Creating Applications with Maven

This chapter covers:

- Setting Up an Application Directory Structure
- Using Project Inheritance
- Managing Dependencies
- Using Snapshots
- Using Version Ranges
- Managing Plugins
- Utilizing the Build Life Cycle
- Using Profiles
- Deploying your Application
- Creating a Web Site for your Application

Walking on water and developing software from a specification are easy if both are frozen.

- Edward V. Berard

3.1. Introduction

In the second chapter you stepped though the basics of setting up a simple project. Now you will delve in a little deeper, using a real-world example. In this chapter, you are going to learn about some of Maven's best practices and advanced uses by working on a small application to manage frequently asked questions (FAQ). In doing so you will be guided through the specifics of setting up an application and managing that structure.

The application that you are going to create is called Proficio, which is Latin for "help". So, lets start by discussing the ideal directory structure for Proficio.

3.2. Setting Up an Application Directory Structure

In setting up the directory structure for Proficio, it is important to keep in mind that Maven emphasizes the practice of discrete, coherent, and modular builds. The natural outcome of this practice is discrete and coherent components, which enable code reusability, a necessary goal for every software development project. The guiding principle in determining how best to decompose your application is called the *Separation of Concerns* (SoC).

SoC refers to the ability to identify, encapsulate, and operate on the pieces of software that are relevant to a particular concept, goal, task, or purpose. Concerns are the primary motivation for organizing and decomposing software into smaller, more manageable and comprehensible parts, each of which addresses one or more concerns.

As such, you will see that the Proficio sample application is made up of several modules:

- **Proficio Model**: The data model for the Proficio application, which consists of all the classes that will be used by Proficio as a whole.
- **Proficio API**: The application programming interface for Proficio, which consists of a set of interfaces. The interfaces for the APIs of major components, like the store, are also kept here.
- **Proficio Core**: The implementation of the API.
- **Proficio Stores**: The module which itself houses all the store modules. Proficio has a very simple `memory-based` store and a simple `XStream-based` store.
- **Proficio CLI**: The code which provides a command line interface to Proficio.

These are default naming conventions that Maven uses, but you are free to name your modules in any fashion your team decides. The only real criterion to which to adhere is that your team has a single convention, and they clearly understand that convention, and can easily identify what a particular module does simply by looking at its name.

In examining the top-level POM for Proficio, you can see in the modules element all the modules that make up the Proficio application. A module is a reference to another Maven project, which really means a reference to another POM. This setup is typically referred to as a multi-module build and this is how it looks in the top-level Proficio POM:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>org.apache.maven.proficio</groupId>
 <artifactId>proficio</artifactId>
 <packaging>pom</packaging>
 <version>1.0-SNAPSHOT</version>
 <name>Maven Proficio</name>
 <url>http://maven.apache.org</url>
...
  <modules>
    <module>proficio-model</module>
    <module>proficio-api</module>
    <module>proficio-core</module>
    <module>proficio-stores</module>
    <module>proficio-cli</module>
  </modules>
...
</project>
```

An important feature to note in the POM above is the value of the version element, which you can see is `1.0-SNAPSHOT`. For an application that has multiple modules it is very common that you will release all the modules together and so it makes sense that all the modules have a common application version. It is recommended that you specify the application version in the top-level POM and use that version across all the modules that make up your application.

Currently there is some variance on the Maven web site when referring to directory structures that contain more than one Maven project. In Maven 1.x these were commonly referred to as multi-project builds and some of this vestigial terminology carried over to the Maven 2.x documentation, but the Maven team is now trying to consistently refer to these setups as multi-module builds.

You should take note of the `packaging` element, which has a value of `pom`. For POMs that contain modules, the packaging must be set to value of `pom`: this tells Maven that you're going to be walking through a set of modules in a structure similar to the example being covered here. If you were to look at Proficio's directory structure you would see the following:

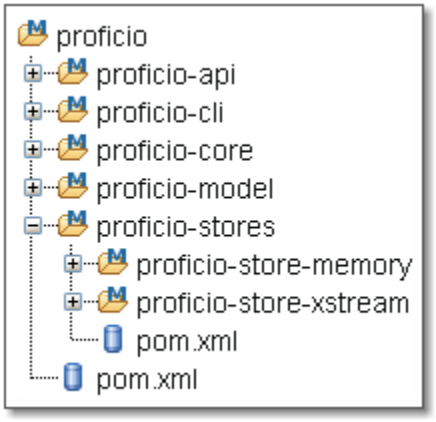

Figure 3-1: Proficio directory structure

You may have noticed that the module elements in the POM match the names of the directories that you see above in the Proficio directory structure. Looking at the module names is how Maven steps into the right directory to process the POM located there. Let's take a quick look at the modules and examine the packaging for each of them:

Table 3-1: Module packaging types

Module	Packaging
`proficio-api`	`jar`
`proficio-cli`	`jar`
`proficio-core`	`jar`
`proficio-model`	`jar`
`proficio-stores`	`pom`

The packaging type in most cases is the default of jar, but the interesting thing here is that we have another project with a packaging type of `pom`, which is the `proficio-stores` module. If you take a look at the POM for the `proficio-stores` module you will see a set of modules contained there:

```
<project>
   <parent>
    <groupId>org.apache.maven.proficio</groupId>
    <artifactId>proficio</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>proficio-stores</artifactId>
  <name>Maven Proficio Stores</name>
  <packaging>pom</packaging>
  <modules>
    <module>proficio-store-memory</module>
    <module>proficio-store-xstream</module>
  </modules>
</project>
```

Examine the directory structure inside the `proficio-stores` directory and you will see the following:

Figure 3-2: Proficio-stores directory

Whenever Maven sees a POM with a packaging of type `pom` Maven knows to look for a set of modules and to then process each of those modules. You can nest sets of projects like this to any level, organizing your projects in groups according to concern, just as has been done with Proficio's multiple storage mechanisms, which are all placed in one directory.

3.3. Using Project Inheritance

One of the most powerful features in Maven is project inheritance. Using project inheritance allows you to do things like state your organizational information, state your deployment information, or state your common dependencies - all in a single place. Being the observant user, you have probably taken a peek at all the POMs in each of the projects that make up the Proficio project and noticed the following at the top of each of the POMs:

```
...
  <parent>
   <groupId>org.apache.maven.proficio</groupId>
   <artifactId>proficio</artifactId>
   <version>1.0-SNAPSHOT</version>
  </parent>
...
```

This is the snippet in each of the POMs that lets you draw on the resources stated in the specified top-level POM and from which you can inherit down to any level you wish - enabling you to add resources where it makes sense in the hierarchy of your projects. Let's examine a case where it makes sense to put a resource in the top-level POM, using our top-level POM for the sample Proficio application.

If you look at the top-level POM for Proficio, you will see that in the dependencies section there is a declaration for JUnit version 3.8.1. In this case the assumption being made is that JUnit will be used for testing in all our child projects. So, by stating the dependency in the top-level POM once, you never have to declare this dependency again, in any of your child POMs. The dependency is stated as following:

```
<project>
...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
...
</project>
```

What specifically happens for each child POM, is that each one inherits the dependencies section of the top-level POM. So, if you take a look at the POM for the `proficio-core` module you will see the following, note specifically there is no visible dependency declaration for JUnit:

```
<project>
  <parent>
    <groupId>org.apache.maven.proficio</groupId>
    <artifactId>proficio</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>proficio-core</artifactId>
  <packaging>jar</packaging>
  <name>Maven Proficio Core</name>
  <dependencies>
    <dependency>
      <groupId>org.apache.maven.proficio</groupId>
      <artifactId>proficio-api</artifactId>
    </dependency>
    <dependency>
      <groupId>org.codehaus.plexus</groupId>
      <artifactId>plexus-container-default</artifactId>
    </dependency>
  </dependencies>
</project>
```

In order for you to see what happens during the inheritance process you will need to use the handy `mvn help:effective-pom` command. This command will show you the final result for a target POM. After you move into the `proficio-core` module and run the command, take a look at the resulting POM you will see the JUnit version 3.8.1 dependency:

```
<project>
...
  <dependencies>
  ...
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  ...
  </dependencies>
...
</project>
```

You will have noticed that the POM that you see when using the `mvn help:effective-pom` is bigger than you expected. But remember from Chapter 2 that the Super POM sits at the top of the inheritance hierarchy. So in this case, the `proficio-core` project inherits from the top-level Proficio project, which in turn inherits from the Super POM. Looking at the effective POM includes everything and is useful to view when trying to figure out what is going on when you are having problems.

3.4. Managing Dependencies

When you are building applications you have many dependencies to manage and the number of dependencies only increases over time, making dependency management difficult to say the least. Maven's strategy for dealing with this problem is to combine the power of project inheritance with specific dependency management elements in the POM.

When you write applications which consist of multiple, individual projects, it is likely that some of those projects will share common dependencies. When this happens it is critical that the same version of a given dependency is used for all your projects, so that the final application works correctly.

You don't want, for example, to end up with multiple versions of a dependency on the classpath when your application executes, as the results can be far from desirable. You want to make sure that all the versions, of all your dependencies, across all of your projects are in alignment so that your testing accurately reflects what you will deploy as your final result. In order to manage, or align, versions of dependencies across several projects, you use the dependency management section in the top-level POM of an application.

To illustrate how this mechanism works, let's look at the dependency management section of the Proficio top-level POM:

```
<project>
...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.apache.maven.proficio</groupId>
        <artifactId>proficio-model</artifactId>
        <version>${project.version}</version>
      </dependency>
      <dependency>
        <groupId>org.apache.maven.proficio</groupId>
        <artifactId>proficio-api</artifactId>
        <version>${project.version}</version>
      </dependency>
      <dependency>
        <groupId>org.apache.maven.proficio</groupId>
        <artifactId>proficio-core</artifactId>
        <version>${project.version}</version>
      </dependency>
      <dependency>
        <groupId>org.codehaus.plexus</groupId>
        <artifactId>plexus-container-default</artifactId>
        <version>1.0-alpha-9</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
...
</project>
```

Note that the `${project.version}` specification is the version specified by the top-level POM's version element, which is the application version.

As you can see within the dependency management section we have several Proficio dependencies and a dependency for the Plexus IoC container. There is an important distinction to be made between the dependencies element contained within the `dependencyManagment` element and the top-level dependencies element in the POM.

The dependencies element contained within the `dependencyManagement` element is used only to state the preference for a version and by itself does not affect a project's dependency graph, whereas the top-level dependencies element does affect the dependency graph. If you take a look at the POM for the `proficio-api` module, you will see a single dependency declaration and that it does not specify a version:

```
<project>
...
  <dependencies>
    <dependency>
      <groupId>org.apache.maven.proficio</groupId>
      <artifactId>proficio-model</artifactId>
    </dependency>
  </dependencies>
</project>
```

The version for this dependency is derived from the `dependencyManagement` element which is inherited from the Proficio top-level POM. The `dependencyManagement` declares a stated preference for the `1.0-SNAPSHOT` (stated as `${project.version}`) for `proficio-model` so that version is injected into the dependency above, to make it complete. The dependencies stated in the `dependencyManagement` only come into play when a dependency is declared without a version.

3.5. Using Snapshots

While you are developing an application with multiple modules, it is usually the case that each of the modules are in flux. Your APIs might be undergoing some change or your implementations are undergoing change and are being fleshed out, or you may be doing some refactoring. Your build system needs to be able to deal easily with this real-time flux, and this is where Maven's concept of a snapshot comes into play. A snapshot in Maven is an artifact which has been prepared using the most recent sources available. If you look at the top-level POM for Proficio you will see a snapshot version specified:

```
<project>
...
 <version>1.0-SNAPSHOT</version>
 <dependencyManagement>
    <dependencies>
     <dependency>
       <groupId>org.apache.maven.proficio</groupId>
       <artifactId>proficio-model</artifactId>
       <version>${project.version}</version>
      </dependency>
      <dependency>
       <groupId>org.apache.maven.proficio</groupId>
        <artifactId>proficio-api</artifactId>
       <version>${project.version}</version>
     </dependency>
      <dependency>
       <groupId>org.codehaus.plexus</groupId>
       <artifactId>plexus-container-default</artifactId>
       <version>1.0-alpha-9</version>
     </dependency>
   </dependencies>
 </dependencyManagement>
...
</project>
```

Specifying a snapshot version for a dependency means that Maven will look for new versions of that dependency without you having to manually specify a new version. Snapshot dependencies are assumed to be changing, so Maven will attempt to update them. By default Maven will look for snapshots on a daily basis, but you can use the `-U` command line option to force the search for updates. Controlling how snapshots work will be explained in detail in Chapter 7. When you specify a non-snapshot version of a dependency Maven will download that dependency once and never attempt to retrieve it again.

3.6. Resolving Dependency Conflicts and Using Version Ranges

With the introduction of transitive dependencies in Maven 2, it became possible to simplify your own POM by only including the dependencies you need directly, and to allow Maven to calculate the full dependency graph. However, as the graph grows, it is inevitable that two or more artifacts will require different versions of a particular dependency. In this case, Maven must choose which version to provide.

In Maven 2.0, the version selected is the one declared “nearest” to the top of the tree - that is, the least number of dependencies that were traversed to get this version. A dependency in the POM being built will be used over anything else.

However, this has limitations:

- The version chosen may not have all the features required by the other dependency.
- If multiple versions are selected at the same depth, then the result is undefined.

While further dependency management features are scheduled for the next release of Maven at the time of writing, there are ways to manually resolve these conflicts as the end user of a dependency, and more importantly ways to avoid it as the author of a reusable library.

To manually resolve conflicts, you can remove the incorrect version from the tree, or you can override both with the correct version. Removing the incorrect version requires identifying the source of the incorrect version by running Maven with the -X flag (for more information on how to do this, see section 6.9 in Chapter 6). For example, if you run `mvn -X test` on the `proficio-core` module, the output will contain something similar to:

```
proficio-core:1.0-SNAPSHOT
  junit:3.8.1 (selected for test)
  plexus-container-default:1.0-alpha-9 (selected for compile)
    plexus-utils:1.0.4 (selected for compile)
    classworlds:1.1-alpha-2 (selected for compile)
    junit:3.8.1 (not setting... to compile; local scope test wins)
  proficio-api:1.0-SNAPSHOT (selected for compile)
    proficio-model:1.0-SNAPSHOT (selected for compile)
    plexus-utils:1.1 (selected for compile)
```

Once the path to the version has been identified, you can exclude the dependency from the graph by adding an exclusion to the dependency that introduced it. In this example, `plexus-utils` occurs twice, and Proficio requires version 1.1 be used. To ensure this, modify the plexus-container-default dependency in the `proficio-core/pom.xml` file as follows:

```
...
<dependency>
 <groupId>org.codehaus.plexus</groupId>
 <artifactId>plexus-container-default</artifactId>
 <version>1.0-alpha-9</version>
 <exclusions>
  <exclusion>
   <groupId>org.codehaus.plexus</groupId>
   <artifactId>plexus-utils</artifactId>
  </exclusion>
 </exclusions>
</dependency>
...
```

This ensures that Maven ignores the 1.0.4 version of `plexus-utils` in the dependency graph, so that the 1.1 version is used instead.

The alternate way to ensure that a particular version of a dependency is used, is to include it directly in the POM, as follows:

```
...
<dependencies>
 <dependency>
  <groupId>org.codehaus.plexus</groupId>
  <artifactId>plexus-utils</artifactId>
  <version>1.1</version>
  <scope>runtime</scope>
 </dependency>
</dependencies>
...
```

However, this approach is **not** recommended unless you are producing an artifact that is bundling its dependencies and is not used as a dependency itself (for example, a WAR file). The reason for this is that it distorts the true dependency graph, which will accumulate if this project is reused as a dependency itself.

You'll notice here that the runtime scope is used. This is because in this situation, the dependency is used only for packaging, not for compilation. In fact, if it were required for compilation, for stability it would always be declared in the current POM as a dependency - regardless of whether another dependency introduces it.

Neither of these solutions is ideal, but it is possible to improve the quality of your own dependencies to reduce the risk of these issues occurring with your own build artifacts. This is extremely important if you are publishing a build, for a library or framework, that will be used widely by others. This is achieved using version ranges instead.

When a version is declared as `1.1`, as shown above for `plexus-utils`, this indicates that the preferred version of the dependency is `1.1`, but that other versions may be acceptable. Maven has no knowledge about what versions will definitely work, so in the case of a conflict with another dependency, Maven assumes that all versions are valid and uses the “nearest dependency” technique described previously to determine which version to use.

However, you may require a feature that was introduced in `plexus-utils` version 1.1. In this case, the dependency should be specified as follows:

```
<dependency>
 <groupId>org.codehaus.plexus</groupId>
 <artifactId>plexus-utils</artifactId>
 <version>[1.1,)</version>
</dependency>
```

What this means is that, while the nearest dependency technique will still be used in the case of a conflict, the version that is used must fit the range given. If the nearest version does not match, then the next nearest will be tested, and so on. Finally, if none of them match, or there were no conflicts originally, the version you are left with is `[1.1,)`. This means that the latest version, that is greater than or equal to 1.1, will be retrieved from the repository.

The notation used above is set notation, and table 3-2 shows some of the values that can be used.

Table 3-2: Examples of version ranges

Range	Meaning
`(,1.0]`	Less than or equal to 1.0
`[1.2,1.3]`	Between 1.2 and 1.3 (inclusive)
`[1.0,2.0)`	Greater than or equal to 1.0, but less than 2.0
`[1.5,)`	Greater than or equal to 1.5
`(,1.1),(1.1,)`	Any version, except 1.1

By being more specific through the use of version ranges, it is possible to make the dependency mechanism more reliable for your builds and to reduce the number of exception cases that will be required. However, you also need to avoid being too specific. If two ranges in a dependency graph do not intersect at all, the build will fail.

To understand how ranges work, it is necessary to understand how versions are compared. You can see how a version is partitioned by Maven in figure 3-3.

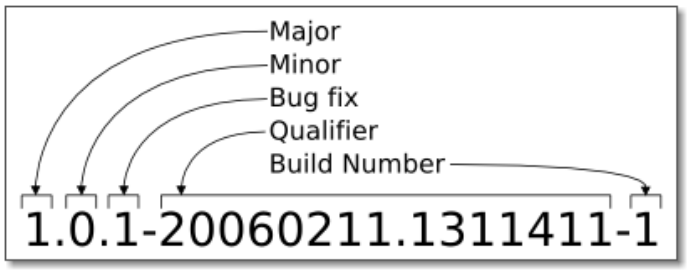

Figure 3-3: Version parsing

As you can see, a version is broken down into five parts: the major, minor and bug fix releases, then the qualifier and finally a build number. In the current version scheme, the snapshot (as shown above) is a special case where the qualifier and build number are both allowed. In a regular version, you can provide only the qualifier, or only the build number. It is intended that the qualifier indicates a version prior to release (for example, `alpha-1`, `beta-1`, `rc1`), while the build number is an increment after release to indicate patched builds.

With regard to ordering, the elements are considered in sequence to determine which is newer - first by major version, second - if the major versions were equal - by minor version, third by bug fix version, fourth by qualifier (using string comparison), and finally, by build number. A version that contains a qualifier is always considered newer than a version without a qualifier; for example, 1.2 is newer than 1.2-beta. A version that also contains a build number is considered newer than a version without a build number; for example, 1.2-beta-1 is newer than 1.2-beta. In some cases, the versions will not match this syntax. In those cases, the two versions are compared entirely as strings.

Because all of these elements are considered part of the version, the ranges do not differentiate. So, if you use the range `[1.1,)`, and the versions `1.1` and `1.2-beta-1` exist, then `1.2-beta-1` will be selected. Often this is not desired, so to avoid such a situation, you must structure your releases accordingly, through the use of a separate repository.

Whether you use snapshots until the final release, or release betas as milestones along the way, you should deploy them to a snapshot repository as is discussed in Chapter 7 of this book. This will ensure that the beta versions are used in a range only if the project has declared the snapshot (or development) repository explicitly.

A final note relates to how version updates are determined when a range is in use. This mechanism is identical to that of the snapshots that you learned in section 3.6. By default, the repository is checked once a day for updates to the versions of artifacts in use. However, this can be configured per-repository to be on a more regular interval, or forced from the command line using the `-U` option for a particular Maven execution.

If it will be configured for a particular repository, the `updatePolicy` value (which is in minutes) is changed for releases. For example:

```
...
<repository>
...
 <releases>
  <updatePolicy>interval:60</updatePolicy>
 </releases>
</repository>
```

3.7. Utilizing the Build Life Cycle

In Chapter 2 Maven was described as a framework that coordinates the execution of its plugins in a well defined way or path, which is actually Maven's default build life cycle. Maven's default build life cycle will suffice for a great number of projects without any augmentation but, of course, all projects have different requirements and it is sometimes necessary to augment the life cycle to satisfy these requirements.

For example, Proficio has a requirement to generate Java sources from a model. Maven accommodates this requirement by allowing the declaration of a plugin, which binds itself to a standard phase in Maven's default life cycle, the `generate-sources` phase.

Plugins in Maven are created with a specific task in mind, which means the plugin is bound to a specific phase in the default life cycle, typically. In Proficio, the Modello plugin is used to generate the Java sources for Proficio's data model. If you look at the POM for the `proficio-model` you will see the `plugins` element with a configuration for the Modello plugin:

```
<project>
  <parent>
    <groupId>org.apache.maven.proficio</groupId>
    <artifactId>proficio</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>proficio-model</artifactId>
  <packaging>jar</packaging>
  <name>Proficio Model</name>
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.modello</groupId>
        <artifactId>modello-maven-plugin</artifactId>
        <version>1.0-alpha-5</version>
        <executions>
          <execution>
            <goals>
              <goal>java</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <version>1.0.0</version>
          <packageWithVersion>false</packageWithVersion>
          <model>src/main/mdo/proficio.mdo</model>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

This is very similar to the declaration for the maven-compiler-plugin that you saw in Chapter 2, but here you see an additional executions element. A plugin in Maven may have several goals, so you need to specify which goal in the plugin you wish to run, by specifying the goal in the executions element.

3.8. Using Profiles

Profiles are Maven's way of letting you create environmental variations in the build life cycle to accommodate things like building on different platforms, building with different JVMs, testing with different databases, or referencing the local file system. Typically you try to encapsulate as much as possible in the POM to ensure that builds are portable, but sometimes you simply have to take into consideration variation across systems and this is why profiles were introduced in Maven.

Profiles are specified using a subset of the elements available in the POM itself (plus one extra section), and can be activated in several ways. Profiles modify the POM at build time, and are meant to be used in complementary sets to give equivalent-but-different parameters for a set of target environments (providing, for example, the path of the application server root in the development, testing, and production environments).

As such, profiles can easily lead to differing build results from different members of your team. However, used properly, you can still preserve build portability with profiles. You can define profiles in one of the following three places:

- The Maven settings file (typically `<your -home-directory>/.m2/settings.xml`)
- A file in the the same directory as the POM, called `profiles.xml`
- The POM itself

In any of the above three profile sites, you can define the following elements:

- repositories
- pluginRepositories
- dependencies
- plugins
- properties (not actually available in the main POM, but used behind the scenes)
- modules
- reporting
- dependencyManagement
- distributionManagement

A subset of the build element, which consists of:

- defaultGoal
- resources
- testResources
- finalName

There are several ways that you can activate profiles:

- Profiles can be specified explicitly using the `-P` CLI option. This option takes an argument that contains a comma-delimited list of profile-ids. When this option is specified, no profiles other than those specified in the option argument will be activated. For example:

```
mvn -Pprofile1,profile2 install
```

- Profiles can be activated in the Maven settings, via the `activeProfiles` section. This section takes a list of `activeProfile` elements, each containing a profile-id. Note that you must have defined the profiles in your `settings.xml` file as well. For example:

```
<settings>
…
  <profiles>
    <profile>
      <id>profile1</id>
      …
    </profile>
  </profiles>
  <activeProfiles>
        <activeProfile>profile1</activeProfile>
  </activeProfiles>
...
</settings>
```

- Profiles can be triggered automatically based on the detected state of the build environment. These activators are specified via an **activation** section in the profile itself. Currently, this detection is limited to prefix-matching of the JDK version, the presence of a system property, or the value of a system property. Here are some examples:

```
<profile>
  <id>profile1</id>
...
   <activation>
     <jdk>1.4</jdk>
   </activation>
</profile>
```

This activator will trigger the profile when the JDK's version starts with "1.4" (eg. "1.4.0_08", "1.4.2_07", "1.4").

```
<profile>
  <id>profile1</id>
...
  <activation>
   <property>
      <name>debug</name>
   </property>
  </activation>
</profile>
```

This will activate the profile when the system property "debug" is specified with any value.

```
<profile>
  <id>profile1</id>
 ...
  <activation>
    <property>
      <name>environment</name>
      <value>test</value>
    </property>
  </activation>
</profile>
```

This last example will activate the profile when the system property "environment" is specified with the value "test".

Now that you are familiar with profiles, you are going to use them to create tailored assemblies: an assembly of Proficio which uses the memory-based store and an assembly of Proficio which uses the XStream-based store. The assemblies will be created in the `proficio-cli` module and the profiles used to control the creation of our tailored assemblies are defined there as well.

If you take a look at the POM for the `proficio-cli` module you will see the profile definitions:

```
<project>
...
 <!-- Profiles for the two assemblies to create for deployment -->
 <profiles>
  <!-- Profile which creates an assembly using the memory based store -->
   <profile>
    <id>memory</id>
     <build>
      <plugins>
        <plugin>
         <artifactId>maven-assembly-plugin</artifactId>
          <configuration>
           <descriptors>
            <descriptor>src/main/assembly/assembly-store-memory.xml</descriptor>
           </descriptors>
          </configuration>
        </plugin>
      </plugins>
     </build>
    <activation>
      <property>
       <name>memory</name>
        </property>
     </activation>
   </profile>
    <!-- Profile which creates an assembly using the xstream based store -->
   <profile>
    <id>xstream</id>
     <build>
       <plugins>
         <plugin>
          <artifactId>maven-assembly-plugin</artifactId>
          <configuration>
           <descriptors>
            <descriptor>src/main/assembly/assembly-store-xstream.xml</descriptor>
              </descriptors>
            </configuration>
          </plugin>
        </plugins>
      </build>
      <activation>
        <property>
          <name>xstream</name>
        </property>
      </activation>
    </profile>
  </profiles>
</project>
```

You can see there are two profiles: one with an id of `memory` and another with an id of `xstream`. In each of these profiles you are configuring the assembly plugin to point at the assembly descriptor that will create a tailored assembly. You will also notice that the profiles are activated using a system property. It should be noted that the examples below depend on other parts of the build having been executed beforehand so it might be useful to run `mvn install` at the top level of the project to ensure that needed components are installed into the local repository.

