java/com/mycompany/domain/SomeUseFullClass.java

|_ webapp

|   |_ src/main/resources

        |_ com/mycompany/Domain.gwt.xml

```

## USING A DIFFERENT GWT SDK VERSION

Currently the plugin uses and has been tested with GWT 2.5.1. This version is defined in the plugin dependencies, so if you want another (newer) version you must change your pom as follows:

```
<project>

  <properties>

    <gwt.version>your preferred GWT SDK version here</gwt.version>

  </properties>

  [...]

  <build>

    <plugins>

      [...]

      <plugin>

        <groupId>org.codehaus.mojo</groupId>

        <artifactId>gwt-maven-plugin</artifactId>

        <version>2.5.1</version>

        <dependencies>

          <dependency>

            <groupId>com.google.gwt</groupId>

            <artifactId>gwt-user</artifactId>

            <version>${gwt.version}</version>

          </dependency>

          <dependency>

            <groupId>com.google.gwt</groupId>

            <artifactId>gwt-dev</artifactId>

            <version>${gwt.version}</version>

          </dependency>

        </dependencies>

      </plugin>

    </plugins>

  </build>

</project>
```

```

        </dependency>

        <dependency>

            <groupId>com.google.gwt</groupId>

            <artifactId>gwt-codeserver</artifactId>

            <version>${gwt.version}</version>

        </dependency>

    </dependencies>

</plugin>

[...]
```

```

</plugins>

</build>

[...]
```

```

</project>
```


