

.....

**Maven GWT Plugin**  
**v. 1.2-SNAPSHOT**  
**Project Documentation**



## Table Of Content

---

1	<b>Back to Mojo</b>	1
2	<b>Version 1.0</b>	
3	<b>Version 1.1</b>	
4	<b>Download as PDF</b>	
5	<b>Index</b>	1
6	<b>Goals</b>	
7	<b>Usage</b>	2
8	<b>GWT 2.0 preview</b>	6
9	<b>FAQ</b>	8
10	<b>Eclipse</b>	9
11	<b>Introduction</b>	13
12	<b>Setup</b>	14
13	<b>the /war folder</b>	17
14	<b>Using the archetype</b>	18
15	<b>Run Hosted Mode</b>	19
16	<b>Generate Async</b>	21
17	<b>Internationalization</b>	23
18	<b>Testing</b>	25
19	<b>Compile and Package</b>	28
20	<b>Multi-project setup</b>	31
21	<b>Writing a GWT library</b>	40
22	<b>Productivity Tip</b>	41
23	<b>compile a GWT module</b>	43
24	<b>generate async interfaces</b>	45
25	<b>generate i18n bundles interfaces</b>	47



# 1 Index

---

## 1.1 Google Web Toolkit Maven Plugin (gwt-maven-plugin)

The gwt-maven-plugin provides support for **GWT** projects, including running the GWT developer tools (the compiler, shell and i18n generation) and performing support operations to help developers make GWT fit more closely in with their standard JEE web application development (debugging, managing the embedded server, running noserver, merging web.xml, generating I18N interfaces, generating Async GWT-RPC interfaces, and more) and environment (Eclipse IDE).

### 1.1.1 Goals, Setup and Overview

To provide you with better understanding of some usages of gwt-maven-plugin, you can take a look at the following goals, setup, and examples information:

- [Goals and Parameters](#)
- [Setup](#)

### 1.1.2 More Details on Specific Goals and Features

- [Testing Support](#)
- [Hosted server Configuration](#)
- [Using the Archetype](#)

### 1.1.3 History

**GWT-maven plugin** has merged with this plugin since version 1.1  
version 1.2 includes a preview of gwt 2.0 support

## 2 Usage

---

### 2.1 Configuring Google Web Toolkit support for Maven

#### 2.1.1 Compile GWT application into JavaScript

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 "compile" phase.

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <configuration>
          <module>com.mycompany.gwt.Module</module>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

You can also configure compilation for multiple modules by nesting them inside a "*modules*" element. If none is set, the plugin will scan project source and resources directories for ".gwt.xml" module files.

See [user guide](#) for more details.

#### 2.1.2 Generate Async interface for GWT-RPC services

The `generateAsync` goal will create a generate-sources folder and Async interface for all RemoteInterface found in the project. To avoid a full scan, only Java source files that matches a pattern (defaults to "\*\*/\*Service.java") are checked.

See [user guide](#) for more details.

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <configuration>
          <servicePattern>*/gwt/**/*Service.java</servicePattern>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
              <goal>generateAsync</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

### 2.1.3 Generate Message / Constants interface for GWT internationalization

The [i18n](#) goal will create a generate-sources folder and Message (or Constants) interface for the ResourceBundles specified in configuration. You can specify the "constantsWithLookup" or "message" parameters to choose the interface hierarchy (Message or Constants) to use, "Message" being used by default.

See [user guide](#) for more details.

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <configuration>
          <i18nMessagesBundle>com.mycompany.MyApp</i18nMessagesBundle>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>i18n</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

#### 2.1.4 Setup hosted mode browser

The [eclipse](#) goal can be used to create an Eclipse launch configuration for executing the GWT module in hosted browser mode. The required native libraries are also downloaded and installed in the user local repository. This goal is not intended to be part of a build phase, but to be executed from command line `mvn gwt:eclipse`.

See [user guide](#) for more details.

#### 2.1.5 Setup GWTTestCase to run in Eclipse

The [eclipseTest](#) goal can be used to create an Eclipse launch configuration for executing the GWTTestCase-based unit tests. The required native libraries are also downloaded and installed in the user local repository. Simply run `mvn gwt:eclipseTest`.

To avoid conflicts with unit tests, we recommend to follow the naming convention to prefix your GWT tests with "GwtTest"

#### 2.1.6 Run GWTTestCase & GWTTestSuite inside Maven

GWTTestCase requires some complex setup that makes it difficult to run in Maven with the Surefire plugin. Such tests also are long and start the whole GWT module, so they are not "unit" but "integration" tests.

The [test](#) goal can be used to run GWTTestCase during the integration-test phase. It will fork a process with the required arguments to run the test, and will report on the console the result. It will also create the standard Surefire reports to be used by project site reporting.

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <goals>
              <goal>test</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

This plugin declaration could be part of a profile, so that it gets only enabled on continuous integration server, as such test execution is long and will penalize the developer.

See [user guide](#) for a detailed description of testing with GWT and Maven, and suggestion about using GwtTestSuite.

