



# Team Collaboration with Maven

This chapter covers:

- How Maven helps teams
- How to set up consistent developer environments
- How to set up a shared internal Maven repository
- Continuous Integration
- Creating shared organization metadata and archetypes
- Releasing a project

Collaboration on a book is the ultimate unnatural act.

- Tom Clancy

## 7.1. The Issues Facing Teams

Software development as part of a team, whether it is 2 people or 200 people, faces a number of challenges to the success of the effort. Many of these challenges are out of any given technology's control – for instance finding the right people for the team, and dealing with differences in opinions.

However, one of the biggest challenges relates to the sharing and management of development information. While it's essential that team members receive all of the project information required to be productive, it's just as important that they don't waste valuable time researching and reading through too many information sources simply to find what they need.

This problem gets exponentially larger as the size of the team increases. As each member retains project information that isn't shared or commonly accessible, every other member (and particularly new members), will inevitably have to spend time obtaining this localized information, repeating errors previously solved or duplicating efforts already made. Even when it is not localized, project information can still be misplaced, misinterpreted, or forgotten, further contributing to the problem.

As teams continue to grow, it is obvious that trying to publish and disseminate all of the available information about a project would create a near impossible learning curve and generate a barrier to productivity.

This problem is particularly relevant to those working as part of a team that is distributed across different physical locations and timezones. However, although a distributed team has a higher communication overhead than a team working in a single location, the key to the information issue in both situations is to reduce the amount of communication necessary to obtain the required information in the first place.

A *Community-oriented Real-time Engineering* (CoRE) process excels with this information challenge. CoRE is based on accumulated learnings from open source projects that have achieved successful, rapid development, working on complex, component-based projects despite large, widely-distributed teams. Using the model of a community, CoRE emphasizes the relationship between project information and project members.

An organizational and technology-based framework, CoRE enables globally distributed development teams to cohesively contribute to high-quality software, in rapid, iterative cycles. This value is delivered to development teams by supporting project transparency, real-time stakeholder participation, and asynchronous engineering, which is enabled by the accessibility of consistently structured and organized information such as centralized code repositories, web-based communication channels and web-based project management tools.

Even though teams may be widely distributed, the fact that everyone has direct access to the other team members through the CoRE framework reduces the time required to not only share information, but also to incorporate feedback, resulting in shortened development cycles. The CoRE approach to development also means that new team members are able to become productive quickly, and that existing team members become more productive and effective.

While Maven is not tied directly to the CoRE framework, it does encompass a set of practices and tools that enable effective team communication and collaboration. These tools aid the team to organize, visualize, and document for reuse the artifacts that result from a software project.

As described in Chapter 6, Maven can gather and share the knowledge about the health of a project. In this chapter, this is taken a step further, demonstrating how Maven provides teams with real-time information on the builds and health of a project, through the practice of continuous integration.

This chapter also looks at the adoption and use of a consistent development environment, and the use of archetypes to ensure consistency in the creation of new projects.

## 7.2. How to Set up a Consistent Developer Environment

Consistency is important when establishing a shared development environment. Without it, the set up process for a new developer can be slow, error-prone and full of omissions. Additionally, because the environment will tend to evolve inconsistently once started that way, it will be the source of time-consuming development problems in the future.

While one of Maven's objectives is to provide suitable conventions to reduce the introduction of inconsistencies in the build environment, there are unavoidable variables that remain, such as different installation locations for software, multiple JDK versions, varying operating systems, and other discrete settings such as user names and passwords.

To maintain build consistency, while still allowing for this natural variability, the key is to minimize the configuration required by each individual developer, and to effectively define and declare them. In Maven, these variables relate to the user and installation settings files, and to user-specific profiles.

In Chapter 2, you learned how to create your own `settings.xml` file. This file can be stored in the `conf` directory of your Maven installation, or in the `.m2` subdirectory of your home directory (settings in this location take precedence over those in the Maven installation directory). The `settings.xml` file contains a number of settings that are user-specific, but also several that are typically common across users in a shared environment, such as proxy settings.

In a shared development environment, it's a good idea to leverage Maven's two different settings files to separately manage shared and user-specific settings. Common configuration settings are included in the installation directory, while an individual developer's settings are stored in their home directory.