Note that starting with 2.5.0-rc1, the plugin depends on `gwt-user`, `gwt-dev` and `gwt-codeserver`. Previous versions depended on `gwt-user`, `gwt-dev` and `gwt-servlet`. All dependencies should be overridden.

CREATE MODULES USING ECLIPSE GWT PLUGIN


## USING GWT PLUGIN FOR ECLIPSE

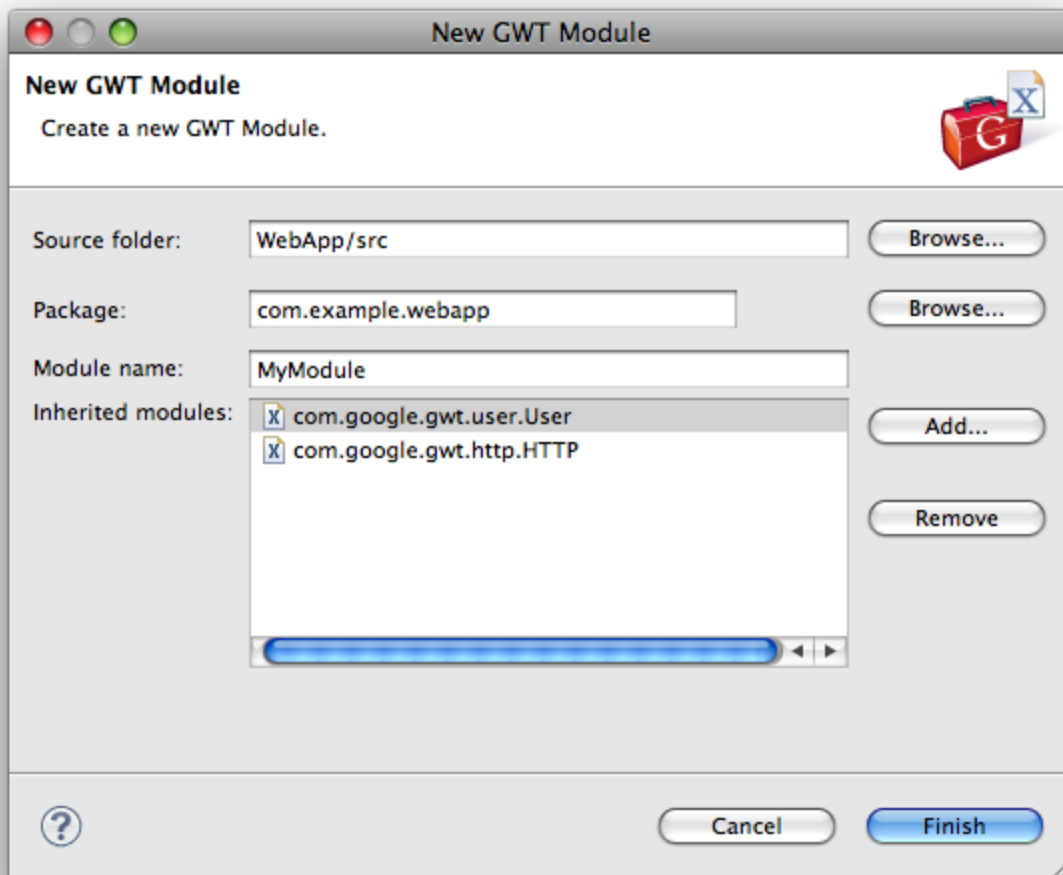
The Google Plugin for Eclipse is designed to make developing GWT applications a breeze. It provides wizards for adding modules, entry points, HTML pages, UiBinders, and ClientBundles. When you're ready to run your application, you can quickly launch development mode or tweak compilation settings without touching command line switches. If your application integrates with JavaScript via JSNI, you'll appreciate a host of features that make working with inline JavaScript easier than ever.

### GWT WIZARDS


The Google Plugin for Eclipse includes a set of wizards to help you quickly create new GWT artifacts. To find them, click on the  **New** button on the toolbar and open the **Google Web Toolkit** category. Alternatively, you can find them in the **File > New** menu.

## CREATING MODULES

The  **New GWT Module** wizard creates a new [GWT module](#) with the specified name. You can customize which modules your new module inherits.



## CREATING ENTRY POINTS

The  **New Entry Point Class** wizard creates a new [entry point](#) class and associated `<entry-point>` module element. It is very similar to the standard **New Java Class** wizard, except that it requires that you also specify the module the entry point is for, and it automatically includes [EntryPoint](#) in the set of interfaces to implement.