## 3 GWT 2.0 preview

---

### 3.1 GWT 2.0

GWT 2.0 is not yet released, but the plugin already support some of it's new features based on nightly-builds.

- draft compilation, usefull in dev or continuous integration to speed up the build
- story of your compiler report
- Dissable Class metadata : remove support for "class.getName()" to reduce script size
- Disable Class Cast checking : remove class cast check in compiled JavaScript code to reduce script size

### 3.2 Setup

As GWT 2.0 is not released, you'll have to install a [JDK preview](#). Set the plugin "gwtHome" parameter to point to this SDK, and configure a System dependency for the gwt-user API.

```
<properties>
  <gwtHome>${basedir}/gwt-0.0.0-6120</gwtHome>
</properties>

<dependencies>
  <dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-user</artifactId>
    <version>2.0-6120</version>
    <scope>system</scope>
    <systemPath>${gwtHome}/gwt-user.jar</systemPath>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>gwt-maven-plugin</artifactId>
      <configuration>
        <gwtHome>${gwtHome}</gwtHome>
        <disableCastChecking>true</disableCastChecking>
        <disableClassMetadata>true</disableClassMetadata>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>generateAsync</goal>
            <goal>compile</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

### 3.3 SOYC report

When GWT 2.0 is detected, the SOYC analysis will be automatically set on (if you don't force it off) so that you can later generate the report.

```
<reporting>
  <excludeDefaults>true</excludeDefaults>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>gwt-maven-plugin</artifactId>
      <reportSets>
        <reportSet>
          <reports>
            <report>soyc</report>
          </reports>
        </reportSet>
      </reportSets>
      <configuration>
        <gwtHome>${gwtHome}</gwtHome>
      </configuration>
    </plugin>
  </plugins>
</reporting>
```

## 4 FAQ

---

### 4.1 Frequently Asked Questions

#### 4.1.1 Do I need GWT SDK on my computer to use the plugin ?

No. The plugin uses maven dependency management to get the required libraries and native components depending on your operating system

#### 4.1.2 How to set the version of GWT to compile my project ?

Simply define a *dependencies* on "com.google.gwt:gwt-user" with the expected version, the plugin will detect this and download the required SDK artifacts.

#### 4.1.3 How does this plugin compare to other maven / gwt plugins ?

This plugin, being part of the Mojo Project, can benefit for a large community of maven developers. Its async interface generation feature is AFAIK not supported by other plugins.

#### 4.1.4 Is there any official Google support for maven ?

GWT guys don't use Maven by themselves but are interested by our work to provide community support. They helped us to make the plugin work with the GWT SDK release candidates.

#### 4.1.5 I get strange NoSuchMethodError running my application unit tests !

`gwt-dev` JAR includes many libraries, including some Apache commons-\* that you may use in your project with more recent version. Don't define `gwt-dev` as a dependency, the plugin will resolve it when necessary for you based on your `gwt-user` version.

## 5 Eclipse

---

### 5.1 Using the Google Eclipse Plugin

The [Google Eclipse Plugin](#) is a nice integration of GWT inside eclipse to make development easier. It can be used to launch the hosted browser with a simple right clic and to manage the GWT sdk used inside Eclipse.

#### 5.1.1 Limitations

A restriction of this plugin is that it will search for gwt modules and host pages only in the first classpath source folder. Using a Maven / Eclipse integration like m2eclipse, this one will be your sourceDirectory ( `src/main/java` ). You'll have to move your `gwt.xml` files in this folder, instead of the standard Maven resource directory. See [Google Eclipse Plugin issue #1597](#)

Another bigger restriction of this plugin, is that it **requires** the hosted mode webapp to use `/war` directory as web application root. We will have to change the maven-war-plugin setup to use this folder and not the default `src/main/webapp` path, but will keep the latest to host the `web.xml` deployment descriptor. We can then have a dedicated hosted mode one where we register stub RPC-servlets for testing purpose. The 'real' web application may be long to start and require more complex resources, like a JDBC DataSource, and we can still run fonctionnal tests on it using the `noserver` option. See [Google Eclipse Plugin issue #1515](#)

#### 5.1.2 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
|         |_WEB-INF
|         |_web.xml
|_war
|  |_Module.html
|  |_WEB-INF
|  |_web.xml

```

The `war` folder will be used to host your test web application (used in hosted mode) where you'll put stub RPC servlets. Your *real* web application can be used for more advanced integration testing with the `noserver` option.

#### 5.1.3 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>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <runTarget>com.mycompany.gwt.Module/Module.html</runTarget>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.0.2</version>
        <configuration>
          <warSourceDirectory>war</warSourceDirectory>
          <webXml>src/main/webapp/WEB-INF/web.xml</webXml>
        </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 hosted browser.

### 5.1.4 Eclipse configuration

Import your maven project into eclipse using m2eclipse import wizard (or your preferred tooling). Manually enable GWT on the project from project preference by setting the `Use Google Web Toolkit` checkbox. Eclipse will add a new `ClassPath` container for GWT. Please check you use the same version of GWT in your maven configuration and in eclipse preferences.

