5.3 Running tests from Maven

Once you have used Ant on several projects, you'll notice that most projects almost always need the same Ant scripts (or at least a good percentage). These scripts are easy enough to reuse through cut and paste, but each new project requires a bit of fussing to get the Ant buildfiles working just right. In addition, each project usually ends up having several subprojects, each of which requires you to create and maintain an Ant buildfile.

Maven (http://maven.apache.org/) picks up where Ant leaves off, making it a natural fit for many teams. Like Ant, Maven is a tool for running other tools, but Maven is designed to take tool reuse to the next level. If Ant is a source-building framework, Maven is a source-building *environment*.

5.3.1 Maven the goal-seeker

Behind each target in every Ant buildfile lies a goal. The goal might be to generate the unit tests, to assemble the Javadocs, or to compile the distribution. The driving force behind Maven is that under the hood, each software project almost always does things the same way, following several years of best practices. Most differences are arbitrary, such as whether you call the target output directory `target` or `output`.

Instead of asking developers to write their own targets with tasks, Maven provides ready-to-use plugins to achieve the goals. At the time of this writing, Maven boasts more than 70 plugins. Once Maven is installed (see the sidebar on the next page), you can type **`maven -g`** to get the full list of the available plugins and goals. Reference documentation for the plugins is available at http://maven.apache.org/reference/plugins. The following list describes a few common Maven plugins:

- `jar`—Generates a project jar and deploys it to a local or remote jar repository
- `junit`—Executes JUnit tests
- `site`—Generates a project documentation web site that contains lots of useful reports and project information, in addition to containing any docs you wish to include
- `changelog`—Generates a change log report (CVS changelog, Starteam changelog, and so forth)
- `checkstyle`—Runs Checkstyle on the source code and generates a report
- `clover`—Runs Clover on the source code and generates a Clover report
- `eclipse`—Automatically generates Eclipse project files from the Maven project description
- `ear`—Packages the application as an ear file
- `cactus`—Automatically packages your code, deploys it in a container of your choice, starts the container, and runs Cactus tests (see chapter 8)
- `jboss`—Supports creation of JBoss Server configurations and deployments of war, ear, and EJB-jar in JBoss using a simple copy or JMX

Having well-defined plugins not only provides unprecedented ease of use, it also standardizes project builds, making it easy for developers to go from project to project.

Installing Maven

Installing Maven is a three-step process:

1. Download the latest distribution from http://maven.apache.org/builds/release/ and unzip/untar it in the directory of your choice (for example, `c:\maven` on Windows or `/opt/maven` on UNIX).
2. Define a `MAVEN_HOME` environment variable pointing to where you have installed Maven.
3. Add `MAVEN_HOME\bin` (`MAVEN_HOME/bin` on UNIX) to your `PATH` environment variable so that you can type **`maven`** from any directory.

You are now ready to use Maven. The first time you execute a plugin, make sure your Internet connection is on, because Maven will automatically download from the Web all the third-party jars the plugin requires.

5.3.2 Configuring Maven for a project

Using Ant alone, you describe your build at the level of the tasks. With Maven, you describe your project structure and the plugins use this directory structure, so you don't have to be an Ant wizard to set up your project. Maven handles the wizardry.

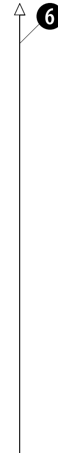
Let's look at a Maven description for a simple project based on the `sampling` project you wrote in chapter 3 and that you ran with Ant earlier in the chapter. The goal is to run your unit tests with Maven.

Configuring Maven for a project requires writing only one file: `project.xml` (also called the POM, short for project object model). It contains the full project description. Listing 5.6 shows the first part of this file, which contains background information about the project.

Listing 5.6 First part of project.xml showing background project information

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<project>
  <pomVersion>3</pomVersion>                              ❶
  <id>junitbook-sampling</id>                             ❷
  <name>JUnit in Action - Sampling JUnit</name>           ❸
  <currentVersion>1.0</currentVersion>                    ❹
  <organization>
    <name>Manning Publications Co.</name>
    <url>http://www.manning.com/</url>
    <logo>http://www.manning.com/front/dance.gif</logo>
  </organization>
  <inceptionYear>2002</inceptionYear>
  <package>junitbook.sampling</package>
  <logo>/images/jia.jpg</logo>

  <description>                                                      ❺
    Chapter 3 presents a sophisticated test case to show how JUnit
    works with larger components. The subject of our case study is
    a component found in many applications: a controller. We
    introduce the case-study code, identify what code to test, and
    then show how to test it. Once we know that the code works as
    expected, we create tests for exceptional conditions, to be
    sure our code behaves well even when things go wrong.
  </description>
  <shortDescription>
    Chapter 3 of JUnit in Action: Sampling JUnit
  </shortDescription>