The following is an example configuration file that you might use in the installation directory, `<maven home>/conf/settings.xml`:

```
<settings>
  <proxies>
    <proxy>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxy</host>
      <port>8080</port>
    </proxy>
  </proxies>
  <servers>
    <server>
      <id>website</id>
      <username>${website.username}</username>
      <filePermissions>664</filePermissions>
      <directoryPermissions>775</directoryPermissions>
    </server>
  </servers>
  <profiles>
    <profile>
      <id>default-repositories</id>
      <repositories>
        <repository>
          <id>internal</id>
          <name>Internal Repository</name>
          <url>http://repo.mycompany.com/internal/</url>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>internal</id>
          <name>Internal Plugin Repository</name>
          <url>http://repo.mycompany.com/internal/</url>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
  <activeProfiles>
    <activeProfile>property-overrides</activeProfile>
    <activeProfile>default-repositories</activeProfile>
  </activeProfiles>
  <pluginGroups>
    <pluginGroup>com.mycompany.plugins</pluginGroup>
  </pluginGroups>
</settings>
```

There are a number of reasons to include these settings in a shared configuration:

- If a proxy server is allowed, it would usually be set consistently across the organization or department.
- The server settings will typically be common among a set of developers, with only specific properties such as the user name defined in the user's settings. By placing the common configuration in the shared settings, issues with inconsistently-defined identifiers and permissions are avoided.
- The mirror element can be used to specify a mirror of a repository that is closer to you, which is typically one that has been set up within your own organization or department. See section 7.3 of this chapter for more information on creating a mirror of the central repository within your own organization.
- The profile defines those common, internal repositories that contain a given organization's or department's released artifacts. These repositories are independent of the central repository in this configuration. See section 7.3 for more information on setting up an internal repository.
- The active profiles listed enable the profile defined previously in every environment. Another profile, `property-overrides` is also enabled by default. This profile will be defined in the user's settings file to set the properties used in the shared file, such as `${website.username}`.
- The plugin groups are necessary only if an organization has plugins, which are run from the command line and not defined in the POM.

You'll notice that the local repository is omitted in the prior example. While you may define a standard location that differs from Maven's default (for example, `${user.home}/maven-repo`), it is important that you do not configure this setting in a way that shares a local repository, at a single physical location, across users. In Maven, the local repository is defined as the repository of a single user.

The previous example forms a basic template that is a good starting point for the settings file in the Maven installation. Using the basic template, you can easily add and consistently roll out any new server and repository settings, without having to worry about integrating local changes made by individual developers. The user-specific configuration is also much simpler as shown below:

```
<settings>
  <profiles>
    <profile>
      <id>property-overrides</id>
      <properties>
        <website.username>myuser</website.username>
      </properties>
    </profile>
  </profiles>
</settings>
```

To confirm that the settings are installed correctly, you can view the merged result by using the following help plugin command:

```
C:\mvnbook> mvn help:effective-settings
```

Separating the shared settings from the user-specific settings is helpful, but it is also important to ensure that the shared settings are easily and reliably installed with Maven, and when possible, easily updated. The following are a few methods to achieve this:

- Rebuild the Maven release distribution to include the shared configuration file and distribute it internally. A new release will be required each time the configuration is changed.
- Place the Maven installation on a read-only shared or network drive from which each developer runs the application. If this infrastructure is available, each execution will immediately be up-to-date. However, doing so will prevent Maven from being available off-line, or if there are network problems.
- Check the Maven installation into CVS, Subversion, or other source control management (SCM) system. Each developer can check out the installation into their own machines and run it from there. Retrieving an update from an SCM will easily update the configuration and/or installation, but requires a manual procedure.
- Use an existing desktop management solution, or other custom solution.

If necessary, it is possible to maintain multiple Maven installations, by one of the following methods:

- Using the `M2_HOME` environment variable to force the use of a particular installation.
- Adjusting the path or creating symbolic links (or shortcuts) to the desired Maven executable, if `M2_HOME` is not set.

Configuring the `settings.xml` file covers the majority of use cases for individual developer customization, however it applies to all projects that are built in the developer's environment. In some circumstances however, an individual will need to customize the build of an individual project. To do this, developers must use profiles in the `profiles.xml` file, located in the project directory. For more information on profiles, see Chapter 3.