**New Entry Point Class**

**Entry Point Class**  
Create a new entry point class.

Source folder: WebApp/src Browse...

Module: com.example.webapp.WebApp Browse...

Package: com.example.webapp.client Browse...

---

Name: MyEntryPoint

Superclass: java.lang.Object Browse...

Interfaces: com.google.gwt.core.client.EntryPoint Add...

Remove

Which method stubs would you like to create?


- ☐ Constructors from superclass
- ☒ Inherited abstract methods

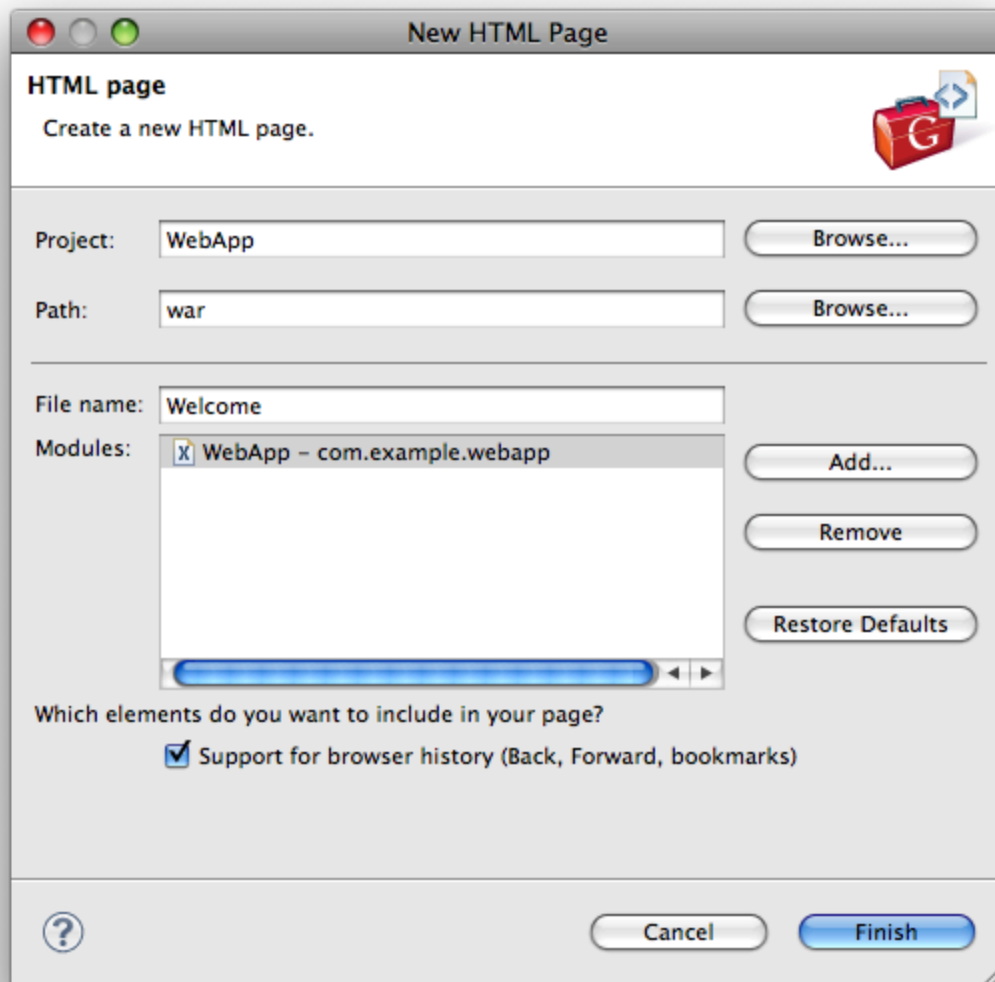
Do you want to add comments? (Configure templates and default value [here](#))

- ☐ Generate comments


? Cancel Finish

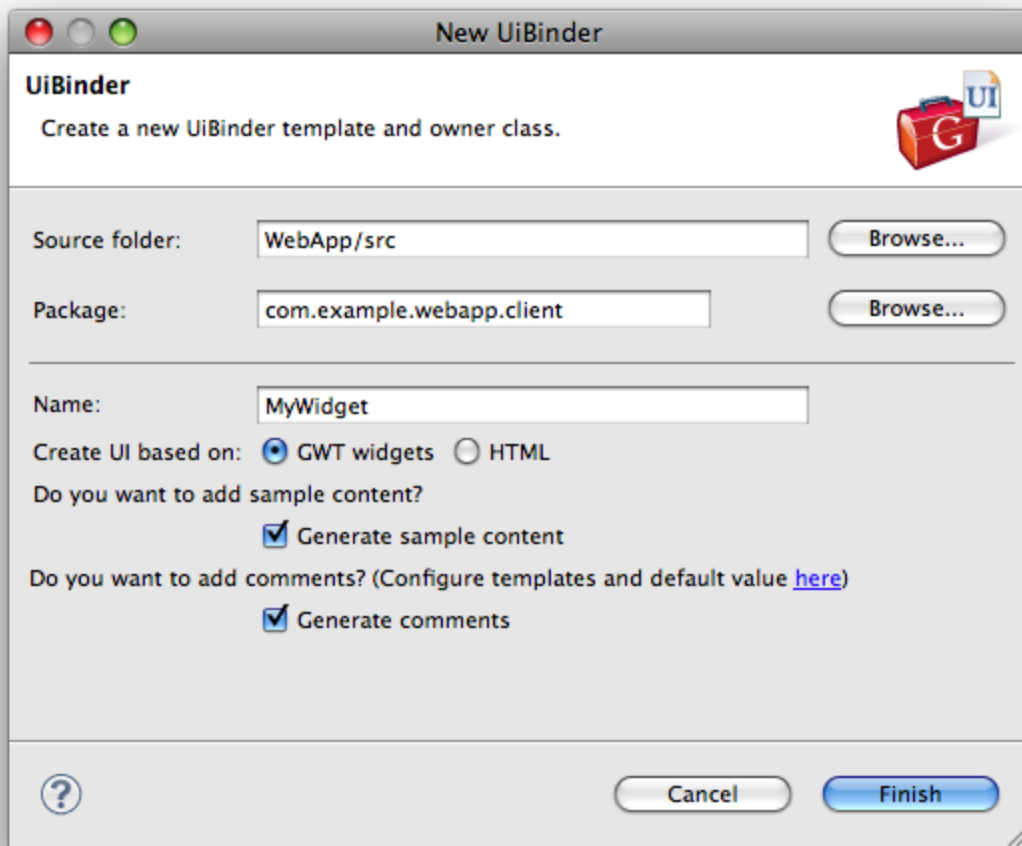
## CREATING HTML PAGES

The  **New HTML Page** wizard generates a new [HTML host page](#) referencing the startup scripts of the selected modules. You can specify whether you would like to include support for [browser history](#).




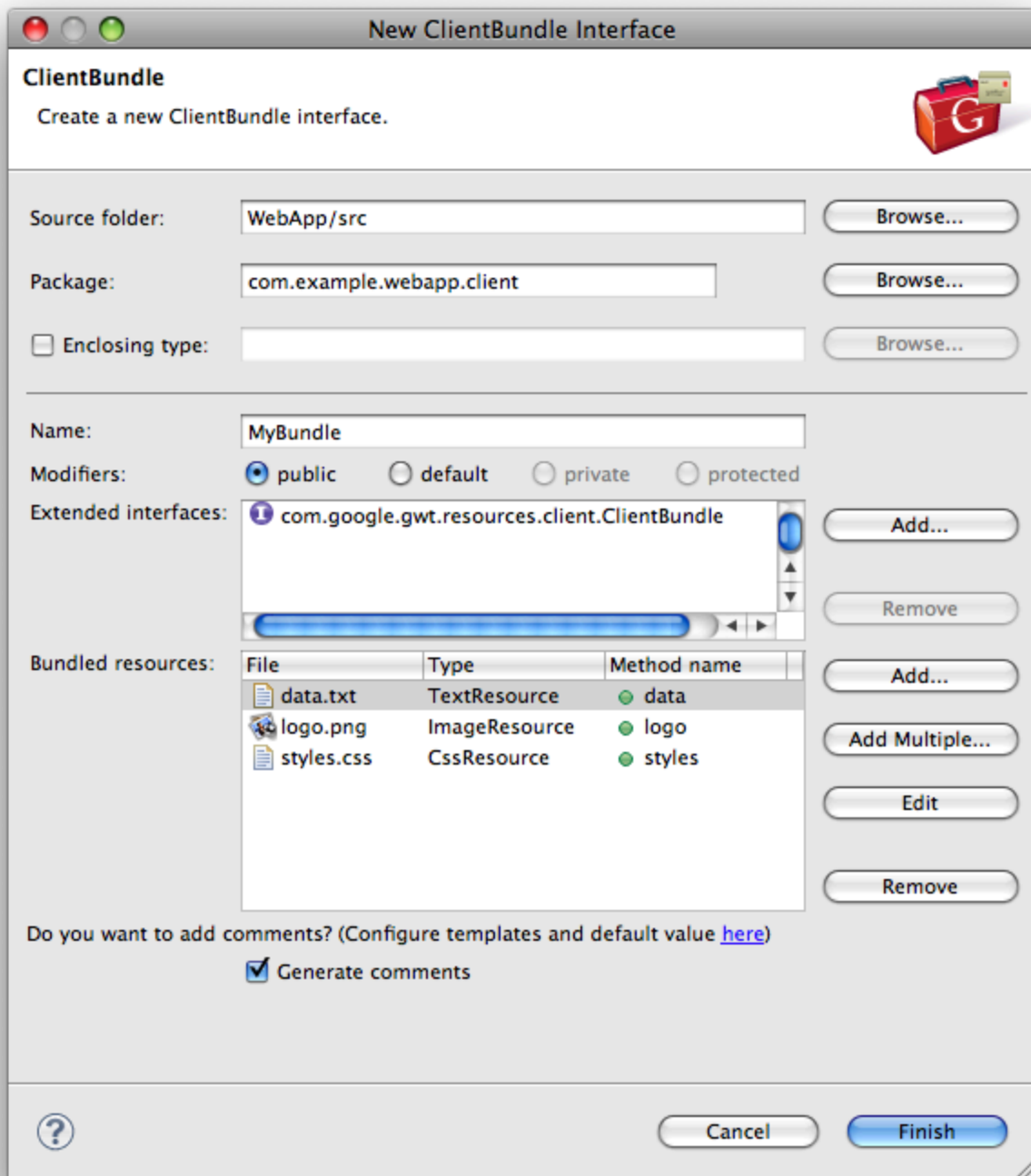
## CREATING UIBINDERS

The  **New UiBinder** wizard creates new [UiBinder](#) templates and owner classes. Once generated, you can use the [custom editor](#) to build up your user interface, while the [automatic validation](#) minimizes errors at compile-time.




## CREATING CLIENTBUNDLE INTERFACES

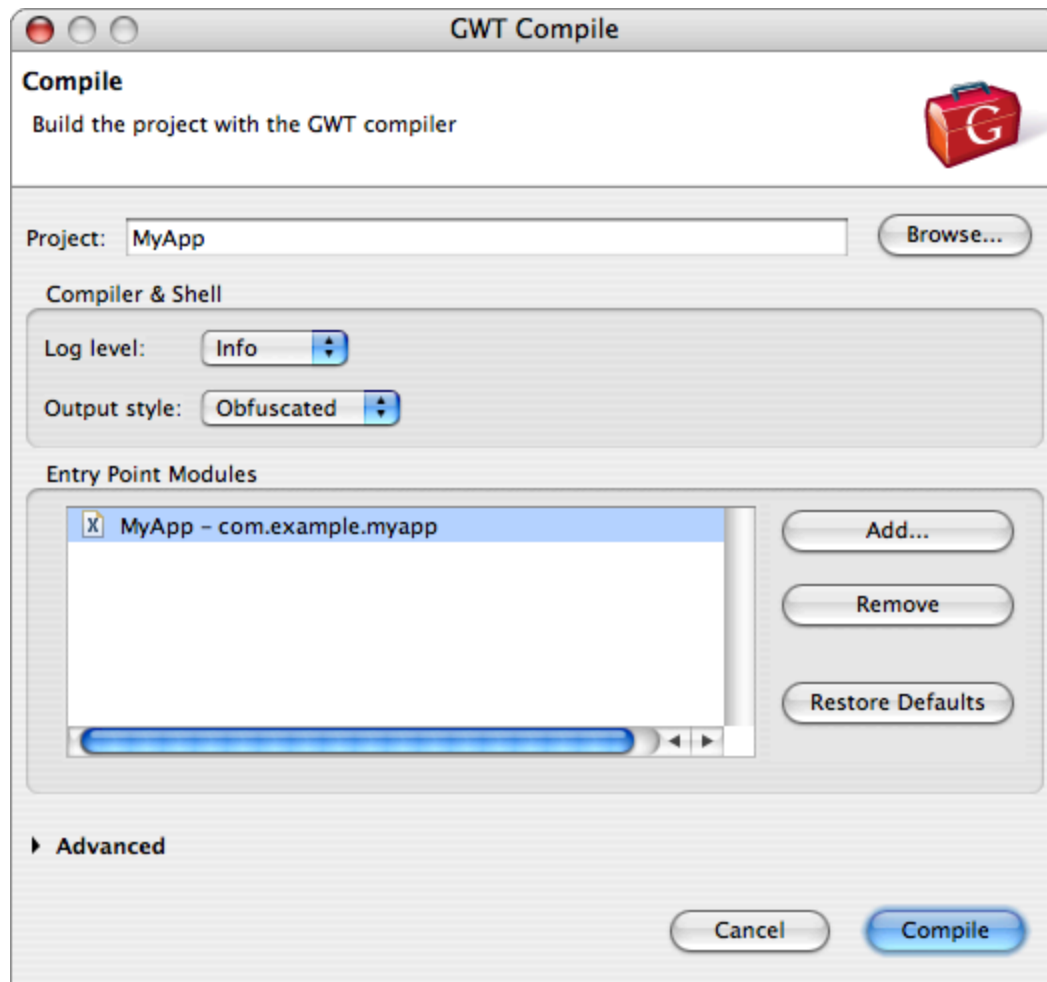
The  **New ClientBundle Interface** wizard creates new [ClientBundle](#) interfaces. Once a new ClientBundle is generated, you can also easily [add more resources](#) and [validate](#) its structure and backing resource files.