Run the `gwt:eclipse` goal (using `m2eclipse Maven2 > build...`) to setup your environment and create the launch configuration for your GWT modules. The plugin expect the HTML host page to be named as the module and put in the module `public` subdirectory. If you don't use this convention you will have to edit manually the launch configuration.

### 5.1.5 Server side components and RPC

We use the `/war` folder to host our hosted-mode web application. It's `web.xml` deployment descriptor is dedicated to hosted mode and will be replaced by the `src/main/webapp/WEB-INF` one when building the WAR. This is a simple way to setup custom RPC servlets for testing and not require a complex setup (`DataSource`, `Security`, `Dependency Injection...`) for quick hosted mode testing.

The `gwt:eclipse` goal will setup the `/war` folder as an exploded webapp, ready for launching GWT hosted server. Please note the Google Eclipse Plugin 1.0.1 still uses the deprecated `GWTShell` and does not support servlets defined in the `/war/WEB-INF/web.xml`, so you **must** declare your (test) servlets in the module descriptor using a `servlet` element :

```
<servlet class="org.codehaus.mojo.gwt.test.server.HelloRemoteServlet" path="/org.
```

This one will be ignored after your GWT module has been compiled into JavaScript.

### 5.1.6 Multiproject setup

Big projects may want to split the client-side application in modules, typically using Maven support for multiprojects. The project layout will the 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 the hosted browser with a simple refresh with no requirement to repackage the modules.

### **images/gwt-maven-logo.png**

m2eclipse detecting the gwt module as project reference

(\*) 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.

## 6 Introduction

---

### 6.1 Introduction

This maven plugin is dedicated to the Google Web Toolkit integration in Maven.

Its primary goal is to run the GWT SDK tools from Maven. It uses Maven dependency management feature to avoid manual installation of the GWT SDK on the developer platform. Importing a GWT-based application as Mproject then requires no dedicated manual setup on the developer computer.

Its secondary goal is to nicely integrate with Eclipse and the Google Eclipse plugin. This has many side effects as this plugin use some hard-coded path and conventions that are not Maven compliant as is.

GWT Maven Plugin supports :

- GWT compiler execution for packaging your application into a WAR,
- creating Eclipse launch configuration to launch/debug the GWT hosted mode browser in one click,
- creating Eclipse launch configuration to run/debug GWTTestCase-based unit tests in Eclipse,
- running tests (or test suites) based on GWTTestCase framework as part of Maven build,
- generate boiler code for GWT-RPC Async interface from synchronous server-side interfaces,
- generate GWT internationalization ("i18n") interfaces from bundles,
- run (or debug) the Hosted browser from Maven.

## 7 Setup

---

### 7.1 Setup Maven for GWT development

In order to use gwt-maven-plugin, you will need to configure it using the plugin configuration in your POM, and you will need to decide how you want to handle GWT being present (automatic or manual mode - more below).

#### 7.1.1 Configuring gwt-maven-plugin itself

Regardless of which "mode" you use, **automatic**, or **manual**, you also need to configure gwt-maven-plugin itself, of course.

Also, as a convenience it helps to define a property for the GWT version (so you can change it in one place later to upgrade).

For example:

```
<properties>
  <gwt.version>1.6.4</gwt.version>
</properties>
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>gwt-maven-plugin</artifactId>
  <version>1.1</version>
  <executions>
    <execution>
      <goals>
        <!-- gwt:* goals to be executed during build -->
      </goals>
    </execution>
  </executions>
</plugin>
```

#### 7.1.2 Getting the Plugin

Getting the plugin is simple. It is deployed in maven default repository "central", so you don't have to do anything !

#### 7.1.3 Handling GWT

The plugin needs to know where to find GWT (jars *and* native libraries). There are two ways you can do this.

- 1 Use the Maven dependency plugin to automatically extract GWT native libraries from central repo (automatic mode)
- 2 Download and unpack GWT yourself and set the google.webtoolkit.home plugin configuration property to the location where it is installed (manual mode)

##### 7.1.3.1 Automatic Mode Setup

If you are going to do automatic setup, you need to include the GWT dependencies in your POM. The plugin will detect this version and use it internally.

For example:

```

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

```

Note : Don't define `gwt-dev` as project dependency : this JAR has many common libraries packaged that may conflict with the ones defined for your project, resulting in uncomprehensible `NoSuchMethodErrors`. The `gwt-maven-plugin` will automatically resolve the required dependency based on your declared `gwt-user` dependency.

### 7.1.3.2 Manual Setup

If you are going to setup GWT manually, for example if you want to test latest build or a patched one, you will need to first unpackage it (<http://code.google.com/webtoolkit/download.html>). Then, you will need to tell `gwt-maven-plugin` where GWT is. This can be done with the `gwtHome` plugin configuration parameter *or* the `google.webtoolkit.home` property (this property is unset for automatic mode, but required for manual mode).

```

<property>
  <google.webtoolkit.home>C:/MyCustomGWT</google.webtoolkit.home>
</property>

```

You can also use the `$ env.GWT_HOME` syntax to refer to an OS environment variable.