  <url>http://sourceforge.net/projects/junitbook/</url>

  <developers>                                              ❻
    <developer>
      <name>Vincent Massol</name>
```

```
      <id>vmassol</id>
      <email>vmassol@users.sourceforge.net</email>
      <organization>Pivolis</organization>
      <roles>
        <role>Java Developer</role>
      </roles>
    </developer>
    <developer>
      <name>Ted Husted</name>
      <id>thusted</id>
      <email>thusted@users.sourceforge.net</email>
      <organization>Husted dot Com</organization>
      <roles>
        <role>Java Developer</role>
      </roles>
    </developer>
  </developers>
 [...]
</project>
```



❶ Tell Maven the version of the POM you are using to describe the project. As of this writing, the version to use is 3. Maven uses it to perform automatic migration of old POM versions to the new one if need be.

❷ Define the project ID. Several plugins use this ID to name files that are generated. For example, if you run the `jar` plugin on the project, it generates a jar named `junitbook-sampling-1.0.jar` (`<id>.<currentVersion>.jar`).

❸ Give a human-readable name for your project. It is used, for example, by the `site` plugin, which generates the documentation web site.

❹ This is the current version of your project. For example, the version suffixed to the jar name comes from the definition here.

❺ Describe background information about your project that is used by the `site` plugin for the web site.

❻ Describe the developers working on this project and their roles. This information is used in a report generated by the `site` plugin.

Executing Maven web-site generation

Let's use the Maven `site` plugin to generate the web site and see how the information you have provided is used. Open a command-line prompt in the `sampling/` project directory (see chapter 3, section 3.4 for details of setting up the project directory structure) and enter **`maven site`**, as shown in figure 5.3.

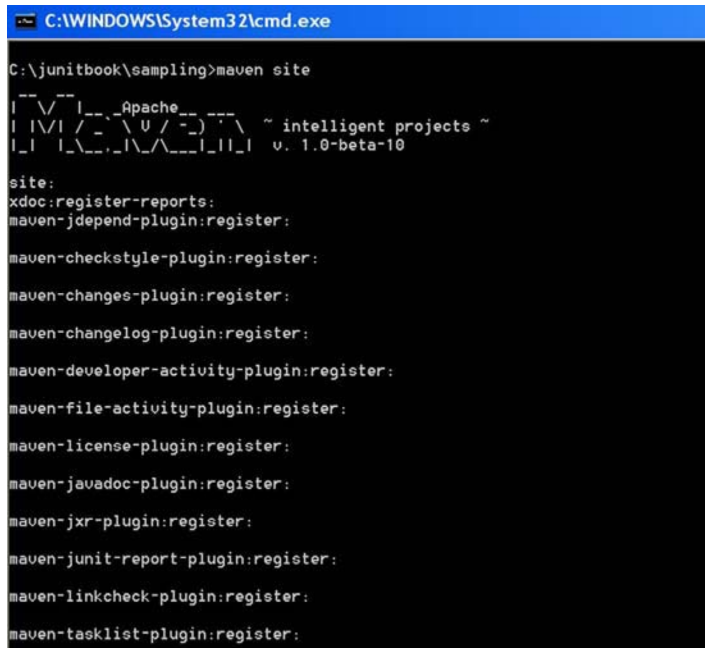


Figure 5.3 Beginning of the site-generation plugin execution showing the names of the different reports that will be generated

Figure 5.3 shows only the very beginning of the `site` plugin's execution. The website generation is quite rich. Maven generates several reports by default, as shown by figure 5.3. For example, you can see that it will generate metrics (`maven-jepend-plugin`), a Checkstyle report (`maven-checkstyle-plugin`), a change log report (`maven-changelog-report`), a JUnit report (`maven-junit-report-plugin`), a check for broken URL links in the documentation (`maven-linkcheck-plugin`), and so forth.

Figure 5.4 shows the welcome page of the generated web site. The generated web site makes good use of the information you entered in `project.xml`. For example, the images at the top are the ones you defined in listing 5.6 with the two `logo` elements. Each is linked to the URLs defined by the two corresponding `url` elements. The description comes from the `description` element, and the header is the project name (`name` element).

On the left are several menus, some of which contain submenus. For example, clicking Project Reports yields the screen in figure 5.5, showing all the generated default reports.

Figure 5.4 Welcome page of the generated site showing how the information entered in `project.xml` is used

- Project Reports
  - Metrics
  - Checkstyle
  - Change Log
  - Developer Activity
  - File Activity
  - Project License
  - JavaDocs
  - JavaDoc Report
  - Source Xref
  - Test Xref
  - Unit Tests
  - Link Check Report
  - Task List
- JavaDocs
- Source XReference
- Test XReference
- Development Process

powered by MAVEN

Overview

Document	Description
Metrics	Report on source code metrics.
Checkstyle	Report on coding style conventions.
Change Log	Report on the source control changelog.
Developer Activity	Report on the amount of developer activity.
File Activity	Report on file activity.
Project License	Displays the primary license for the project.
JavaDocs	JavaDoc API documentation.
JavaDoc Report	Report on the generation of JavaDoc.
Source Xref	A set of browsable cross-referenced sources.
Test Xref	A set of browsable cross-referenced test sources.
Unit Tests	Report on the results of the unit tests.
Link Check Report	Report on the validity of all links in the documentation.
Task List	Report on tasks specified in the source code.

Figure 5.5 List of Maven-generated default reports for the `sampling` project

It is possible to control exactly what reports you want for your web site by explicitly listing the desired reports in `project.xml`. For example, if you want only the unit test report and the checkstyle report, you can write the following at the end of `project.xml`:

```
<reports>
  <report>maven-junit-report-plugin</report>
  <report>maven-checkstyle-plugin</report>
</reports>
```

Describing build-related information

Let's complete the project object model (POM) by entering build-related information into `project.xml` (listing 5.7).

Listing 5.7 Second part of project.xml containing build-related information

```
<!--dependencies>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>                    ❶
  <version>1.2.8</version>
</dependencies-->

<build>
  <sourceDirectory>src/java</sourceDirectory>            ❷
  <unitTestSourceDirectory>src/test</unitTestSourceDirectory>   ❸
  <unitTest>
    <includes>
      <include>**/Test*.java</include>
    </includes>                                                 ❹
    <excludes>
      <exclude>**/Test*All.java</exclude>
      <exclude>**/TestDefaultController?.java</exclude>
    </excludes>
  </unitTest>
</build>

</project>
```

❶ Describe the project's external dependencies. A dependency typically specifies a jar, but a dependency can be of any type. All jar dependencies are added to the classpath and used by the different plugins, such as the `junit` plugin. In this case, you have no external dependencies, which is why the log4j dependency is commented out in listing 5.7. In section 5.3.4, we describe in detail how Maven handles dependencies with the notion of local and remote *repositories.*

❷ Describe the location of the runtime sources.

❸ This is the location of the test sources.

❹ Define the test classes you expect to include/exclude in the tests. Notice that you exclude the `TestDefaultController?.java` classes created in chapter 3 (the `?` stands for any character), because they are unfinished tutorial classes and are not meant to be executed.

❸ ❹ These code segments are used by the `junit` plugin.

Given just the description in listing 5.7, you can now run any of Maven's plugins to compile, package, and test your project, and more.

5.3.3 Executing JUnit tests with Maven

Executing JUnit tests in Maven is as simple as invoking the `junit` plugin with `maven test` from a command shell (see figure 5.6). This is close to the result of running the Ant script, back in figure 5.1—*but without writing a single line of script*! Generating a JUnit report is just as easy: Enter **`maven site`**, and the web site is generated, along with your JUnit report (among others). Figure 5.7 shows the JUnit report summary page for the `sampling project`.

5.3.4 Handling dependent jars with Maven

Maven solves another difficult issue: jar proliferation. You have probably noticed that more and more high-quality libraries are available in the Java community. Instead of reinventing the wheel, increasing numbers of projects import third-party libraries. The dark side is that building a project from its sources can be a nightmare, because you have to gather all the external jars, all in their correct versions.

Maven handles project *dependencies* (also called *artifacts*) through the use of two repositories: a remote repository and a local one. Figure 5.8 explains the workflow.

The first step for a project is to declare its dependencies in its `project.xml`. Although the `sampling` project does not depend on any external jars, let's imagine it needs to use Log4j. You would add the following to `project.xml`:

```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.8</version>
</dependency>
```

When you execute a Maven goal on a project, here's what happens (following the numbers from figure 5.8):

❶ Check dependencies. Maven parses the dependencies located in `project.xml`.

❷ Check the dependency's existence in the local repository. For each dependency, Maven checks if it can be found in the local repository. This local repository is