## COMPILING WITH GWT COMPILER

To [compile a GWT application to JavaScript](#), click the  **GWT Compile Project** button in the toolbar.

In the **GWT Compile** dialog, you can customize the compilation settings and specify the set of modules to be compiled. The **Advanced** link at the bottom displays text boxes where you can add additional options for the GWT compiler and the Java VM.




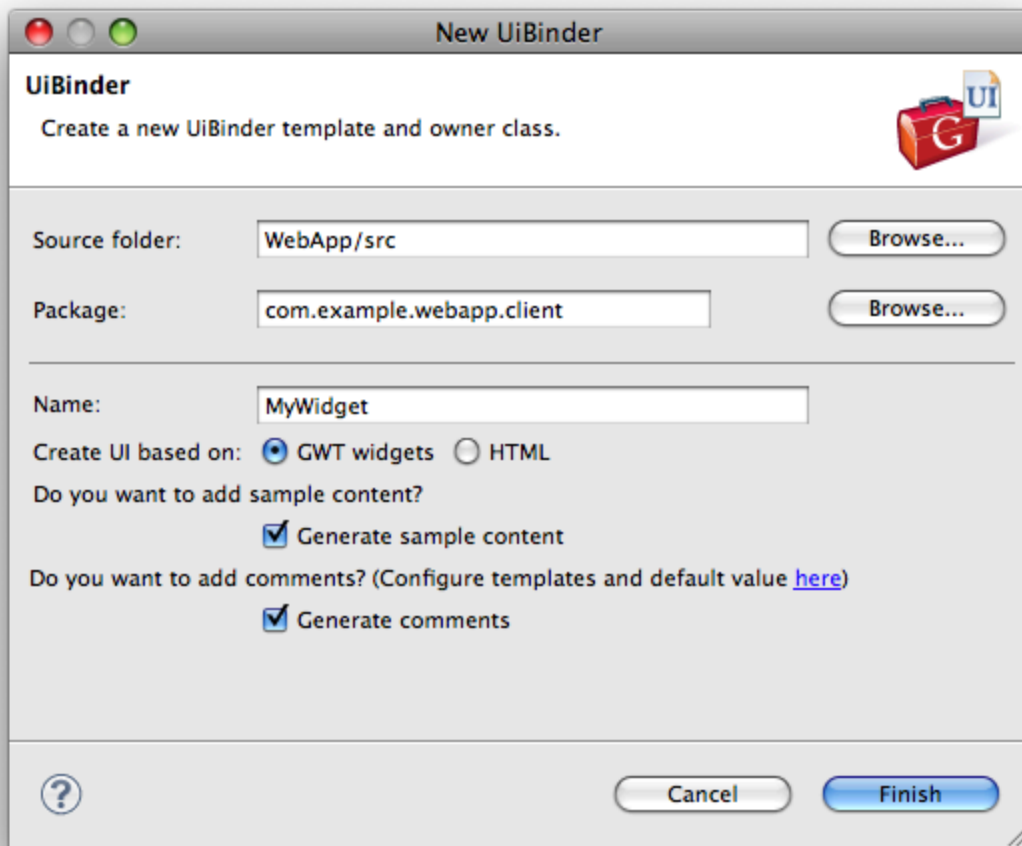
## UiBINDER

The [UiBinder framework](#) allows you to build your apps as HTML pages with GWT widgets sprinkled throughout them. The Google Plugin for Eclipse helps you use UiBinder by providing a UiBinder template editor with auto-completion, validation, and inlined CSS support.

**Note:** For a general overview of the UiBinder framework, see [Declarative Layout with UiBinder](#).

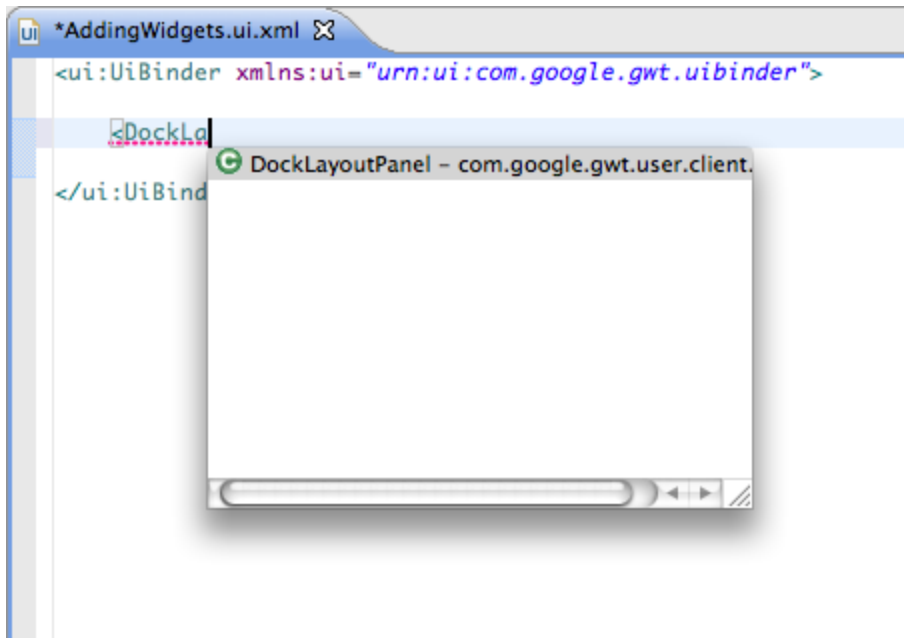
## CREATING A UIBINDER

Select **File > New >  UiBinder** to open the **New UiBinder** wizard. This wizard lets you specify the name and location of your UiBinder, as well as choose whether it will be [composed of Widgets](#) or [HTML-based](#). You can also optionally add sample content and/or comments to the generated files.



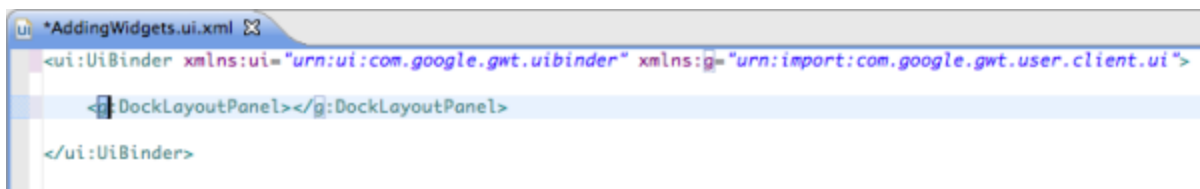
## ADDING GWT WIDGETS

The UiBinder template editor provides auto-completion for widget names and panel-specific child elements.

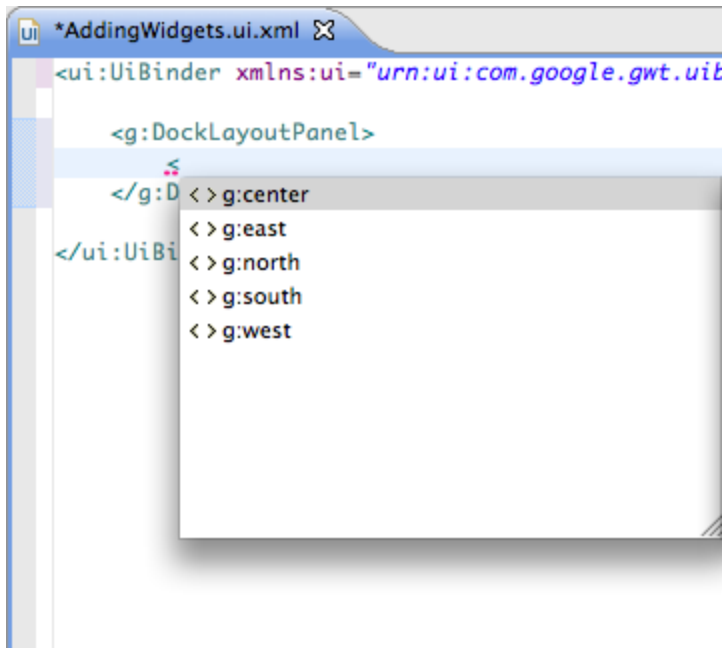


The first time a completion proposal for a widget from an unimported package is chosen, the unimported package will be imported by adding a new namespace to the root `<ui:UiBinder>` element.