For manual mode, you also need the GWT dependencies defined (these are required because plugins and goals other than `gwt-maven-plugin` need them, like the standard compiler).

Note that with manual mode, even though the dependencies are still needed, the difference is that the source of the dependencies can be your locally installed GWT location (`GWT_HOME`), rather than a Maven repository, and there is no separate step to unpack the native libraries (they are already in `gwtHome`).

For example:

```

<property>
  <google.webtoolkit.home>${env.GWT_HOME}</google.webtoolkit.home>
</property>
<dependency>
  <groupId>com.google.gwt</groupId>
  <artifactId>gwt-servlet</artifactId>
  <version>custom</version>
  <scope>system</scope>
  <systemPath>${env.GWT_HOME}/gwt-servlet.jar</systemPath>
</dependency>
<dependency>
  <groupId>com.google.gwt</groupId>
  <artifactId>gwt-user</artifactId>
  <version>custom</version>
  <scope>system</scope>
  <systemPath>${env.GWT_HOME}/gwt-user.jar</systemPath>
</dependency>
. . . .
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>gwt-maven-plugin</artifactId>
  <version>1.1</version>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
        <goal>generateAsync</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

#### 7.1.4 Configuring maven-war-plugin

Google Eclipse Plugin **requires** your web application to be created as an exploded WAR in `/war` (hard coded path). To match this requirement, you have two options :

- Move your `src/main/webapp` folder to `/war` and configure `maven-war-plugin` `warSourceDirectory` parameter. This is the in-place setup, as the webapp source will be used to build the exploded WAR layout.
- Configure `maven-war-plugin` to build the webapp in `/war` (`webappDirectory` parameter) instead of the standard `outputDirectory`.

The `gwt-maven-plugin` has a boolean `inplace` parameter to configure the option you choose. The in-place mode is interesting if you use JSPs as no build/packaging/deployment is required to see changes made in the JSP file. Same applies to static files like CSS.

## 8 the /war folder

---

### 8.1 The /war folder

The Google Web Toolkit and The [Google Eclipse Plugin](#) both use the /war folder as base directory for the hosted mode web application. Maven users used to follow the maven-war-plugin convention for src/main/webapp.

#### 8.1.1 Solution 1 : change maven-war-plugin configuration

You can change the maven-war-plugin configuration to use /war as web application source folder. The benefit of this setup is that you can define an alternate web.xml deployment descriptor, and have the hosted mode run a lightweight, test-only server. The war plugin [webXml](#) parameter can then be used to setup the *real* deployment descriptor to be used by your web application.

To use this setup you'll need to configure the gwt-maven-plugin in inline mode. The main side effect is that the /war folder will then contain classes and libs you have to exclude from your SCM (using svn:ignore).

To be even more productive you should also configure the build.outputDirectory to output classes into /war/WEB-INF/classes so that any change to a java source in your favourite IDE is immediately usable in the hosted browser with a simple refresh.

**This is the recommended solution as you will then get a simple, productive environment with simple testing support**

#### 8.1.2 Solution 2 : change maven-war-plugin webappDirectory

Another option is to configure the maven-war-plugin to build the exploded web application in /war, using the [webappDirectory](#) parameter. With this setup you won't see changes in your source folders **but** :

- the hosted mode will run the full web application, with all its frameworks, security and resources constraints
- any change to a configuration file or java source will require the web application to be repackaged using `war:exploded`

## 9 Using the archetype

---

### 9.1 gwt-maven-plugin Archetype Information

The gwt-maven-plugin Archetype creates a bare bones GWT application with all the gwt-maven-plugin setup and structure in place (a template).

#### 9.1.1 Using the Archetype

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=1.1 \  
  -DgroupId=myGroupId \  
  -DartifactId=myArtifactId
```

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

## 10 Run Hosted Mode

---

### 10.1 Configuring the Hosted mode server

Another aspect of gwt-maven-plugin to be familiar with it can help you configure the embedded Servlet container GWT uses in the Hosted Mode shell - Tomcat Lite or Jetty (since 1.6).

#### 10.1.1 Jetty (GWT 1.6)

Jetty is used as a lightweight servlet engine for GWT 1.6+ Hosted mode.

Jetty requires a standard servlet 2.4 webapp structure to run. To match this requirement, the gwt-maven-plugin uses a hosted mode testing webapp folder : `war`. This convention is set to match the Google Eclipse plugin requirement. The `hostedWebapp` parameter can be used to override this value if you don't care about Google Eclipse Plugin.

This folder must contain the project a ready-to-run exploded WAR web application structure. This means your runtime dependencies must be copied to `WEB-INF/lib` and your classes to `WEB-INF/classes`.

##### 10.1.1.1 Using Eclipse

Project dependencies are copied by the plugin when generating the Hosted mode launch scripts ( `mvn gwt:eclipse` ). To match the second requirement, configure your POM to set the `outputDirectory` to `war/WEB-INF/classes`. Don't forget to exclude both folders from your SCM, using `svn:ignore` property or equivalent !