```
C:\WINDOWS\System32\cmd.exe

C:\junitbook\sampling>maven test
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-beta-10

java:prepare-filesystem:
    [mkdir] Created dir: C:\junitbook\sampling\target\classes

java:compile:
    [echo] Compiling to C:\junitbook\sampling/target/classes
    [javac] Compiling 6 source files to C:\junitbook\sampling\target\classes

java:jar-resources:

test:prepare-filesystem:
    [mkdir] Created dir: C:\junitbook\sampling\target\test-classes
    [mkdir] Created dir: C:\junitbook\sampling\target\test-reports

test:test-resources:

test:compile:
    [javac] Compiling 9 source files to C:\junitbook\sampling\target\test-classes

test:test:
    [junit] dir attribute ignored if running in the same VM
    [junit] Running junitbook.sampling.TestDefaultController
    [junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 0.04 sec
BUILD SUCCESSFUL
Total time:  7 seconds

C:\junitbook\sampling>_
```

Figure 5.6 Results of executing `maven test` on the `sampling` project

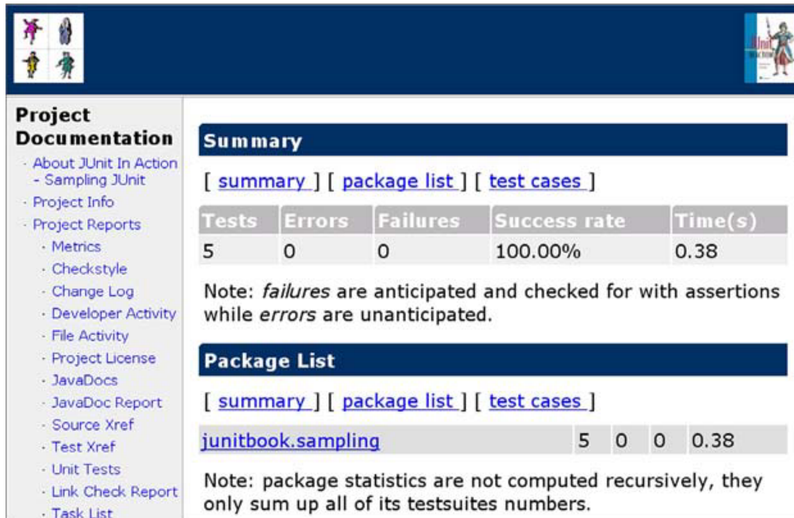


Figure 5.7 JUnit report generated by Maven

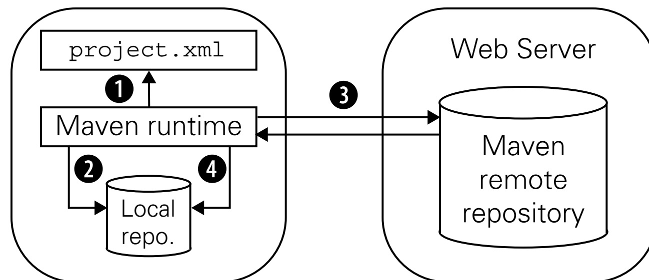


Figure 5.8 How Maven resolves external dependencies

automatically created when you execute Maven the first time (it is located in your home user directory under `.maven/repository/`).

❸ Download the dependency. If the dependency is not found in the local repository, Maven downloads it from a Maven remote repository. The default Maven remote repository is http://www.ibiblio.org/maven/. You can easily override this default by setting the `maven.repo.remote` property in a `project.properties` or `build.properties` file in the same location as your `project.xml` file. This is especially useful if you wish to set up a project-wide or company-wide Maven remote repository.

❹ Store the dependency. Once the dependency has been downloaded, Maven stores it in your local repository to prevent having to fetch it again next time.

The structure of the local and remote repositories is the same. Figure 5.9 shows a very simple repository.

In figure 5.9 you can see that jars are put in a `jars/` directory. The names are suffixed with the version to let you put several versions in the same directory and for easy identification. (For example, the `log4j` jar is available in versions 1.1.3,

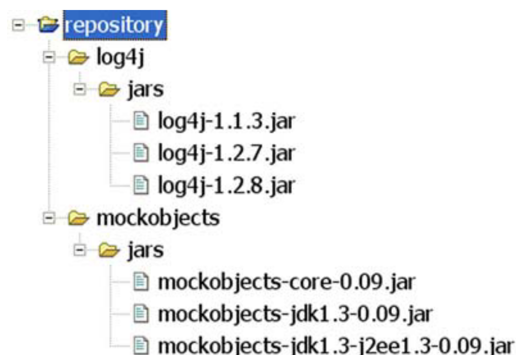


Figure 5.9 Very simple portion of a Maven repository (local or remote)

1.2.7, and 1.2.8.) In addition, you can collect several jars into a common group. Figure 5.9 also shows the MockObjects jars organized into a `mockobjects` group. Although the figure only shows jars, it is possible to put any type of dependency in a Maven repository.