**Tip:** You can change the generated prefix for the new namespace after choosing the completion proposal.

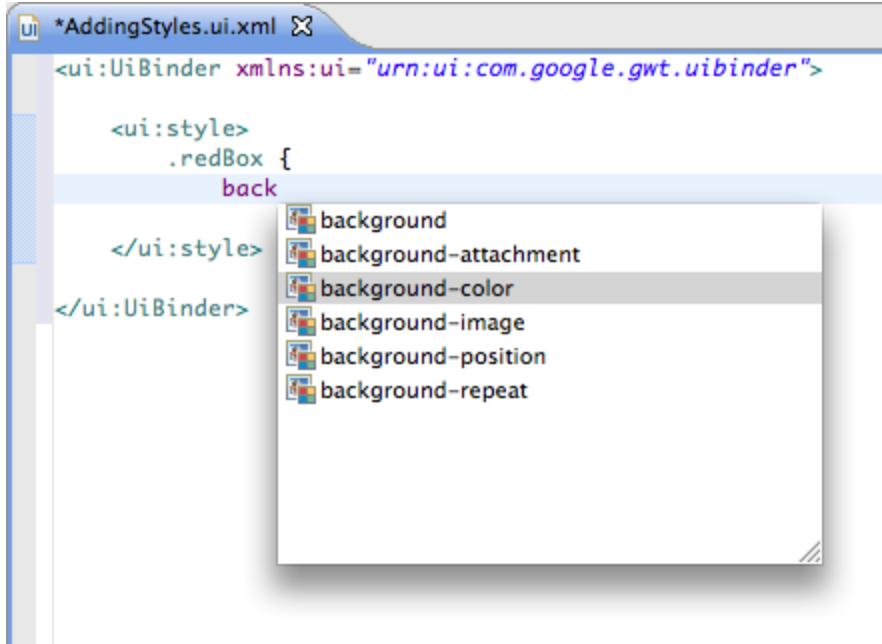


Some layout panels only allow panel-specific child elements, for example `DockLayoutPanel`. In this case, use auto-completion to pick a valid child element.



## ADDING INLINED STYLES

With the `<ui:style>` element, you can define the CSS for your UI right where you need it. The editor provides auto-completion for the inlined CSS.



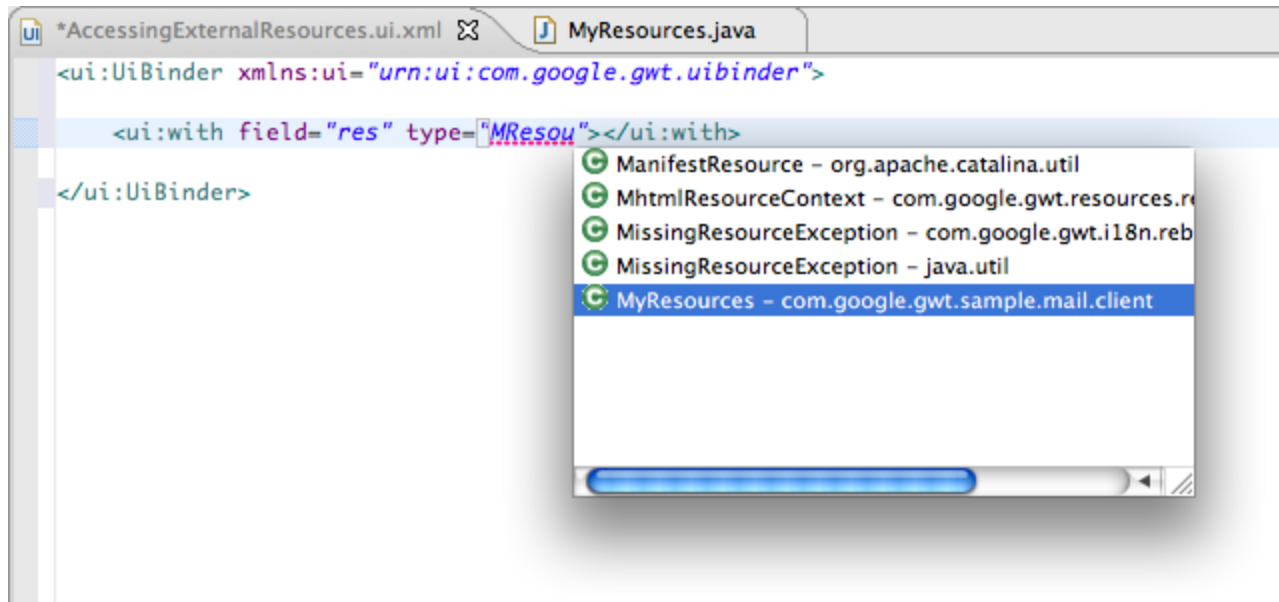
Once your CSS is defined, you can use auto-completion from the field reference to choose a CSS class.



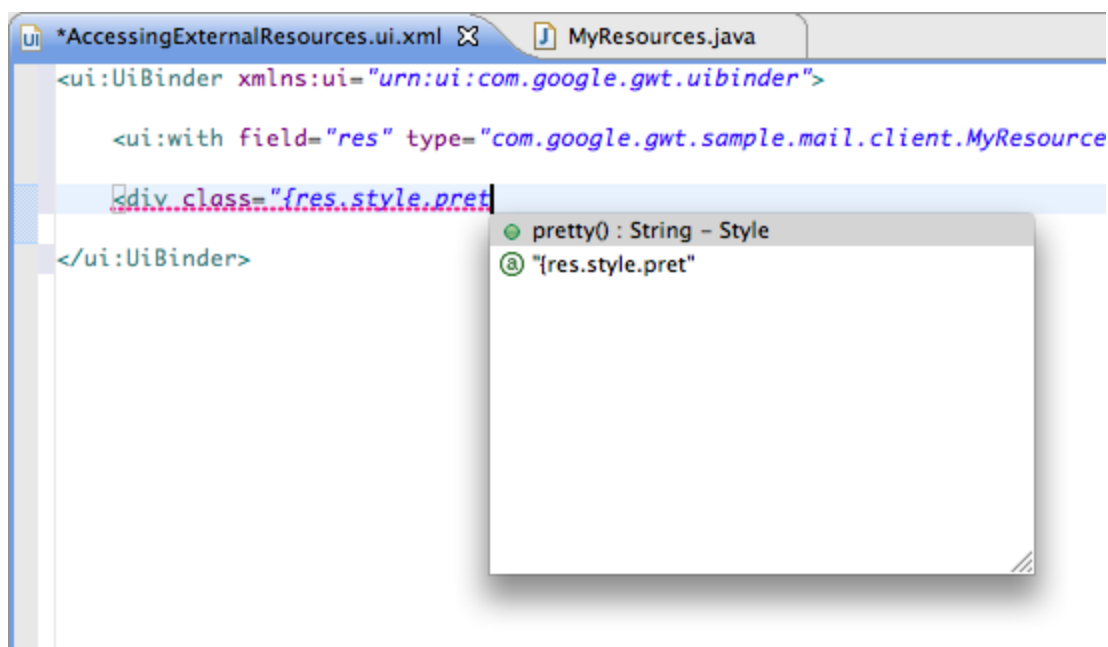


## ACCESSING EXTERNAL RESOURCES

With the `<ui:with>` element, you can refer to Java types instantiable with `GWT.create()`. The editor provides auto-completion for picking types.



Once the type is defined, you can use auto-completion from the field reference to choose a method.



## AS-YOU-TYPE VALIDATION

The plugin provides as-you-type validation for many areas of UiBinder templates and owner classes. For example, if a field reference points to an undefined selector, it will be marked as an error.

**Note:** You can change the default severity of any type of problem with **Preferences > Google > Errors/Warnings**.

The screenshot shows an IDE with two tabs: `Contacts.java` and `Contacts.ui.xml`. The `Contacts.ui.xml` tab is active, displaying the following XML code:

```
<ui:UiBinder
  xmlns:ui='urn:ui:com.google.gwt.uibinder'
  xmlns:g='urn:import:com.google.gwt.user.client.ui'
  xmlns:mail='urn:import:com.google.gwt.sample.mail.client'>

  <ui:style field='style'
    type='com.google.gwt.sample.mail.client.Contacts.Style'>

    .contacts {
      padding: 0.5em;
      line-height: 150%;
    }

    .item {
      display: block;
    }
  </ui:style>

  <g:FlowPanel styleName='{style.contact}' ui:field='panel' />
</ui:UiBinder>
```

A red error icon is visible next to the `<g:FlowPanel styleName='{style.contact}' ui:field='panel' />` line. Below the code editor, the **Problems** tab is selected, showing the following error:

Description	Resource
1 error, 0 warnings, 0 others	
▼ Errors (1 item)	
✖ CSS selector contact is undefined	Contacts.ui.xml

Owner classes also get validation. For example, if a field marked with `@UiField` does not exist in its corresponding UiBinder template, it will be marked as an error.

The screenshot shows an IDE with two tabs: `Contacts.java` and `Contacts.ui.xml`. The `Contacts.java` tab is active, displaying the following Java code:

```
new Contact("John von Neumann", "john@example.com");

@UiField ComplexPanel panel;
@UiField Style styel;
```

A red error icon is visible next to the `@UiField Style styel;` line. Below the code editor, the **Problems** tab is selected, showing the following error:

Description	Resource
1 error, 0 warnings, 0 others	
▼ Errors (1 item)	
✖ Field styel has no corresponding field in template file Contacts.ui.xml	Contacts.java

## CLIENTBUNDLE INTERFACES

[ClientBundle](#) interfaces let you bundle together resources used by your application, such as images, CSS stylesheets, and text, into a single unit that can be statically verified by the compiler and inlined into your app's initial download. The Google Plugin for Eclipse makes it easy to create custom ClientBundle interfaces and keep them in sync with your resource files.

## CREATING A CLIENTBUNDLE INTERFACE

Select the resources you want to bundle in the **Package Explorer** view and then select **File > New >  ClientBundle** to open the **New ClientBundle Interface** wizard.

**New ClientBundle Interface**

**ClientBundle**  
Create a new ClientBundle interface.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected

Extended interfaces:  com.google.gwt.resources.client.ClientBundle

Bundled resources:

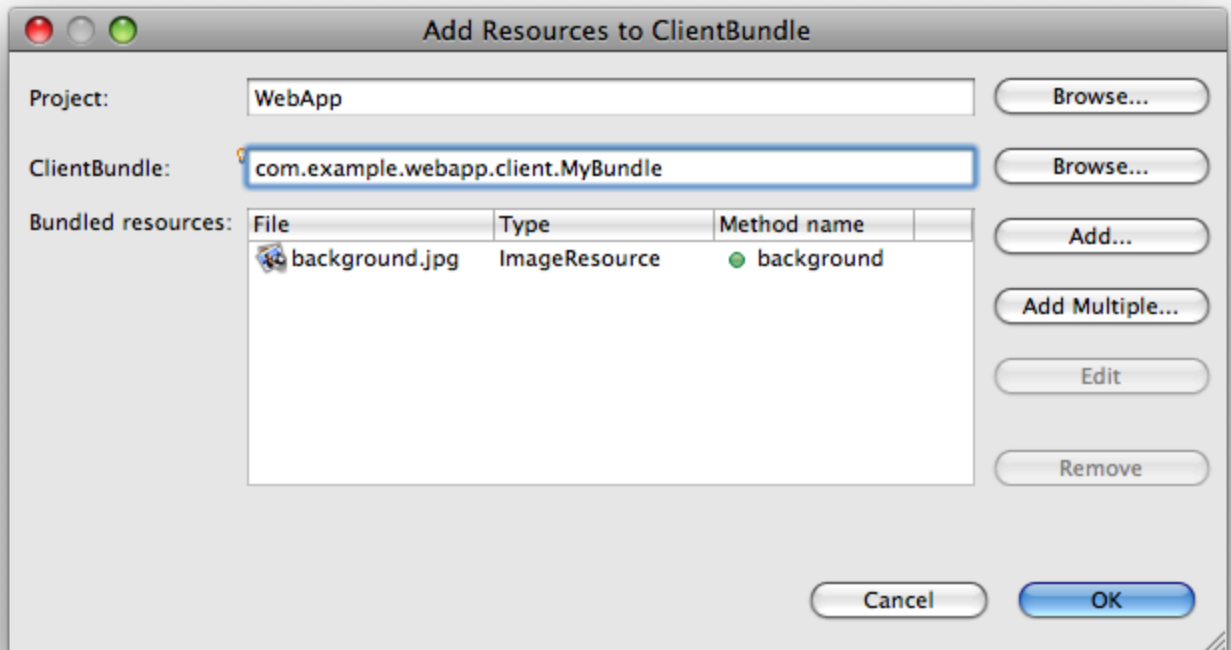
File	Type	Method name
data.txt	TextResource	data
logo.png	ImageResource	logo
styles.css	CssResource	styles

Do you want to add comments? (Configure templates and default value [here](#))  
☒ Generate comments

The wizard is very similar to the built-in **New Java Interface** wizard, except for the addition of the **Bundled resources** list. Each resource is identified by its filename, resource type (e.g. ImageResource), and the name of its corresponding method in the generated ClientBundle interface.

## ADDING RESOURCES TO AN EXISTING CLIENTBUNDLE

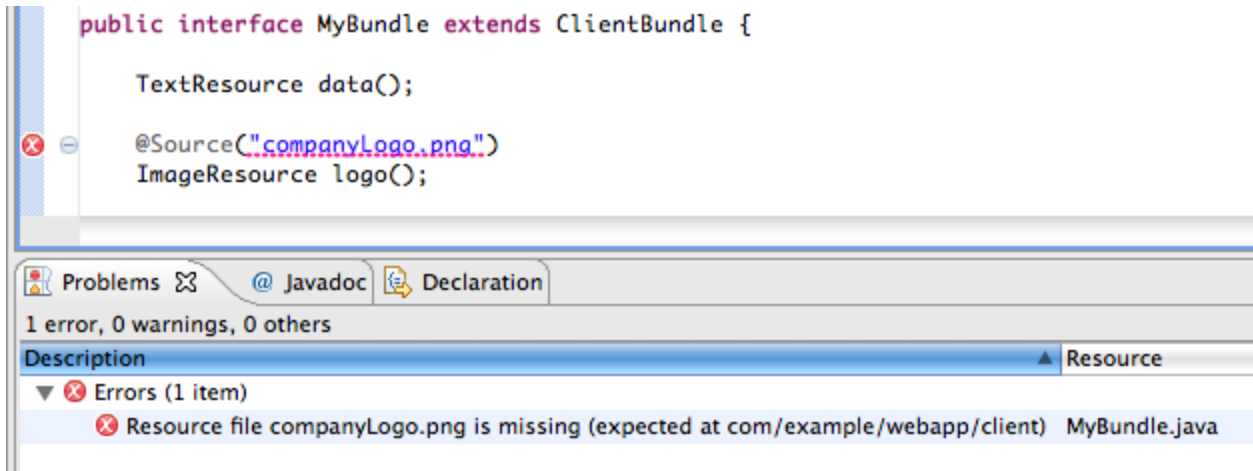
To add resources to an existing ClientBundle interface, select the resource(s) and/or ClientBundle interface in the **Package Explorer** view and select **Google > Add to ClientBundle...** from the context menu.



## CLIENTBUNDLE VALIDATION

Usually, each method defined by a ClientBundle interface is backed by a resource file. For example, a method defined as `ImageResource logo();` might be backed by a file named `logo.png` located in the same package. By default, the GWT compiler will look for a resource with the same name as the method. You can also specify a resource file explicitly using the `@Source` annotation. In either case, the plugin will flag an error on any method that is missing a corresponding resource file. It will also generate errors if the method does not conform to other ClientBundle conventions.

**Note:** You can change the default severity of any type of problem with **Preferences > Google > Errors/Warnings**.



## WORKING WITH JSNI

The Google Plugin for Eclipse adds special treatment for one particular feature of GWT: [JavaScript Native Interface \(JSNI\)](#).

## JSNI METHODS

With GWT, you can embed raw JavaScript directly into your web applications with JSNI methods. JSNI methods are declared as native Java methods and contain inline JavaScript code within a specially-formatted multi-line comment to hide it from the Java compiler. However, this means that by default, Eclipse treats JSNI method bodies as regular Java comments.

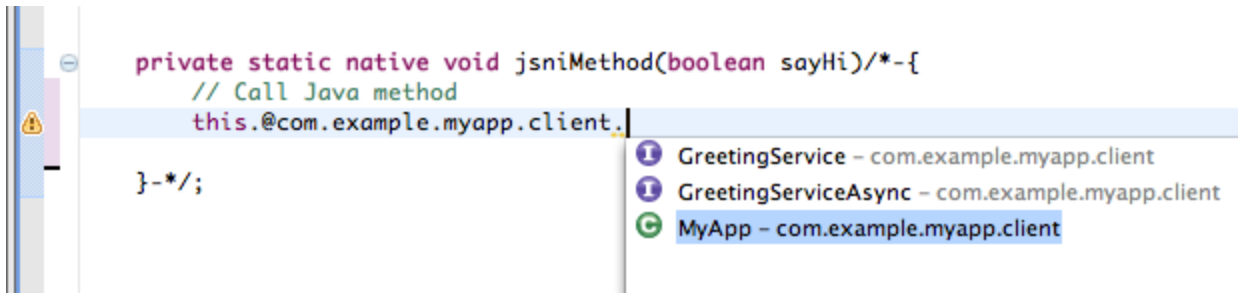
The Google plugin, however, understands JSNI methods and makes working with embedded JavaScript nearly as easy as editing your regular Java code. It provides syntax coloring and auto-indent functionality.