With this setup, when you change some java source code, your IDE compiler will update the webapp classes folder and you can see changes in the hosted browser with a simple refresh.

##### 10.1.1.2 Using command line

You can use `mvn gwt:run` to launch the hosted mode from command line. In such case, the plugin will copy runtime dependencies to `${hostedWebapp}/WEB-INF/lib` and your compiled classes to `${hostedWebapp}/WEB-INF/classes`.

##### 10.1.1.3 Distinction between hostedWebapp and maven-war-plugin webappDirectory

The [maven-war-plugin](#) defines a `webappDirectory` (defaults to `src/main/webapp` to host the static web application structure, including the `web.xml` deployment descriptor).

All server-side components used in hosted mode need to be configured in `${hostedWebapp}/WEB-INF/web.xml`. The hosted mode is designed for testing only, and you should not use it to deploy all your server-side frameworks. Your application will in many cases use a complex setup with a database, many frameworks like Spring and Hibernate, and some other out-of-testing stuff like user authentication.

Having a distinct webapp directory for hosted mode and for the generated war is helpful in this case. This can help to debug the web application as it is easier to use stub RPC services and servlets.

If you want to test your GWT client with a "real" server-side webapp, run the hosted browser with the `-noserver` option set and start your webapp in a standalone process, for example using `tomcat:run` or `jetty:run` goals.

If you still want to use the same webapp directory for hosted mode and target WAR, you can simply change the maven-war-plugin configuration :

```
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <version>2.1-beta-1</version>
  <configuration>
    <warSourceDirectory>war</warSourceDirectory>
  </configuration>
</plugin>
```

### 10.1.2 Tomcat Lite (GWT 1.5)

Tomcat Lite is a stripped down version of Tomcat that works a bit differently than what most developers are used to. It's also an older version of Tomcat, 5.0.28, and some of the configuration is different than 6.x or 5.5.x.

The old documentation about Tomcat Lite has moved [here](#).

# 11 Generate Async

---

## 11.1 Generate Async interfaces for GWT-RPC services

### 11.1.1 About

GWT client will communicate with server-side components using GWT-RPC data serialization protocol. If you're not familiar with this please review [the developer's guide](#).

The GWT-RPC model requires you to define two interfaces : one on server side to handle requests, and a sibling one on client side for invoking the RPC serialization process. The second one is asynchronous, and the two interfaces must match together.

Considering the following Remote Service interface :

```
import com.google.gwt.user.client.rpc.RemoteService;
@RemoteServiceRelativePath( "HelloWorld" )
public interface HelloWorldService
    extends RemoteService
{
    String helloWorld( String message );
}
```

The asynchronous interface used on client-side code is :

```
public interface HelloWorldServiceAsync
{
    String helloWorld( String message, AsyncCallback<String> callback );
}
```

gwt-maven-plugin includes a simple code generator to create all Async interfaces from your server-side remote service interfaces.

### 11.1.2 Generate Async interface for GWT-RPC services

The [generateAsync](#) goal will create a generate-sources folder and Async interface for all RemoteInterface found in the project. To avoid a full scan, only Java source files that matches a pattern are checked. Defaults is to only check `**/*Service.java`, but you can override this convention using the `servicePattern` parameter.

```

<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <configuration>
              <servicePattern>**/gwt/**/*Service.java</servicePattern>
            </configuration>
            <goals>
              <goal>generateAsync</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>

```

### 11.1.3 Utility class

The generated code includes an Utility nested class to retrieve the RemoteService async instance from client-side code. This avoid writing boiler plate code for getting RPC services from your GWT code.

```

public interface HelloWorldServiceAsync
{
    String helloWorld( String message, AsyncCallback<String> callback );
    public static class Util
    {
        public static ContactPrefereServiceAsync getInstance()
        ...
    }
}

```

This utility class must know the server URI the Remote Service is exposed. The plugin will use `@RemoteServiceRelativePath` annotation on the service interface to set the URI in this utility class. In previous example, the service will be binded to URI `[module]/HelloWorld`.

If no annotation is set, the service URI is constructed from interface name applying the `rpcPattern` format. Setting this parameter to " `.rpc`" will create an Util class to bind the service to URI `[module]/HelloWorldService.rpc`.

The `generateAsync` goal also has a `failOnError` parameter that can be helpfull is you have issue with the generator.

## 12 Internationalization

---

### 12.1 Generate i18n interfaces for message bundles

#### 12.1.1 About

GWT uses standard java message bundles for internationalization (i18n), but use the compile-time multiple-permutation strategy to create a JavaScript output file per supported language. On client-side code, developers use Message interface to read String from the bundle. Such interfaces are generated by a SDK tool.

If you're not familiar with this please review [i18n interfaces generator](#) documentation.

