

Implementing Maven Plugins

Implementing Maven plugins has never been Groovier!

- [Groovy Mojos](#)
 - [A Simple Groovy Mojo](#)
- [Building Plugins](#)
 - [Project Definition](#)
 - [Mojo Parameters](#)
- [Putting More Groove into your Mojo](#)
 - [Using ant](#)
 - [Using fail\(\)](#)
- [gmaven-archetype-mojo Archetype](#)
- [gmaven-plugin Packaging](#)
- [Resources](#)
 - [Mojo Development](#)
 - [Groovy Development](#)

Groovy Maven plugins are very similar to Java Maven plugins. Actually a Groovy Maven plugin is compiled into Java byte-code, and once built, Maven can not tell the difference between a plugin which has been implemented in Java or Groovy.

For the most part, the existing guide for [Developing Maven plugins with Java](#) will also apply to developing plugins with Groovy. There are some differences though, which are covered here which can make Maven plugin development with Groovy better, faster, stronger 😊

Be sure to read over the documentation for [Building Groovy Projects](#). A Groovy Maven plugin is a Groovy project, so most of the information there is relevant here as well.

Groovy Mojos

Just like Java-based Mojo's at its simplest, a Groovy Mojo consists of a single class. For more complicated plugins you are free to use as many classes as needed of course, just like Java... and you can even write Java classes too if you need too.

Groovy Mojo implementations are denoted by the `@goal` Javadoc annotation on the class (er, just like Java). Actually, **all** of the Javadoc annotations which are supported by Java plugins are also supported by Groovy plugins. This is because, when building a Groovy project, the sub-generator spits out Java sources with Javadocs intact, which the [Maven Plugin Plugin](#) then parses for annotations 😊

A Simple Groovy Mojo

Here is our simple Mojo class which has no parameters and spits out a relatively meaningless string via logging:

```
package sample.plugin

import org.codehaus.gmaven.mojo.GroovyMojo

/**
 * Says "Hi" to the user... er well not
 really :-P.
 *
 * @goal sayhi
 */
public class GreetingMojo
    extends GroovyMojo
{
    void execute() {
        log.info('Groovy baby!')
    }
}
```

There are a few minor points to note here. First, we are referencing the plugins logger via `log.info()`, where in Java one would need to `getLog().info()`.

Second, there is no need to mark the `execute()` method as throwing any exceptions, though you can still throw any exceptions you need to. You can of course declare what your methods are throwing if you want to.

Lastly, this example mojo extends from [org.codehaus.groovy.maven.mojo.GroovyMojo](#), which is recommended for most Groovy mojos. Of course you can always use `org.apache.maven.plugin.AbstractMojo` at the cost of some additional bits of happiness which help make your Mojo's groovier.

Building Plugins

Project Definition

Groovy plugins use the same Java plugin descriptor extractor. The Java extractor uses the generated stubs to build the descriptor, so be sure to invoke the `generateStubs` goal.

Instead of depending on `maven-plugin-api` Groovy plugins depend on `gmaven-mojo`, which picks up the required Maven dependencies and the default Groovy runtime provider.

And of course, we need to hook up the `gmaven-plugin` to compile the Groovy sources into class files (as well as a few others).

Below is a POM for the simple sample groovy mojo:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>sample.plugin</groupId>

  <artifactId>maven-hello-plugin</artifactId>
  <packaging>maven-plugin</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Sample Maven Plugin</name>

  <dependencies>
    <dependency>

<groupId>org.codehaus.gmaven</groupId>

<artifactId>gmaven-mojo</artifactId>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>

<groupId>org.codehaus.gmaven</groupId>

<artifactId>gmaven-plugin</artifactId>
      <executions>
        <execution>
```

```
                <goals>

<goal>generateStubs</goal>

<goal>compile</goal>

<goal>generateTestStubs</goal>

<goal>testCompile</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
```

```
        </plugins>
    </build>
</project>
```

Dependency and Plugin Versions

To use the above example you must configure the `<version>` element for the dependencies and plugins.

Once you have your pom setup then you can build the plugin in the normal way via:

```
mvn install
```

Mojo Parameters

Mojo parameters work **exactly** the same as they do for Java plugins. Simply define a field in your Mojo implementation and annotate the field with a `@parameter` Javadoc tag.

