Guide to Configuring Plug-ins

Introduction

In Maven, there are the build and the reporting plugins:

- **Build plugins** will be executed during the build and then, they should be configured in the `<build/>` element.
- **Reporting plugins** will be executed during the site generation and they should be configured in the `<reporting/>` element.

All plugins should have minimal required information (/ref/current/maven-model/maven.html#class_plugin): `groupId`, `artifactId` and `version`.

Important Note: It is recommended to always define each version of the plugins used by the build to guarantee the build reproducibility. A good practice is to specify them in the `<build><pluginManagement/></build>` elements for **each** build plugins (generally, you will define a <pluginManagement/> element in a parent POM). For reporting plugins, you should specify each version in the `<reporting><plugins/></reporting>` elements (and surely in the `<build><pluginManagement/></build>` elements too).

Generic Configuration

Maven plugins (build and reporting) are configured by specifying a `<configuration>` element where the child elements of the `<configuration>` element are mapped to fields, or setters, inside your Mojo (remember that a plug-in consists of one or more Mojos where a Mojo maps to a goal). Say, for example, we had a Mojo that performed a query against a particular URL, with a specified timeout and list of options. The Mojo might look like the following:

```
/**
 * @goal query
 */
public class MyQueryMojo
    extends AbstractMojo
{
    @Parameter(property = "query.url", required = true)
    private String url;

    @Parameter(property = "timeout", required = false, defaultValue = "50")
    private int timeout;

    @Parameter(property = "options")
    private String[] options;

    public void execute()
        throws MojoExecutionException
    {
        ...
    }
}
```

To configure the Mojo from your POM with the desired URL, timeout and options you might have something like the following:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-myquery-plugin</artifactId>
        <version>1.0</version>
        <configuration>
          <url>http://www.foobar.com/query</url>
          <timeout>10</timeout>
          <options>
            <option>one</option>
            <option>two</option>
            <option>three</option>
          </options>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

As you can see the elements in the configuration match the names of the fields in the Mojo. The configuration mechanism Maven employs is very similar to the way XStream (http://x-stream.github.io/) works where elements in XML are mapped to objects. So from the example above you can see that the mapping is pretty straight forward the `url` element maps to the `url` field, the `timeout` element maps to the `timeout` field and the `options` element maps to the `options` field. The mapping mechanism can deal with arrays by inspecting the type of the field and determining if a suitable mapping is possible.

For mojos that are intended to be executed directly from the CLI, their parameters usually provide a means to be configured via system properties instead of a `<configuration>` section in the POM. The plugin documentation for those parameters will list an *expression* that denotes the system properties for the configuration. In the mojo above, the parameter `url` is associated with the expression `${query.url}`, meaning its value can be specified by the system property `query.url` as shown below:

```
mvn myquery:query -Dquery.url=http://maven.apache.org
```

Note that the name of the system property does not necessarily match the name of the mojo parameter. While this is a rather common practice, you will often notice plugins that employ some prefix for the system properties to avoid name clashes with other system properties. Though rarely, there are also plugin parameters that (e.g. for historical reasons) employ system properties which are completely unrelated to the parameter name. So be sure to have a close look at the plugin documentation.

Help Goal

Most recent Maven plugins have a `help` goal that prints a description of the plugin and its parameters and types. For instance, to see help for the javadoc goal, type:

```
mvn javadoc:help -Ddetail -Dgoal=javadoc
```

And you will see all parameters for the javadoc:javadoc goal, similar to this page (/plugins/maven-javadoc-plugin/javadoc-mojo.html).

Configuring Parameters

Mapping Simple Objects

Mapping simple types, like Boolean or Integer, is very simple. The `<configuration>` element might look like the following:

```
...
<configuration>
  <myString>a string</myString>
  <myBoolean>true</myBoolean>
  <myInteger>10</myInteger>
  <myDouble>1.0</myDouble>
  <myFile>c:\temp</myFile>
  <myURL>http://maven.apache.org</myURL>
</configuration>
...
```

Mapping Complex Objects

Mapping complex types is also fairly straight forward in Maven so let's look at a simple example where we are trying to map a configuration for Person object. The `<configuration/>` element might look like the following:

```
...
<configuration>
  <person>
    <firstName>Jason</firstName>
    <lastName>van Zyl</lastName>
  </person>
</configuration>
...
```

The rules for mapping complex objects are as follows:

- There must be a private field that corresponds to name of the element being mapped. So in our case the `person` element must map to a `person` field in the mojo.
- The object instantiated must be in the same package as the Mojo itself. So if your mojo is in `com.mycompany.mojo.query` then the mapping mechanism will look in that package for an object named `Person`. As you can see the mechanism will capitalize the first letter of the element name and use that to search for the object to instantiate.
- If you wish to have the object to be instantiated live in a different package or have a more complicated name then you must specify this using an `implementation` attribute like the following:

```
...
<configuration>
  <person implementation="com.mycompany.mojo.query.SuperPerson">
    <firstName>Jason</firstName>
    <lastName>van Zyl</lastName>
  </person>
</configuration>
...
```

Mapping Collections

The configuration mapping mechanism can easily deal with most collections so let's go through a few examples to show you how it's done:

Mapping Lists

Mapping lists works in much the same way as mapping to arrays where you a list of elements will be mapped to the List. So if you have a mojo like the following:

```
public class MyAnimalMojo
    extends AbstractMojo
{
    @Parameter(property = "animals")
    private List animals;

    public void execute()
        throws MojoExecutionException
    {
        ...
    }
}
```

Where you have a field named `animals` then your configuration for the plug-in would look like the following:

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-myanimal-plugin</artifactId>
        <version>1.0</version>
        <configuration>
          <animals>
            <animal>cat</animal>
            <animal>dog</animal>
            <animal>aardvark</animal>
          </animals>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Where each of the animals listed would be entries in the `animals` field. Unlike arrays, collections have no specific component type. In order to derive the type of a list item, the following strategy is used:

1. If the XML element contains an `implementation` hint attribute, that is used
2. If the XML tag contains a `.`, try that as a fully qualified class name
3. Try the XML tag (with capitalized first letter) as a class in the same package as the mojo/object being configured
4. If the element has no children, assume its type is `String`. Otherwise, the configuration will fail.

Mapping Maps

In the same way, you could define maps like the following:

```java
...
    @Parameter(property = "myMap")
    private Map myMap;
...
```

```xml
...
  <configuration>
    <myMap>
      <key1>value1</key1>
      <key2>value2</key2>
    </myMap>
  </configuration>
...
```

Mapping Properties

Properties should be defined like the following:

```java
...
    @Parameter(property = "myProperties")
    private Properties myProperties;
...
```

```
...
  <configuration>
    <myProperties>
      <property>
        <name>propertyName1</name>
        <value>propertyValue1</value>
      <property>
      <property>
        <name>propertyName2</name>
        <value>propertyValue2</value>
      <property>
    </myProperties>
  </configuration>
...
```

Configuring Build Plugins

The following is only to configure Build plugins in the `<build>` element.

Using the `<executions>` Tag

You can also configure a mojo using the `<executions>` tag. This is most commonly used for mojos that are intended to participate in some phases of the build lifecycle (../introduction/introduction-to-the-lifecycle.html). Using `MyQueryMojo` as an example, you may have something that will look like:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-myquery-plugin</artifactId>
        <version>1.0</version>
        <executions>
          <execution>
            <id>execution1</id>
            <phase>test</phase>
            <configuration>
              <url>http://www.foo.com/query</url>
              <timeout>10</timeout>
              <options>
                <option>one</option>
                <option>two</option>
                <option>three</option>
              </options>
            </configuration>
            <goals>
              <goal>query</goal>
            </goals>
          </execution>
          <execution>
            <id>execution2</id>
            <configuration>
              <url>http://www.bar.com/query</url>
              <timeout>15</timeout>
              <options>
                <option>four</option>
                <option>five</option>
                <option>six</option>
              </options>
            </configuration>
            <goals>
              <goal>query</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

The first execution with id "execution1" binds this configuration to the test phase. The second execution does not have a `<phase>` tag, how do you think will this execution behave? Well, goals can have a default phase binding as discussed further below. If the goal has a default phase binding then it will execute in that phase. But if the goal is not bound to any lifecycle phase then it simply won't be executed during the build lifecycle.

Note that while execution id's have to be unique among all executions of a single plugin within a POM, they don't have to be unique across an inheritance hierarchy of POMs. Executions of the same id from different POMs are merged. The same applies to executions that are defined by profiles.

How about if we have a multiple executions with different phases bound to it? How do you think will it behave? Let us use the example POM above again, but this time we shall bind `execution2` to a phase.

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        ...
        <executions>
          <execution>
            <id>execution1</id>
            <phase>test</phase>
            ...
          </execution>
          <execution>
            <id>execution2</id>
            <phase>install</phase>
            <configuration>
              <url>http://www.bar.com/query</url>
              <timeout>15</timeout>
              <options>
                <option>four</option>
                <option>five</option>
                <option>six</option>
              </options>
            </configuration>
            <goals>
              <goal>query</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

If there are multiple executions bound to different phases, then the mojo is executed once for each phase indicated. Meaning, `execution1` will be executed applying the configuration setup when the phase of the build is test, and `execution2` will be executed applying the configuration setup when the build phase is already in install.