If you wanted to create the assembly using the memory-based store you would execute the following:

```
mvn -Dmemory clean assembly:assembly
```

If you wanted to create the assembly using the XStream-based store you would execute the following:

```
mvn -Dxstream clean assembly:assembly
```

Both of the assemblies are created in the target directory and if you use the `jar tvf` command on the resulting assemblies you will see that the memory-based assembly only contains the `proficio-store-memory-1.0-SNAPSHOT.jar` file and the XStream-based store only contains the `proficio-store-xstream-1.0-SNAPSHOT.jar` file. This is a very simple example but illustrates how you can customize the execution of the life cycle using profiles to suit any requirement you might have.

3.9. Deploying your Application

Now that you have an application assembly, you'll want to share it with as many people as possible! So, it is now time to deploy your application assembly.

Currently Maven supports several methods of deployment, including simple file-based deployment, SSH2 deployment, SFTP deployment, FTP deployment, and external SSH deployment. In order to deploy you need to correctly configure your `distributionManagement` element in your POM, which would typically be your top-level POM so that all child POMs can inherit this information. Here are some examples of how to configure your POM via the various deployment mechanisms.

3.9.1. Deploying to the File System

To deploy to the file system you would use something like the following:

```
<project>
...
  <distributionManagement>
    <repository>
      <id>proficio-repository</id>
      <name>Proficio Repository</name>
      <url>file://${basedir}/target/deploy</url>
    </repository>
  </distributionManagement>
...
</project>
```

3.9.2. Deploying with SSH2

To deploy to an SSH2 server you would use something like the following:

```
<project>
...
  <distributionManagement>
    <repository>
      <id>proficio-repository</id>
      <name>Proficio Repository</name>
      <url>scp://sshserver.yourcompany.com/deploy</url>
    </repository>
  </distributionManagement>
...
</project>
```

3.9.3. Deploying with SFTP

To deploy to an SFTP server you would use something like the following:

```
<project>
...
  <distributionManagement>
    <repository>
      <id>proficio-repository</id>
      <name>Proficio Repository</name>
      <url>sftp://ftpserver.yourcompany.com/deploy</url>
    </repository>
  </distributionManagement>
...
</project>
```

3.9.4. Deploying with an External SSH

Now, the first three methods illustrated are included with Maven, so only the `distributionManagement` element is required, but to use an external SSH command to deploy you must configure not only the `distributionManagement` element, but also a build extension.

```
<project>
...
  <distributionManagement>
    <repository>
      <id>proficio-repository</id>
      <name>Proficio Repository</name>
      <url>scpexe://sshserver.yourcompany.com/deploy</url>
    </repository>
  </distributionManagement>

  <build>
    <extensions>
      <extension>
        <groupId>org.apache.maven.wagon</groupId>
         <artifactId>wagon-ssh-external</artifactId>
         <version>1.0-alpha-6</version>
      </extension>
    </extensions>
  </build>
...
</project>
```

The build extension specifies the use of the Wagon external SSH provider, which does the work of moving your files to the remote server. Wagon is the general purpose transport mechanism used throughout Maven.

3.9.5. Deploying with FTP

To deploy with FTP you must also specify a build extension. To deploy to an FTP server you would use something like the following:

```
<project>
...
  <distributionManagement>
    <repository>
      <id>proficio-repository</id>
      <name>Proficio Repository</name>
      <url>ftp://ftpserver.yourcompany.com/deploy</url>
    </repository>
  </distributionManagement>
  <build>
    <extensions>
      <extension>
        <groupId>org.apache.maven.wagon</groupId>
         <artifactId>wagon-ftp</artifactId>
         <version>1.0-alpha-6</version>
      </extension>
    </extensions>
  </build>
...
</project>
```

Once you have configured your POM accordingly, and you are ready to initiate deployment, simply execute the following command:

```
mvn deploy
```

3.10. Creating a Web Site for your Application

Now that you have walked though the process of building, testing and deploying Proficio, it is time for you to see how a standard web site is created for an application. For applications like Proficio it is recommended that you create a source directory at the top-level of the directory structure to store the resources that are used to generate the web site. If you take a look you will see that we have something similarly the following:

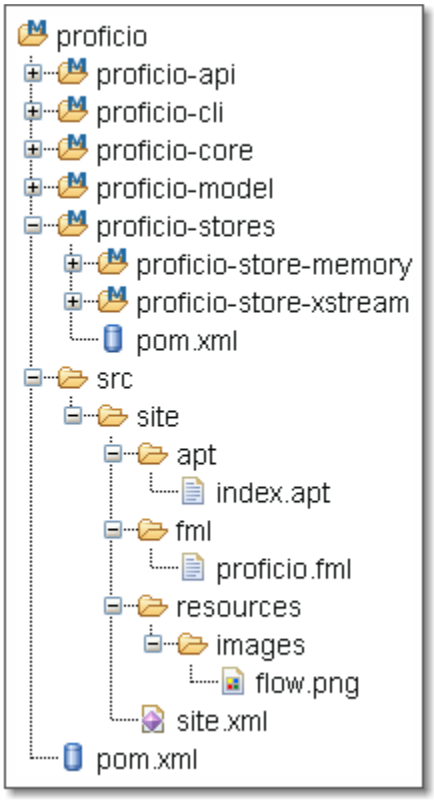

Figure 3-4: The site directory structure

Everything that you need to generate the site resides within the `src/site` directory. Within the `src/site` directory, there is a subdirectory for each of the supported documentation formats that you are using for your site and the very important site descriptor. Maven supports a number of different documentation formats to accommodate various needs and preferences.