#### 12.1.2 gwt-maven-plugin i18n goal

The plugin can be used to generate such Message interface. Just add the required goal and configuration to plugin execution :

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <goals>
              <goal>i18n</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <resourceBundle>com.mycompany.gwt.Bundle</resourceBundle>
        </configuration>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

#### 12.1.3 Declare resource bundles

The `resourceBundle` parameter is used to declare your application bundle for i18n processing. If your application uses more than one bundle, you can nest multiple `resourceBundle` elements :

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <goals>
              <goal>i18n</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <resourceBundles>
            <resourceBundle>com.mycompany.gwt.Bundle1</resourceBundle>
            <resourceBundle>com.mycompany.gwt.Bundle2</resourceBundle>
            <resourceBundle>...</resourceBundle>
          </resourceBundles>
        </configuration>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

#### 12.1.4 Choose the message API to be generated

By default, the `i18nCreator` is configured to generate interface based on the `Messages` interface.

If you prefer to use the `Constants` class as base class, set the `messages` parameter to `false`.

You can also configure the plugin using the `constantsWithLookup` parameter to use `ConstantsWithLookup` interface.

For more information on distinctions between those interfaces, please review the GWT `i18n` documentation.

## 13 Testing

---

### 13.1 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 during the "test" phase in order to support `GWTTestCase` and `GWTTestSuite` derived GWT tests.

It's a long story as to why this is needed (having to do with classpath inspection and setup issues inside `GWTTestCase/JUnitShell`), but the regular Maven Surefire testing plugin does not work for `GWTTestCase` tests (at least not with any configuration we have tried, and we have given it a lot of efforts).

Using this special testing support though, requires that you know a few key things, as outlined below:

#### 13.1.1 Invoking tests

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 test`.

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <goals>
              <goal>test</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

#### 13.1.2 Separate GwtTest 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>2.4.3</version>
  <configuration>
    <excludes>
      <exclude>**/*GwtTest.java</exclude>
    </excludes>
  </configuration>
</plugin>

```

A simpler way to separate classic and GWT tests is to name latests GwtTest "Something.java" - they *start with* "GwtTest". Surefire looks for tests that are named Something"Test".java by default - they *end with* "Test".

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

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

```

<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>gwt-maven-plugin</artifactId>
  <version>1.1</version>
  <configuration>
    <includes>**/CustomPattern*.java</includes>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>test</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

### 13.1.3 Test results output

The plugin mimics as far as possible standard Surefire execution and contributes to execution report. If you use the `surefire-report-plugin`, you will see the GwtTests result included in generated project site.

### 13.1.4 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). <http://google-web-toolkit.googlecode.com/svn/javadoc/1.5/com/google/gwt/junit/tools/GWTTestSuite.html> `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)

### 13.1.5 Run `GWTTestCase` during "test" phase

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. To do this, you just need to define a dedicated `execution` and to override the default phase

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <id>test</id>
            <goals>
              <goal>test</goal>
            </goals>
            <phase>test</phase>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

## 14 Compile and Package

---

### 14.1 Compile GWT application into JavaScript

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 "process-classes" phase to run as late as possible.

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <configuration>
              <module>com.mycompany.gwt.Module</module>
            </configuration>
            <goals>
              <goal>compile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

#### 14.1.1 Configure GWT modules

The `module` parameter 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.

#### 14.1.2 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.

### 14.1.3 Compilation process failing

You may get compilation errors due to `OutOfMemoryException` or `StackOverflowException`. The compilation and permutation process used by `GWTCCompiler` 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>1.1</version>
        <executions>
          <execution>
            <configuration>
              <extraJvmArgs>-Xmx512M -Xss1024k</extraJvmArgs>
            </configuration>
            <goals>
              <goal>compile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

### 14.1.4 Compiler output directory

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

**Standard** uses simple integration of `GWTCCompiler` 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 `tomcat:run` or `jetty:run` goals. Those plugins don't require any packaging to launch the webapp, and handle nicelly 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.

## 15 Multi-project setup

----- GWT in a multi-project setup ----- Stefan HÃ¼bner (sthuebner@googlemail.com) -----

### 15.1 Multi-Project Setup

Often, larger Maven projects will be divided into sub projects. How to use GWT-maven-plugin in such a setup is described in this section.

First, we will setup a basic Maven project structure consisting of two subprojects: one containing domain code and another one containing the actual GWT application. After this has been set up, we will see, how Maven packages the application using the GWT-maven-plugin.

#### 15.1.1 Introduction

In Maven sub projects are called modules. Maven-modules support the concept of **project aggregation**. In the context of a GWT application, like in a normal web application, a common example could be to separate GUI functionality from domain functionality (among others).

We will use **this plugin's reactor it-test** as a sample project layout (note: this is not a comprehensive example. It just demonstrates the basic principles):

```

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

The reactor project contains two subprojects - jar and war. jar contains a Domain GWT module consisting of it's module descriptor Domain.gwt.xml and a User class. The war subproject contains another GWT module called Hello consisting of it's module descriptor Hello.gwt.xml and a Hello class.

**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.

## 15.1.2 Necessary Steps

### 15.1.2.1 Subproject: jar

Here's how the jar subproject looks like. We'll define a `org.codehaus.mojo.gwt.test.domain.User` class:

```
package org.codehaus.mojo.gwt.test.domain;
public class User
{
    public String sayHello()
    {
        return "Hello";
    }
}
```

This class will be living in a GWT module called `org.codehaus.mojo.gwt.test.Domain`:

```
<module>
  <inherits name="com.google.gwt.user.User" />
  <source path="domain" />
</module>
```

The last step is to setup the `pom.xml`. Please note the special `build/resources` declaration:

```

<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.
<modelVersion>4.0.0</modelVersion>
<parent>
  <groupId>org.codehaus.mojo.gwt.it</groupId>
  <artifactId>reactor</artifactId>
  <version>1.0-SNAPSHOT</version>
</parent>
<artifactId>domain</artifactId>
<packaging>jar</packaging>
<build>
  <!--
    sources need to be bundled with the jar,
    so they are visible to GWT's compiler
  -->
  <!--
    You can either setup a resource to point to your java sources ...
<resources>
  <resource>
    <directory>src/main/java</directory>
    <includes>
      <include>/**/*.java</include>
      <include>/**/*.gwt.xml</include>
    </includes>
  </resource>
</resources>
  -->
<plugins>
  <!--
    ... or ask the plugin to detect them based on gwt modules files and copy th
  -->
  <plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>gwt-maven-plugin</artifactId>
    <executions>
      <!-- GWT version detected from dependencyManagement -->
      <execution>
        <goals>
          <goal>resources</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>
</project>

```

Sources need to be bundled with the JAR in order for the GWT compiler to be able to compile them during reactor build.

Next is to setup the war subproject.

## 15.1.2.2 Subproject: war

First, here's the class `org.codehaus.mojo.gwt.test.client.Hello`:

```

package org.codehaus.mojo.gwt.test.client;
import org.codehaus.mojo.gwt.test.domain.User;
import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
public class Hello
    implements EntryPoint
{
    final HelloServiceAsync service = HelloServiceAsync.Util.getInstance();
    public void onModuleLoad()
    {
        User user = new User();
        final Label l = new Label( "GWT says: " + user.sayHello() );
        RootPanel.get().add( l );
        Button b = new Button( "click me !" );
        RootPanel.get().add( b );
        b.addClickHandler( new ClickHandler()
        {
            public void onClick( ClickEvent event )
            {
                service.sayHello( "hello", new AsyncCallback<String>()
                {
                    public void onFailure( Throwable caught )
                    {
                        l.setText( "RPC failure " + caught.getMessage() );
                        GWT.log( "RPC failure", caught );
                    }
                    public void onSuccess( String result )
                    {
                        l.setText( result );
                    }
                } );
            }
        } );
    }
}

```

It calls Domain's `User` class and displays the result in the root panel.

Next is to setup GWT module `org.codehaus.mojo.gwt.test.Hello`. The important thing to note is that this module inherits the Domain module:

```
<module>
  <inherits name="com.google.gwt.user.User" />
  <inherits name='com.google.gwt.user.theme.standard.Standard' />
  <inherits name="org.codehaus.mojo.gwt.test.Domain" />
  <entry-point class="org.codehaus.mojo.gwt.test.client.Hello" />
  <servlet class="org.codehaus.mojo.gwt.test.server.HelloRemoteServlet" path="/org.
</module>
```

Finally, in the war's POM a dependency to the jar-artifact will be declared and the gwt-maven-plugin needs to be setup:

```

<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.
<modelVersion>4.0.0</modelVersion>
<parent>
  <groupId>org.codehaus.mojo.gwt.it</groupId>
  <artifactId>reactor</artifactId>
  <version>1.0-SNAPSHOT</version>
</parent>
<artifactId>webapp</artifactId>
<packaging>war</packaging>
<dependencies>
  <dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-user</artifactId>
    <scope>provided</scope>
  </dependency>
  <!-- this is the dependency to the "jar"-subproject -->
  <dependency>
    <groupId>org.codehaus.mojo.gwt.it</groupId>
    <artifactId>domain</artifactId>
    <version>1.0-SNAPSHOT</version>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>gwt-maven-plugin</artifactId>
      <executions>
        <!-- GWT version detected from dependencyManagement -->
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>generateAsync</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <runTarget>org.codehaus.mojo.gwt.test.Hello/Hello.html</runTarget>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <version>2.0.2</version>
      <configuration>
        <warSourceDirectory>war</warSourceDirectory>
        <webXml>src/main/webapp/WEB-INF/web.xml</webXml>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>

```

Since the `jar`-dependency bundles its sources, they will be visible to GWT's compiler.

#### 15.1.2.3 Main Project: reactor

Eventually, we will glue it all together in the main project's POM:

```

<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.
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.mojo.gwt.it</groupId>
  <artifactId>reactor</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <properties>
    <!-- you will need to replace this with the correct version -->
    <gwtPluginVersion>@pom.version@</gwtPluginVersion>
    <gwtVersion>1.6.4</gwtVersion>
  </properties>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>gwt-maven-plugin</artifactId>
          <version>${gwtPluginVersion}</version>
        </plugin>
        <plugin>
          <artifactId>maven-war-plugin</artifactId>
          <version>2.1-alpha-2</version>
        </plugin>
      </plugins>
    </pluginManagement>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>com.google.gwt</groupId>
        <artifactId>gwt-user</artifactId>
        <version>${gwtVersion}</version>
        <scope>provided</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <modules>
    <module>jar</module>
    <module>war</module>
  </modules>
</project>

```

Note the two subprojects are declared as modules here.

### 15.1.3 Compiling the Application

In order to compile the whole project Maven can be invoked like this:

```
reactor$ mvn clean package
```

Since GWT-maven-plugin's `compile-goal` is bound to Maven's `process-classes` lifecycle phase, the GWT compiler will compile all client code into JavaScript. This code will be packaged along with any other resources into the resulting WAR bundle.

You'll find the packaged application in `war/target/war-1.0-SNAPSHOT.war`

## 16 Writing a GWT library

---

### 16.1 Writing a GWT library

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.

#### 16.1.1 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>1.1</version>
  <executions>
    <execution>
      <goals>
        <goal>resources</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

## 17 Productivity Tip

---

### 17.1 Productivity tip for multi-project setup

#### 17.1.1 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.

#### 17.1.2 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.

#### 17.1.3 Solution

The solution to those two issues is to create a profile in your pom (or settings.xml) which is only activated 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.

## 18 compile a GWT module

---

GWT-maven-plugin will run the GWT compiler when the project is built. If your project defines a single module, use the following configuration:

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <configuration>
              <module>com.mycompany.gwt.Module</module>
            </configuration>
            <goals>
              <goal>compile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

You can also configure compilation for multiple modules by nesting them inside a `<modules>` element. If none is set, the plugin will scan project source and resources directories for `.gwt.xml` module files.

```

<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <configuration>
              <modules>
                <module>com.mycompany.gwt.Module1</module>
                <module>com.mycompany.gwt.Module2</module>
                <module>...</module>
              </modules>
            </configuration>
            <goals>
              <goal>compile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>

```

You can also ignore the "module" parameter, so that the plugin will scan your project for `gwt.xml` module files.

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 override this behaviour by setting the "outputDirectory" parameter. For example, you may want to configure output to `$ basedir/src/main/webapp` if you have configured your servlet container to use the "inplace" mode of the war plugin.

## 19 generate async interfaces

---

GWT-maven-plugin can generate asynchronous interface for your GWT-RPC services. Considering the following Server-side RPC interface:

```
import com.google.gwt.user.client.rpc.RemoteService;
public interface HelloWorldService
    extends RemoteService
{
    String helloWorld( String message );
}
```

.. the plugin will automatically generate the asynchronous interface used on client-side code. Thanks to this feature, you don't have to write and maintain this boring code. As a bonus, the generated code includes an utility class to retrieve the RPC proxy from client-side code:

```
@RemoteServiceRelativePath( "HelloWorld" )
public interface HelloWorldServiceAsync
{
    String helloWorld( String message, AsyncCallback<String> callback );
    public static class Util
    {
        public static ContactPrefererServiceAsync getInstance()
        ...
    }
}
```

The generated interface includes a nested Util class to help retrieve an HelloWorldServiceAsync implementation.

The plugin will use @RemoteServiceRelativePath annotation on the service interface to set the service URI in this utility class. In previous example, the service will be bound to URI [module]/HelloWorld. If not present, the service URI is constructed from interface name applying the rpcPattern format. Setting this parameter to ".rpc" will create an Util class to retrieve the service on URI [module]/HelloWorldService.rpc.

To enable this feature, simply include the generateAsync goal in your POM.xml:

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <goals>
              <goal>generateAsync</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

To avoid full scan of project sources, the plugin uses a naming convention for RPC services. By default, it only checks `**/*Service.java` source files. You can override this convention using the `servicePattern` parameter.

The `rpcExtension` is used to configure the URL pattern used on the server to publish the GWT-RPC services. It will be used by the generator to create helper code (see later).

The `failOnError` parameter can also be helpful if you have issue with the generator.

## 20 generate i18n bundles interfaces

---

GWT-maven-plugin can run the GWT [i18n interfaces generator](#) for your messages bundles. To enable this feature, simply include the `i18n` goal in your `pom.xml`:

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <goals>
              <goal>i18n</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <i18nMessagesBundle>com.mycompany.gwt.Bundle</i18nMessagesBundle>
        </configuration>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

If your application uses more than one bundle, you can nest multiple `i18nMessagesBundle` elements:

```
<project>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>1.0</version>
        <executions>
          <execution>
            <goals>
              <goal>i18n</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <i18nMessagesBundles>
            <i18nMessagesBundle>com.mycompany.gwt.Bundle1</i18nMessagesBundle>
            <i18nMessagesBundle>com.mycompany.gwt.Bundle2</i18nMessagesBundle>
            <i18nMessagesBundle>...</i18nMessagesBundle>
          </i18nMessagesBundles>
        </configuration>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>
```

You can use the `i18n` goal to generate interfaces for either `Messages`, `Constants` and `ConstantsWithLookup` interfaces, using the associated `i18nMessagesBundles`, `i18nConstantsBundles` and `i18nConstantsWithLookupBundles` parameters