```
private static native void jsniMethod(boolean sayHi)/*-{  
    // Display a pop-up  
    if (sayHi) {  
        var name = this.@com.example.myapp.client.MyApp::name;  
        window.alert('Hello, ' + name);  
    }  
}-*/;
```

## JAVA REFERENCES FROM JSNI

JSNI methods often refer to fields or classes defined in your application's Java source. The plugin provides several features to make referencing Java from JSNI easier.

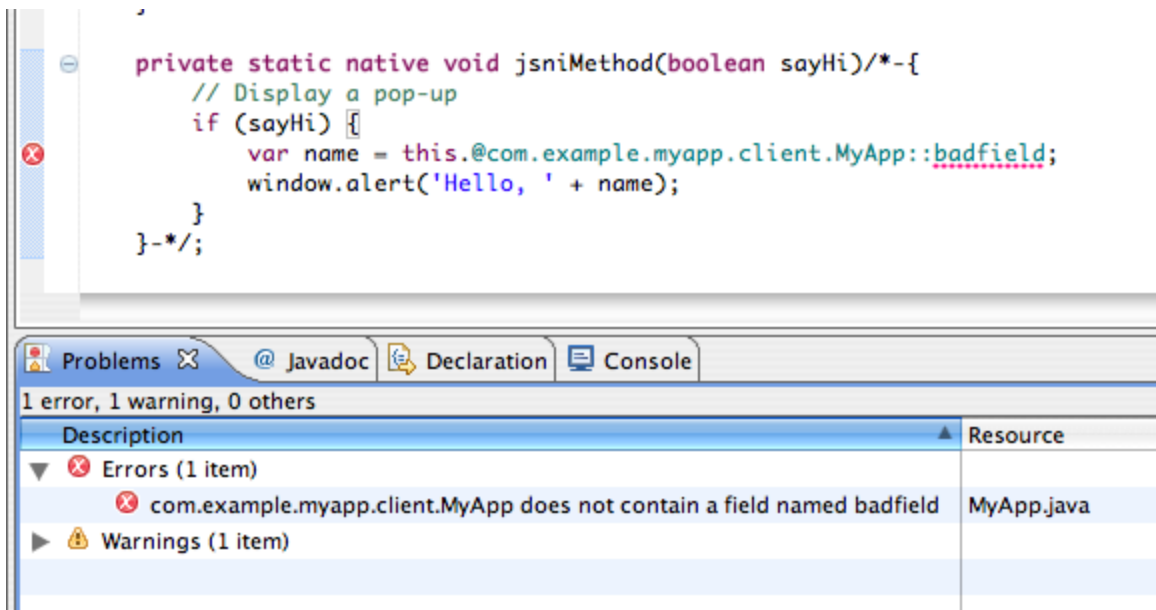
## AUTO-COMPLETION



## VALIDATION

The plugin validates Java references and alerts you when one is broken.

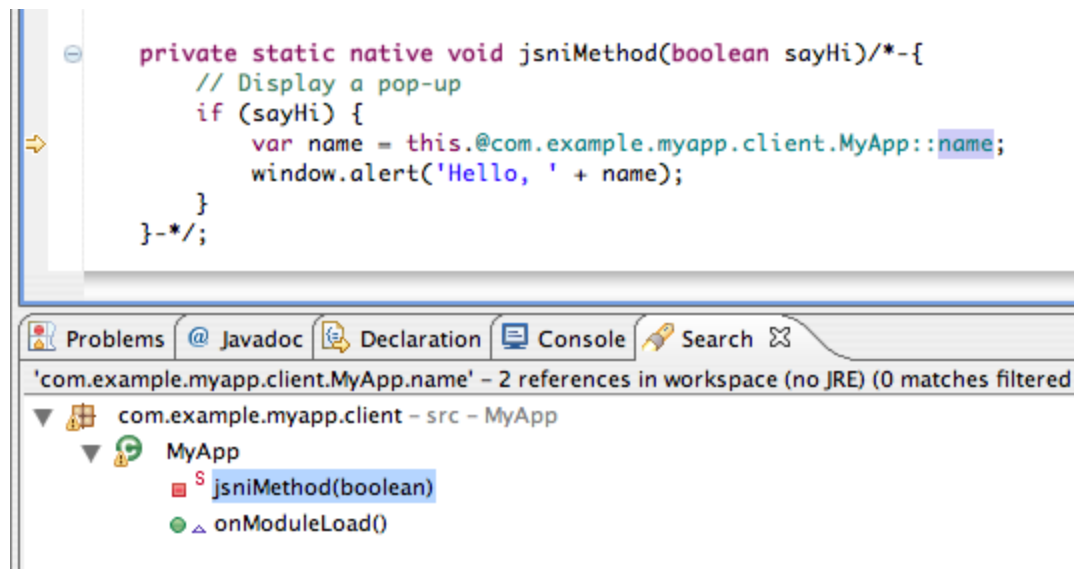
**Note:** You can change the default severity of any type of problem with **Preferences > Google > Errors/Warnings**.



## SEARCH INTEGRATION

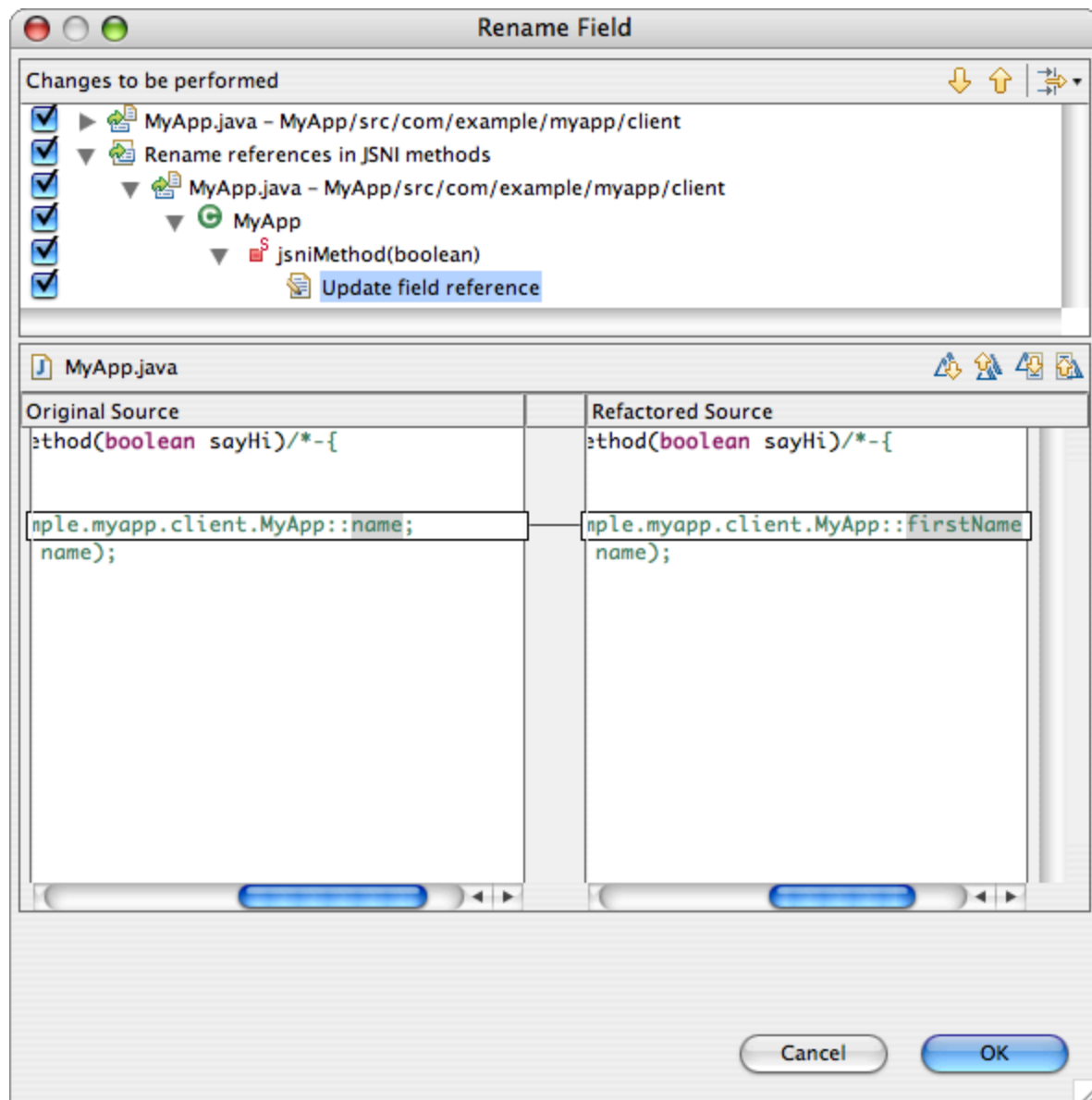
Java references inside JSNI are also included when you perform a Java Search.





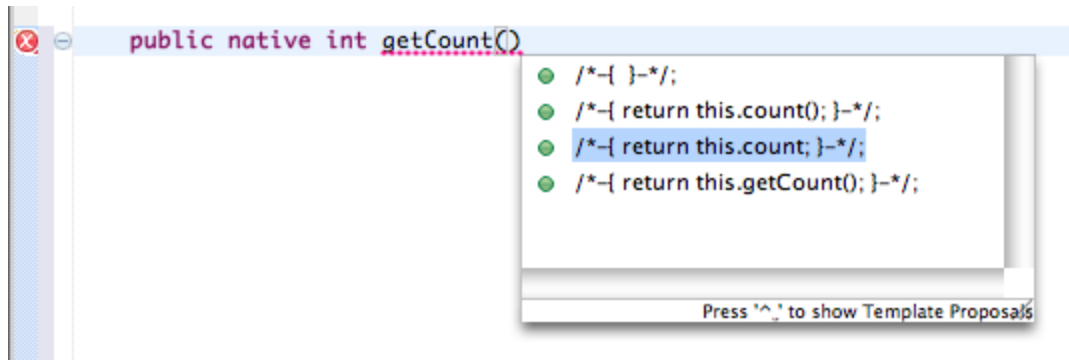
## REFACTORING INTEGRATION

When you refactor a referenced Java class or member, the JSNI references will be updated as well.



## JSNI METHOD BODY AUTO-COMPLETION

GWT 1.5 introduced [JavaScript Overlay Types](#), which are Java classes that act as thin wrappers around actual JavaScript objects. The plugin makes it a snap to implement overlay types, by providing code-completions for generating the most common JSNI method bodies. Just hit Ctrl-Space at the end of a JSNI instance method signature.

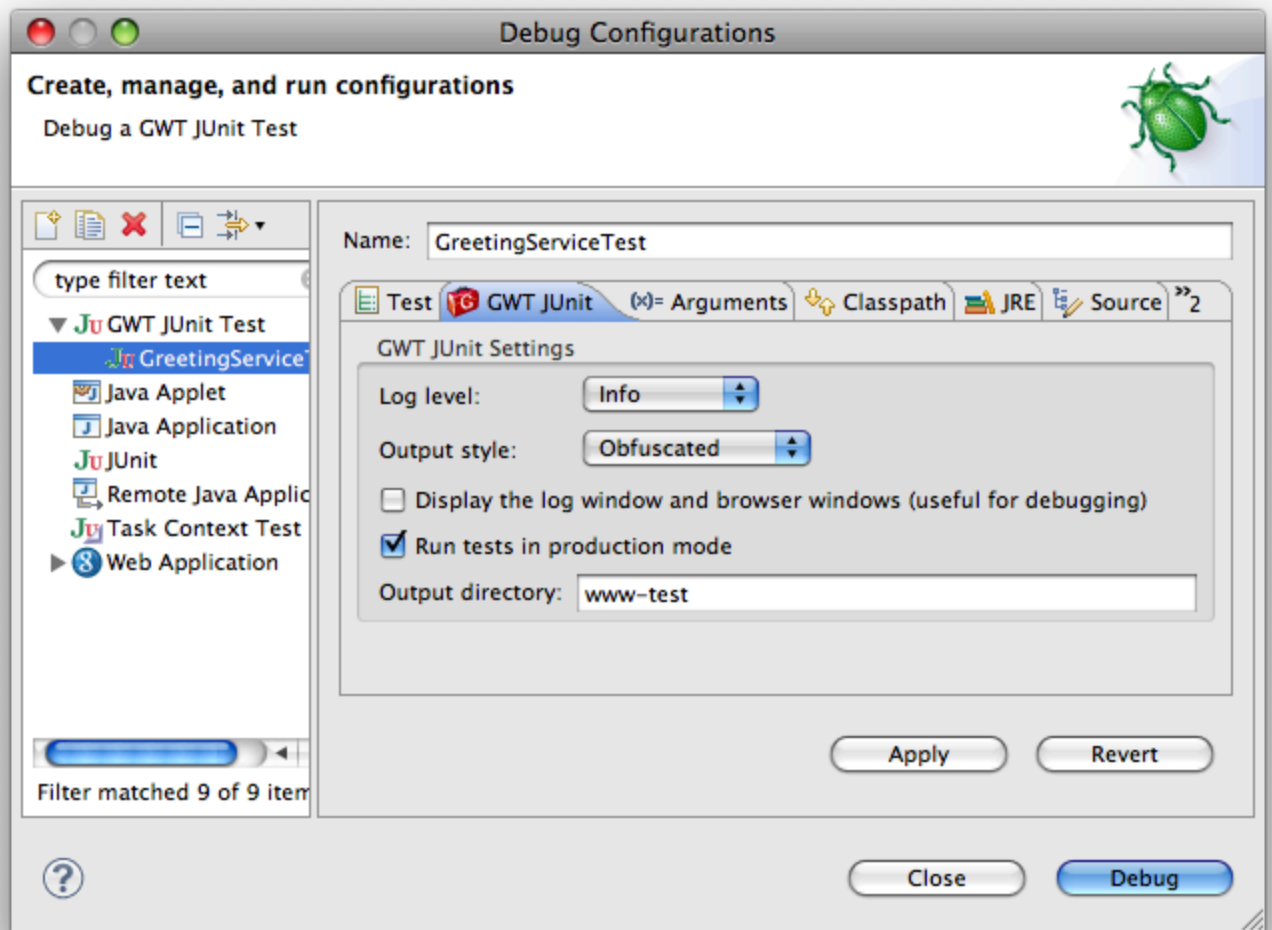


## GWT JUNIT TESTING

When [unit testing your GWT application](#), you can specify additional GWT-specific settings such as log level and compiler output style. You can also run tests in either development mode or production mode. To simplify running GWT tests, the plugin provides a custom launch type: **GWT JUnit Test**.

To launch a [GWT unit test](#), right-click the test class and select **Run As > GWT JUnit Test** or **Run As > GWT JUnit Test (production mode)**. You should see the test results appear in Eclipse's JUnit view.

To customize the settings for a unit test launch, open the **Run Configurations** dialog and select the **Ju** **GWT JUnit Test** launch you want to edit.



## PROJECT VALIDATIONS

Google Plugin for Eclipse performs verifications to ensure that your project is properly configured. If any problems are detected, the problems are displayed in the **Problems** view. You can activate quick-fixes for these problems by selecting the problem in the **Problems** view, and hitting Ctrl-1 (or Command-1 on the Mac).

For each Google SDK your project uses, the validator will verify that:

- the SDK exists on disk
- the SDK libraries are on the build path

For projects that use a WAR directory, the validator verifies that:

- the WAR directory exists on disk
- the project has a `<WAR>/WEB-INF/web.xml` file

If the WAR directory is used for launches and deploys (configurable in project properties: **Google > Web Application**), the validator also checks that:

- the output directory is set to `<WAR>/WEB-INF/classes`
- the libraries on your build path (including SDK libraries) exist in `<WAR>/WEB-INF/lib`

For App Engine projects in particular, the validator verifies that the project has a `<WAR>/WEB-INF/appengine-web.xml` file.