Currently the most well supported formats available are:

- The XDOC format, which is a simple XML format used widely at Apache.
- The APT format (Almost Plain Text), which is a wiki-like format that allows you to write simple, structured documents (like this) very quickly. A full reference of the APT Format is available.
- The FML format, which is the FAQ format. A simple XML format for managing FAQs.
- The DocBook Simple format, which is a less complex version of the full DocBook format.

Maven also has limited support for:

- The Twiki format, which is a popular Wiki markup format.
- The Confluence format, which is another popular Wiki markup format.
- The DocBook format.

We will look at a few of the more well supported formats further on in the chapter, but you should become familiar with the site descriptor as it is used to:

- Configure the appearance of the banner
- Configure the skin used for the site
- Configure the format of the publish date
- Configure the links displayed below the banner
- Configure additional information to be fed into the <head/> element of the generated pages
- Configure the menu items displayed in the navigation column
- Configure the appearance of project reports

If you look in the `src/site` directory of the Proficio application and look at the site descriptor you will see the following:

```
<project name="Proficio">

  <bannerLeft>
    <name>Proficio</name>
    <href>http://maven.apache.org/</href>
  </bannerLeft>

  <bannerRight>
    <name>Proficio</name>
    <src>http://maven.apache.org/images/apache-maven project.png</src>
  </bannerRight>

  <skin>
    <groupId>org.apache.maven.skins</groupId>
    <artifactId>maven-default-skin</artifactId>
    <version>1.0-SNAPSHOT</version>
  </skin>

  <publishDate format="dd MMM yyyy" />
  <body>
    <links>
      <item name="Apache" href="http://www.apache.org/"/>
      <item name="Maven" href="http://maven.apache.org/"/>
      <item name="Continuum" href="http://maven.apache.org/continuum"/>
    </links>
    <head>
      <meta name="faq" content="proficio"/>
    </head>

    <menu name="Quick Links">
      <item name="Features" href="/maven-features.html"/>
    </menu>

    <menu name="About Proficio">
      <item name="What is Proficio?" href="/what-is-maven.html"/>
    </menu>

    ${reports}
  </body>
</project>
```

This is a fairly standard site descriptor, but you should know about the specifics of each of the elements in the site descriptor:

Table 3-3: Site descriptor

Site Descriptor Element	Description
`bannerLeft and bannerRight`	These elements take a name, `href` and optional `src` element, which can be used for images.
`skin`	This element looks very much like a dependency (the same mechanism is used to retrieve the skin) and controls which skin is used for the site.
`publishDate`	The format of the publish date is that of the `SimpleDateFormat` class in Java.
`body/links`	The link elements control the references that are displayed below the banner and take a simple name and `href`.
`body/head`	The head element allows you to insert anything in the head element of generated pages. You may wish to add metadata, or script snippets for activating tools like Google Analytics.
`body/menu`	The menu elements control the list of menus and their respective menu items that are displayed in the navigation column of the site. You can have any number of menu items each containing any number of links.
`body/${reports}`	The inclusion of the `${reports}` reference controls whether the project reports are displayed in the web site. You might exclude ${reports} reference for a site that consists purely of user documentation for example.

One of the most popular features of Maven are the standard reports that can be produced with little effort. If you simply include the `${reports}` reference in your site descriptor you will, by default, have the standard project information reports generated and displayed automatically for you. The standard project information reports consist of the following:

- Dependencies Report
- Mailing Lists Report
- *Continuous* Integration Report
- Source Repository Report
- Issue Tracking Report
- Project Team Report
- License

Even though the standard reports are very useful, often you will want to customize the projects reports that are created and displayed in your web site. The reports created and displayed are controlled in the build/reports element in the POM. You may want to be more selective about the reports that you generate and to do so you need to list each report that you want to include as part of the site generation. You do so by configuring the plugin as follows:

```
<project>
...
  <reporting>
...
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-project-info-reports-plugin</artifactId>
      <reportSets>
        <reportSet>
          <reports>
            <report>dependencies</report>
            <report>project-team</report>
            <report>mailing-list</report>
            <report>cim</report>
            <!--
            Issue tracking report will be omitted
            <report>issue-tracking</report>
            -->
            <report>license</report>
            <report>scm</report>
          </reports>
        </reportSet>
      </reportSets>
    </plugin>
  </plugins>
...
  </reporting>
...
</project>
```

Now that you have a good grasp on what formats are supported, how the site descriptor works, and how to configure reports it's time for you to generate the site. You can do so by executing the following command:

```
mvn site
```

After executing this command you will end up with a directory structure (generated inside the target directory) with the generated content that looks like this:

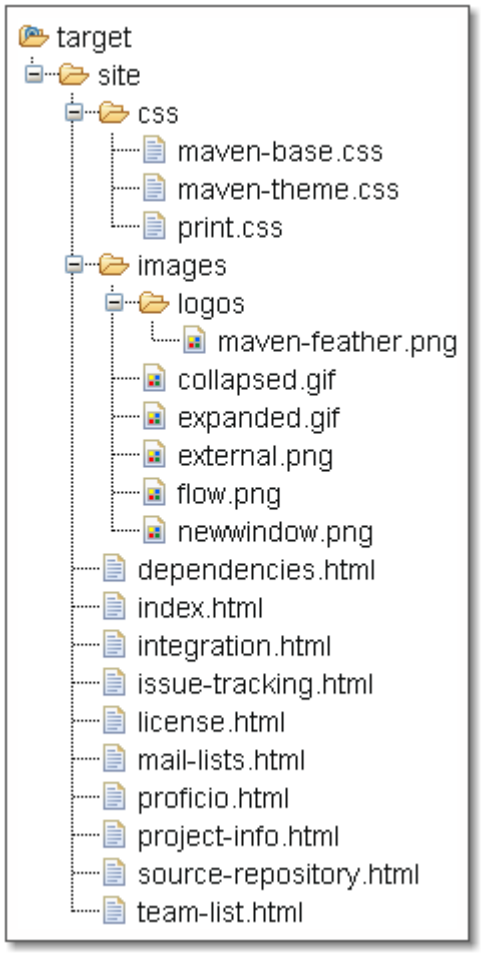

Figure 3-5: The `target` directory

If you look at the generated site with your browser this is what you will see:

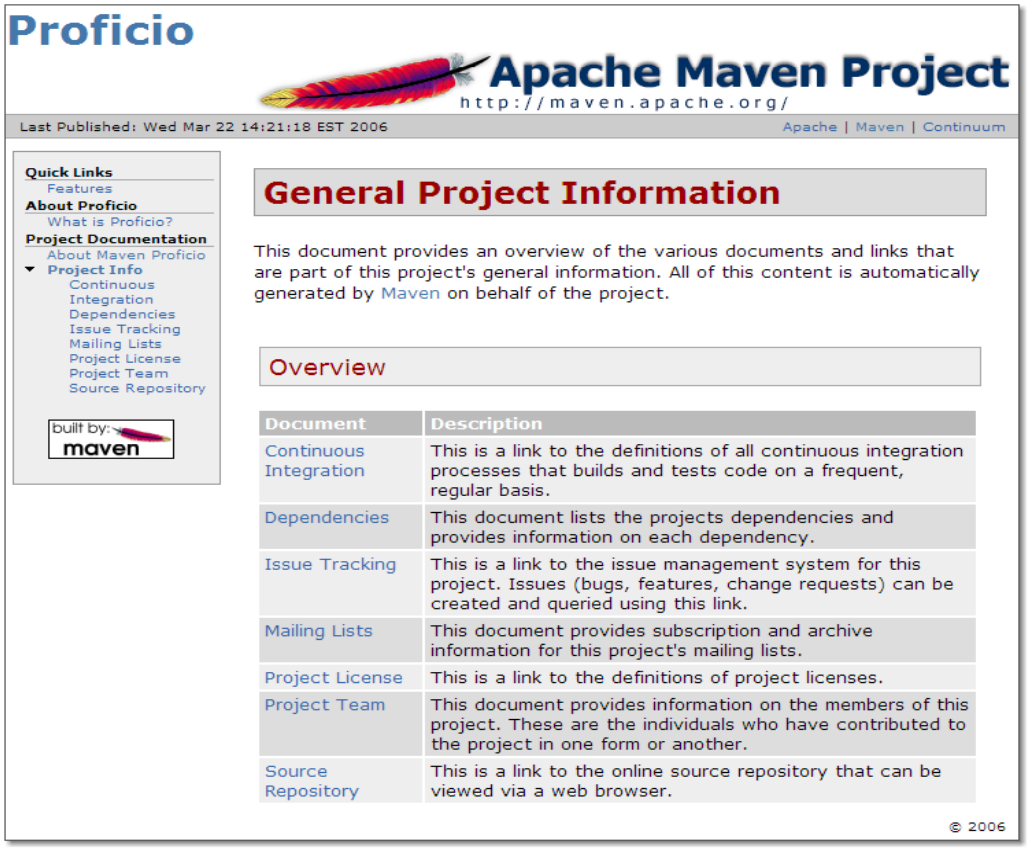

Figure 3-6: The sample generated site

If you have resources like images or PDFs that you want to be able to reference in your documents you can use the `src/site/resources` to store them and when the site is generated the content of `src/site/resources` will be copied to the top-level directory of the site.

If you remember the directory structure for the the sources of our site you will have noticed the `src/site/resources` directory and that it contains an images directory and as you can see in the directory listing above it is located within the images directory of the generated site. Keeping in mind this simple rule you can add any resources you wish to your site.

3.11. Summary

In this chapter you have learned to setup a directory structure for a typical application and learned the basics of managing the application's development with Maven. You should now have a grasp of how project inheritance works, how to manage your application's dependencies, how to make small modifications to Maven's build life cycle, how to deploy your application, and how to create a simple Web site for your application. You are now prepared to move on and learn about more advanced application directory structures like the J2EE example you will see in Chapter 4 and more advanced uses of Maven like creating your own plugins, augmenting your site to view quality metrics, and using Maven in a collaborative environment.