Now that each individual developer on the team has a consistent set up that can be customized as needed, the next step is to establish a repository to and from which artifacts can be published and dependencies downloaded, so that multiple developers and teams can collaborate effectively.

## 7.3. Creating a Shared Repository

Most organizations will need to set up one or more shared repositories, since not everyone can deploy to the central Maven repository. To publish releases for use across different environments within their network, organization's will typically want to set up what is referred to as an *internal repository*. This internal repository is still treated as a *remote repository* in Maven, just as any other external repository would be. For an explanation of the different types of repositories, see Chapter 2.

Setting up an internal repository is simple. While any of the available transport protocols can be used, the most popular is HTTP. You can use an existing HTTP server for this, or create a new server using Apache HTTPd, Apache Tomcat, Jetty, or any number of other servers.

To set up your organization's internal repository using Jetty, create a new directory in which to store the files. While it can be stored anywhere you have permissions, in this example

`C:\mvnbook\repository` will be used. To set up Jetty, download the Jetty 5.1.10 server bundle from the book's Web site and copy it to the repository directory. Change to that directory, and run:

```
C:\mvnbook\repository> java -jar jetty-5.1.10-bundle.jar 8081
```

You can now navigate to `http://localhost:8081/` and find that there is a web server running displaying that directory. Your repository is now set up.

The server is set up on your own workstation for simplicity in this example. However, you will want to set up or use an existing HTTP server that is in a shared, accessible location, configured securely and monitored to ensure it remains running at all times.

This chapter will assume the repositories are running from `http://localhost:8081/` and that artifacts are deployed to the repositories using the file system. However, it is possible to use a repository on another server with any combination of supported protocols including `http`, `ftp`, `scp`, `sftp` and more. For more information, refer to Chapter 3.

Later in this chapter you will learn that there are good reasons to run multiple, separate repositories, but rather than set up multiple web servers, you can store the repositories on this single server. For the first repository, create a subdirectory called `internal` that will be available at `http://localhost:8081/internal/`.

This creates an empty repository, and is all that is needed to get started.

```
C:\mvnbook\repository> mkdir internal
```

It is also possible to set up another repository (or use the same one) to mirror content from the Maven central repository. While this isn't required, it is common in many organizations as it eliminates the requirement for Internet access or proxy configuration. In addition, it provides faster performance (as most downloads to individual developers come from within their own network), and gives full control over the set of artifacts with which your software is built, by avoiding any reliance on Maven's relatively open central repository.

You can create a separate repository under the same server, using the following command:

```
C:\mvnbook\repository> mkdir central
```

This repository will be available at `http://localhost:8081/central/`.

To populate the repository you just created, there are a number of methods available:

- Manually add content as desired using `mvn deploy:deploy-file`
- Set up the Maven Repository Manager as a proxy to the central repository. This will download anything that is not already present, and keep a copy in your internal repository for others on your team to reuse.
- Use `rsync` to take a copy of the central repository and regularly update it. At the time of writing, the size of the Maven repository was 5.8G.

The *Maven Repository Manager* (MRM) is a new addition to the Maven build platform that is designed to administer your internal repository. It is deployed to your Jetty server (or any other servlet container) and provides remote repository proxies, as well as friendly repository browsing, searching, and reporting. The repository manager can be downloaded from <http://maven.apache.org/repository-manager/>.

When using this repository for your projects, there are two choices: use it as a mirror, or have it override the central repository. You would use it as a mirror if it is intended to be a copy of the central repository exclusively, and if it's acceptable to have developers configure this in their settings as demonstrated in section 7.2. Developers may choose to use a different mirror, or the original central repository directly without consequence to the outcome of the build.

On the other hand, if you want to prevent access to the central repository for greater control, to configure the repository from the project level instead of in each user's settings (with one exception that will be discussed next), or to include your own artifacts in the same repository, you should override the central repository.