**Note:** You can change the default severity of any type of problem with **Preferences > Google > Errors/Warnings**.

## USING GOOGLE PLUGIN FOR ECLIPSE WITH EXISTING PROJECTS

The Google Plugin for Eclipse makes it easy to create new projects that use GWT and App Engine. It's also easy to set up your existing projects to make use of the plugin.

## IMPORTING YOUR APPLICATION INTO ECLIPSE

Create an Eclipse project for your source, if you haven't already done so, by selecting **File > New > Java Project**. Then choose **Create project from existing source** and set up your project. At this point, your source will be loaded in Eclipse, but the project's build path may not be set up properly, and you may see build errors.

Alternatively, if your application's source tree already contains a `.project` file, either because you had previously worked on it in Eclipse or because it was generated by a tool like GWT's [webAppCreator](#), you can import the project by going to **File > Import > General** and selecting **Existing Projects into Workspace**.

## ENABLING GOOGLE SDKs IF YOUR PROJECT ALREADY HAS A WAR DIRECTORY

To enable Google Web Toolkit, right-click your project and select **Google > Web Toolkit Settings**. Check the **Use Google Web Toolkit** box and click **OK** to apply the change.

Enabling App Engine for your project is similar: right-click your project and select **Google > App Engine Settings**. Check the **Use Google App Engine** box and click **OK**.

Finally, open project properties and navigate to **Google > Web Application** and check the box for **This project has a WAR directory** and then enter the project-relative path to your WAR directory.

If your project depends on libraries not provided by the GWT and App Engine SDKs, put them in `<WAR>/WEB-INF/lib` and add them to your Java build path. This approach should work for most existing Java web applications, including server-side Java applications (such as those made from the App Engine template), and GWT 1.6 or later projects.

## WORKING WITH GWT 1.5 AND EARLIER PROJECTS

The Google Plugin for Eclipse and GWT 1.6 or later support project structures from previous versions of GWT, which expect HTML pages to live in a directory called `public` underneath a GWT module directory. If you're more comfortable using an existing GWT 1.5 SDK, that will work too. Just add it on the GWT Preferences Page. To do this, select **Window > Preferences** (or **Eclipse > Preferences** on the Mac) and navigate to **Google > Web Toolkit**.

However, we recommend upgrading to the new project layout and using the current GWT SDK, which is bundled with the plugin. See the [GWT Upgrade Guide](#) for instructions on how to migrate your layout. Once you've added the necessary files to your project and switched over to the new project layout, you'll need to follow the instructions below in order to have Eclipse pick up the changes:.

1. Ensure that the project conforms to the new WAR-style structure, as described in the upgrade guide.
2. If you don't have continuous refresh enabled in your Eclipse configuration, refresh your project manually (either by pressing F5 or right-clicking your project and selecting refresh)
3. In your project's properties dialog, navigate to **Google > Web Application** and check the box for **This project has a WAR directory** and then enter the project-relative path to your WAR directory.
4. Click OK to apply changes and close the properties dialog.