Maven plugins

6.1 Introduction to plugins

Standard lifecycles can be enhanced with functionality using Maven plugins. Plugins allow you to insert new steps into the standard cycle (for example, distribution to the application server) or extend existing steps.

Plugins in Maven are not something extraordinary, on the contrary, they are the most common and frequently encountered thing. After all, if you want to set some nuances of building your project, then you need to specify the necessary information in pom.xml. And the only way to do this is to write a “plugin”.

Since plugins are just as much artifacts as dependencies, they are described in much the same way. Instead of the dependencies section - plugins, instead of dependency - plugin, instead of repositories - pluginRepositories, repository - pluginRepository.

Example:

```
<plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
        <version>2.6</version>
    </plugin>
</plugins>
```

Declaring a plugin in pom.xml allows you to fix the plugin version, as well as set the necessary parameters for it, define various configuration parameters and bind to phases.

In other words, Maven runs certain plugins that do all the work. That is, if we want to teach Maven about special builds of the project, **then we need to add to pom.xml an indication to launch the desired plugin in the required phase and with the required parameters** .

The number of plugins available is very large, there are various plugins that allow you to run a web application directly from maven to test it in a browser, generate resources, and the like. The main task of the developer in this situation is **to find and apply the most appropriate set of plugins** .

6.2 Life cycle and plugins

Very often, a plugin is used to launch some kind of console utility during the execution of a certain phase. Moreover, we can even run a regular Java class (which has a main method, of course).

Example:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.2.1</version>
  <executions>
    <execution>
      <goals>
        <goal>java</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <mainClass>com.example.Main</mainClass>
    <arguments>
      <argument>first-argument</argument>
      <argument>second-argument</argument>
    </arguments>
  </configuration>
</plugin>
```

Usually plugins can be very flexibly configured. All official plugins from Maven developers are very well documented on the official Maven website. For example, for **the maven-compiler-plugin** on the Apache Maven Project page, you can see a list of all the variables that control the plugin. Information on the plugin is available at the link

More important information. Different plugins are called by Maven at different stages of their life cycle. For example, a project describing a swing Java desktop application has different life cycle stages from those that are typical for the development of a web application (war).

Or, for example, when the command “mvn test” is executed, a whole set of steps in the life cycle of the project is initiated: “process-resources”, “compile”, “process-classes”, “process-test-resources”, “test-compile” , test. You can see the mention of these phases in the messages displayed by Maven:

6.3 Goals in Maven - goals

In Maven, there is also such a thing as a goal (goal). goal is sort of like the Maven startup target. The main goals coincide with the main phases:

- **validate;**
- **compile;**
- **test;**
- **package;**
- **verify;**
- **install;**
- **deploy.**

In each phase of the project life cycle, a specific plugin (jar-library) is called, which includes a number of goals (goal)

For example, the "maven-compiler-plugin" plugin contains two targets: compiler:compile for compiling the project's main source code, and compiler:testCompile for compiling tests. Formally, the list of phases can be changed, although this is rarely necessary.

If you need to perform some non-standard actions in a certain phase, then you just need to add the appropriate plugin to pom.xml

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>Name-plugin</artifactId>
  <executions>
    <execution>
      <id>customTask</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>pluginGoal</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

The most important thing in this case is to define for the plugin the name of the “execution/phase” phase, in which you need to embed the call to the goal of the plugin “goal”. For example, you need to generate Java code based on xml. Then you need the “generate-sources” phase, which is placed before the call to the compile phase and is ideal for generating part of the project's sources.