```
/**
 * The greeting to display.
 *
 * @parameter default-value="Groovy baby!"
 */
private String greeting
```

Configuration of Mojo parameters is... as you might expect, the same as with Java Mojos:

```
<plugin>
  <groupId>sample.plugin</groupId>

  <artifactId>maven-hello-plugin</artifactId>
  <configuration>
    <greeting>what say, you, we go out
on the down and swing, baby? Yea</greeting>
  </configuration>
</plugin>
```

The main thing to note about Groovy Mojos and parameters, is that fields should be typed so that Maven (or well, Plexus) can inject objects of the proper type. Using the `def` keyword is the same as typing the field as `Object` which Maven will probably still inject just fine, but it won't perform any conversion.

Putting More Groove into your Mojo

Using ant

Using `ant` allows your Mojo access to any [Ant](#) tasks. Groovy Mojo's which extend from `GroovyMojo` have access to an `AntBuilder` instance bound to the `ant` field.

This field is lazily initialized when you first reference it.

Below are some small examples of using `ant` to perform common tasks, but really the sky is the limit on what you can do. See the [Ant User Manual](#) for more tasks you can execute 😊

NOTE

The following examples assume that your Mojo implementation has already defined a project property as in:

```
/**
 * @parameter expression="${project}"
 * @required
 * @readonly
 */
org.apache.maven.project.MavenProject project
```

Touching a File

```
ant.touch(file:
"${project.build.directory}/stamp")
```

Copying Files

```
def dir =
"${project.build.directory}/backup"
ant.mkdir(dir: dir)
ant.copy(todir: dir) {
    fileset(dir:
"${project.build.outputDirectory}") {
        include(name: '**/*')
    }
}
```

Execute Something

```
def propname = 'lsout'
ant.exec(executable: '/bin/ls',
outputproperty: propname)
def value =
ant.antProject.properties[propname]
```

Using fail()

Most mojos need to report back some failure status, which is normally done by throwing a `MojoExecutionException`. Groovy Mojos can simply invoke the `fail()` method, which handles the details of throwing for you.

Failing with a simple string:

```
fail("That ain't no woman! It's a man,  
man!")
```

Failing with an exception detail:

```
try {  
    ....  
}  
catch (Exception e) {  
    fail(e)  
}
```

Failing with an exception detail and a message:

```
try {  
    ....  
}  
catch (Exception e) {  
    fail("She's the village bicycle!  
Everybody's had a ride.", e)  
}
```

gmaven-archetype-mojo Archetype

To help get Groovy plugins started faster, you can use the `gmaven-archetype-mojo`. This will create a new project with the basic POM configuration and an example Groovy-based Mojo class to get you started quickly:

```
mvn archetype:generate
-DarchetypeGroupId=org.codehaus.gmaven.archetypes
types
-DarchetypeArtifactId=gmaven-archetype-mojo
-DarchetypeVersion=1.0-rc-2
```

✓ TIP

To use a specific version of an archetype specify `-DarchetypeVersion=<VERSION>`.

The [Maven Archetype Plugin](#) will ask a few questions about your new project:

```
[INFO] [archetype:generate]
...
Define value for groupId: :
org.mycompany.myproject
Define value for artifactId: :
example-maven-plugin
Define value for version: : 1.0-SNAPSHOT
Define value for package: :
org.mycompany.myproject.example
Confirm properties configuration:
name: Example Maven Plugin
groupId: org.mycompany.myproject
artifactId: example-maven-plugin
version: 1.0-SNAPSHOT
package: org.mycompany.myproject.example
  Y: : y
...
[INFO]
-----
-----
[INFO] BUILD SUCCESSFUL
[INFO]
-----
-----
...

```

 **NOTE**

Please ignore any `ReferenceException` warnings about `${...}`, they are harmless.

The above example would have created the following project structure:

```
example-maven-plugin
example-maven-plugin/pom.xml
example-maven-plugin/src
example-maven-plugin/src/main
example-maven-plugin/src/main/groovy
example-maven-plugin/src/main/groovy/org
example-maven-plugin/src/main/groovy/org/myc
ompany
example-maven-plugin/src/main/groovy/org/myc
ompany/myproject
example-maven-plugin/src/main/groovy/org/myc
ompany/myproject/example
example-maven-plugin/src/main/groovy/org/myc
ompany/myproject/example>HelloMojo.groovy
```

gmaven-plugin Packaging

 TODO

TODO

Resources

Mojo Development

- [Mojo API specification](#)
- [Guide to Developing Java Plugins](#)

Groovy Development

- [Using Ant from Groovy](#)
- [Ant User Manual](#)
- [Groovy JDK Methods](#)
- [Groovy Javadocs](#)