Now, let us have another mojo example which shows a default lifecycle phase binding.

```java
/**
 * @goal query
 * @phase package
 */
public class MyBoundQueryMojo
    extends AbstractMojo
{
    @Parameter(property = "query.url", required = true)
    private String url;

    @Parameter(property = "timeout", required = false, defaultValue = "50")
    private int timeout;

    @Parameter(property = "options")
    private String[] options;

    public void execute()
        throws MojoExecutionException
    {
        ...
    }
}
```

From the above mojo example, `MyBoundQueryMojo` is by default bound to the package phase (see the `@phase` notation). But if we want to execute this mojo during the install phase and not with package we can rebind this mojo into a new lifecycle phase using the `<phase>` tag under `<execution>`.

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-myquery-plugin</artifactId>
        <version>1.0</version>
        <executions>
          <execution>
            <id>execution1</id>
            <phase>install</phase>
            <configuration>
              <url>http://www.bar.com/query</url>
              <timeout>15</timeout>
              <options>
                <option>four</option>
                <option>five</option>
                <option>six</option>
              </options>
            </configuration>
            <goals>
              <goal>query</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Now, `MyBoundQueryMojo` default phase which is package has been overridden by install phase.

Note: Configurations inside the `<executions>` element used to differ from those that are outside `<executions>` in that they could not be used from a direct command line invocation because they were only applied when the lifecycle phase they were bound to was invoked. So you had to move a configuration section outside of the executions section to apply it globally to all invocations of the plugin. Since Maven 3.3.1 this is not the case anymore as you can specify on the command line the execution id for direct plugin goal invocation. Hence if you want to run the above plugin and it's specific execution1's configuration from the command-line, you can execute:

```
mvn myqyeryplugin:queryMojo@execution1
```

Using the `<dependencies>` Tag

You could configure the dependencies of the Build plugins, commonly to use a more recent dependency version.

For instance, the Maven Antrun Plugin version 1.2 uses Ant version 1.6.5, if you want to use the latest Ant version when running this plugin, you need to add `<dependencies>` element like the following:

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.2</version>
        ...
        <dependencies>
          <dependency>
            <groupId>org.apache.ant</groupId>
            <artifactId>ant</artifactId>
            <version>1.7.1</version>
          </dependency>
          <dependency>
            <groupId>org.apache.ant</groupId>
            <artifactId>ant-launcher</artifactId>
            <version>1.7.1</version>
          </dependency>
         </dependencies>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Using the `<inherited>` Tag In Build Plugins

By default, plugin configuration should be propagated to child POMs, so to break the inheritance, you could use the `<inherited>` tag:

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.2</version>
        <inherited>false</inherited>
        ...
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Configuring Reporting Plugins

The following is only to configure Reporting plugins in the `<reporting>` element.

Using the `<reporting>` Tag VS `<build>` Tag

Configuring a reporting plugin in the <reporting> or <build> elements in the pom does **NOT** have the same behavior!

`mvn site`
: It uses **only** the parameters defined in the <configuration> element of each reporting Plugin specified in the <reporting> element, i.e. `site` always **ignores** the parameters defined in the <configuration> element of each plugin specified in <build>.

`mvn aplugin:areportgoal`
: It uses **firstly** the parameters defined in the <configuration> element of each reporting Plugin specified in the <reporting> element; if a parameter is not found, it will look up to a parameter defined in the <configuration> element of each plugin specified in <build>.

Using the `<reportSets>` Tag

You can configure a reporting plugin using the `<reportSets>` tag. This is most commonly used to generate reports selectively when running `mvn site`. The following will generate only the project team report.

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <version>2.1.2</version>
        <reportSets>
          <reportSet>
            <reports>
              <report>project-team</report>
            </reports>
          </reportSet>
        </reportSets>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

Notes:

1. To exclude all reports, you need to use:
   ```
     <reportSets>
       <reportSet>
         <reports/>
       </reportSet>
     </reportSets>
   ```
2. Refer to each Plugin Documentation (i.e. plugin-info.html) to know the available report goals.

Using the `<inherited>` Tag In Reporting Plugins

Similar to the build plugins, to break the inheritance, you can use the `<inherited>` tag:

```xml
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <version>2.1.2</version>
        <inherited>false</inherited>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```