To override the central repository with your internal repository, you must define a repository in a settings file and/or POM that uses the identifier central. Usually, this must be defined as both a regular repository and a plugin repository to ensure all access is consistent. For example:

```
<repositories>
  <repository>
    <id>central</id>
    <name>Internal Mirror of Central Repository</name>
    <url>http://localhost:8081/central/</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>central</id>
    <name>Internal Mirror of Central Plugins Repository</name>
    <url>http://localhost:8081/central/</url>
  </pluginRepository>
</pluginRepositories>
```



You should not enter this into your project now, unless you have mirrored the central repository using one of the techniques discussed previously, otherwise Maven will fail to download any dependencies that are not in your local repository.

Repositories such as the one above are configured in the POM usually, so that a project can add repositories itself for dependencies located out of those repositories configured initially. However, there is a problem – when a POM inherits from another POM that is not in the central repository, it must retrieve the parent from the repository. This makes it impossible to define the repository in the parent, and as a result, it would need to be declared in every POM. Not only is this very inconvenient, it would be a nightmare to change should the repository location change!

The solution is to declare your internal repository (or central replacement) in the shared `settings.xml` file, as shown in section 7.2. If you have multiple repositories, it is necessary to declare only those that contain an inherited POM.

It is still important to declare the repositories that will be used in the top-most POM itself, for a situation where a developer might not have configured their settings and instead manually installed the POM, or had it in their source code check out.

The next section discusses how to set up an “organization POM”, or hierarchy, that declares shared settings within an organization and its departments.



## 7.4. Creating an Organization POM

As previously mentioned in this chapter, consistency is important when setting up your build infrastructure. By declaring shared elements in a common parent POM, project inheritance can be used to assist in ensuring project consistency.

While project inheritance was limited by the extent of a developer's checkout in Maven 1.0 – that is, the current project – Maven 2 now retrieves parent projects from the repository, so it's possible to have one or more parents that define elements common to several projects. These parents (levels) may be used to define departments, or the organization as a whole.

As an example, consider the Maven project itself. It is a part of the Apache Software Foundation, and is a project that, itself, has a number of sub-projects (Maven, Maven SCM, Maven Continuum, etc.). As a result, there are three levels to consider when working with any individual module that makes up the Maven project. This project structure can be related to a company structure, wherein there's the organization, its departments, and then the teams within those departments. Any number of levels (parents) can be used, depending on the information that needs to be shared.

To continue the Maven example, consider the POM for Maven SCM:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-parent</artifactId>
    <version>1</version>
  </parent>
  <groupId>org.apache.maven.scm</groupId>
  <artifactId>maven-scm</artifactId>
  <url>http://maven.apache.org/maven-scm/</url>
  ...
  <modules>
    <module>maven-scm-api</module>
    <module>maven-scm-providers</module>
    ...
  </modules>
</project>
```

If you were to review the entire POM, you'd find that there is very little deployment or repository-related information, as this is consistent information, which is shared across all Maven projects through inheritance.

You may have noticed the unusual version declaration for the parent project. Since the version of the POM usually bears no resemblance to the software, the easiest way to version a POM is through sequential numbering. Future versions of Maven plan to automate the numbering of these types of parent projects to make this easier.



It is important to recall, from section 7.3, that if your inherited projects reside in an internal repository, then that repository will need to be added to the `settings.xml` file in the shared installation (or in each developer's home directory).

If you look at the Maven project's parent POM, you'd see it looks like the following:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.apache</groupId>
    <artifactId>apache</artifactId>
    <version>1</version>
  </parent>
  <groupId>org.apache.maven</groupId>
  <artifactId>maven-parent</artifactId>
  <version>5</version>
  <url>http://maven.apache.org/</url>
  ...
  <mailingLists>
    <mailingList>
      <name>Maven Announcements List</name>
      <post>announce@maven.apache.org</post>
    ...
  </mailingList>
</mailingLists>

  <developers>
    <developer>
      ...
    </developer>
  </developers>
</project>
```

The Maven parent POM includes shared elements, such as the announcements mailing list and the list of developers that work across the whole project. Again, most of the elements are inherited from the organization-wide parent project, in this case the Apache Software Foundation:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.apache</groupId>
  <artifactId>apache</artifactId>
  <version>1</version>
  <organization>
    <name>Apache Software Foundation</name>
    <url>http://www.apache.org</url>
  </organization>
  <url>http://www.apache.org</url>
  ...
  <repositories>
    <repository>
      <id>apache.snapshots</id>
      <name>Apache Snapshot Repository</name>
      <url>http://svn.apache.org/maven-snapshot-repository</url>
      <releases>
        <enabled>false</enabled>
      </releases>
    </repository>
  </repositories>
  ...
  <distributionManagement>
    <repository>
      ...
    </repository>
    <snapshotRepository>
      ...
    </snapshotRepository>
  </distributionManagement>
</project>
```

The Maven project declares the elements that are common to all of its sub-projects – the snapshot repository (which will be discussed further in section 7.6), and the deployment locations.

An issue that can arise, when working with this type of hierarchy, is regarding the storage location of the source POM files. Source control management systems like CVS and SVN (with the traditional intervening trunk directory at the individual project level) do not make it easy to store and check out such a structure.

These parent POM files are likely to be updated on a different, and less frequent schedule than the projects themselves. For this reason, it is best to store the parent POM files in a separate area of the source control tree, where they can be checked out, modified, and deployed with their new version as appropriate. In fact, there is no best practice requirement to even store these files in your source control management system; you can retain the historical versions in the repository if it is backed up (in the future, the Maven Repository Manager will allow POM updates from a web interface).

## 7.5. Continuous Integration with Continuum

If you are not already familiar with it, continuous integration enables automated builds of your project on a regular interval, ensuring that conflicts are detected earlier in a project's release life cycle, rather than close to a release. More than just nightly builds, continuous integration can enable a better development culture where team members can make smaller, iterative changes that can more easily support concurrent development processes. As such, continuous integration is a key element of effective collaboration.

Continuum is Maven's continuous integration and build server. In this chapter, you will pick up the Proficio example from earlier in the book, and learn how to use Continuum to build this project on a regular basis. The examples discussed are based on Continuum 1.0.3, however newer versions should be similar. The examples also assumes you have Subversion installed, which you can obtain for your operating system from <http://subversion.tigris.org/>.

First, you will need to install Continuum. This is very simple – once you have downloaded it and unpacked it, you can run it using the following command:

```
C:\mvnbook\continuum-1.0.3> bin\win32\run
```

There are scripts for most major platforms, as well as the generic `bin/plexus.sh` for use on other Unix-based platforms. Starting up continuum will also start a http server and servlet engine.

You can verify the installation by viewing the web site at <http://localhost:8080/continuum/>.

The first screen to appear will be the one-time setup page shown in figure 7-1. The configuration on the screen is straight forward – all you should need to enter are the details of the administration account you'd like to use, and the company information for altering the logo in the top left of the screen.

For most installations this is all the configuration that's required, however, if you are running Continuum on your desktop and want to try the examples in this section, some additional steps are required. As of Continuum 1.0.3, these additional configuration requirements can be set only after the previous step has been completed, and you must stop the server to make the changes (to stop the server, press **Ctrl-C** in the window that is running Continuum).

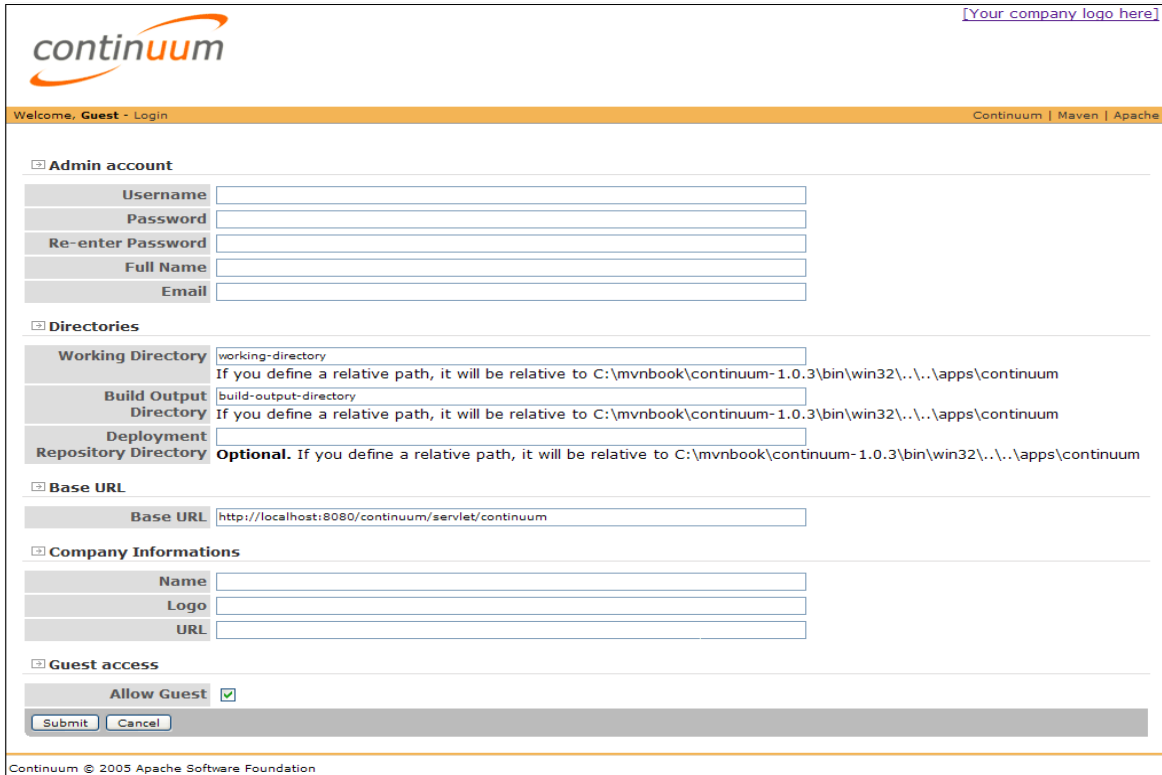


Figure 7-1: The Continuum setup screen

To complete the Continuum setup page, you can cut and paste field values from the following list:

Field Name	Value
Working Directory	working-directory
Build Output Directory	build-output-directory
Base URL	http://localhost:8080/continuum/servlet/continuum

In the following examples, POM files will be read from the local hard disk where the server is running. By default, this is disabled as a security measure, since paths can be entered from the web interface. To enable this setting, edit `apps/continuum/conf/application.xml` and verify the following line isn't commented out:

```
...
<implementation>
  org.codehaus.plexus.formica.validation.UrlValidator
</implementation>
<configuration>
  <allowedSchemes>
    ...
    <allowedScheme>file</allowedScheme>
  </allowedSchemes>
</configuration>
...
```

To have Continuum send you e-mail notifications, you will also need an SMTP server to which to send the messages. The default is to use `localhost:25`. If you do not have this set up on your machine, edit the file above to change the `smtp-host` setting. For instructions, refer to <http://maven.apache.org/continuum/guides/mini/guide-configuration.html>.

After these steps are completed, you can start Continuum again.

The next step is to set up the Subversion repository for the examples. This requires obtaining the `Code_Ch07.zip` archive and unpacking it in your environment. You can then check out Proficio from that location, for example if it was unzipped in `C:\mvnbook\svn`:

```
C:\mvnbook> svn co file://localhost/C:/mvnbook/svn/proficio/trunk \
proficio
```

The POM in this repository is not completely configured yet, since not all of the required details were known at the time of its creation. Edit `proficio/pom.xml` to correct the e-mail address to which notifications will be sent, and edit the location of the Subversion repository, by uncommenting and modifying the following lines:

```
...
<ciManagement>
  <system>continuum</system>
  <url>http://localhost:8080/continuum
  <notifiers>
    <notifier>
      <type>mail</type>
      <configuration>
        <address>youremail@yourdomain.com</address>
      </configuration>
    </notifier>
  </notifiers>
</ciManagement>
...
<scm>
  <connection>
    scm:svn:file://localhost/c:/mvnbook/svn/proficio/trunk
  </connection>
  <developerConnection>
    scm:svn:file://localhost/c:/mvnbook/svn/proficio/trunk
  </developerConnection>
</scm>
...
<distributionManagement>
  <site>
    <id>website</id>
    <url>
      file://localhost/c:/mvnbook/repository/sites/proficio
      /reference/${project.version}
    </url>
  </site>
</distributionManagement>
...
```

The `ciManagement` section is where the project's continuous integration is defined and in the above example has been configured to use Continuum locally on port 8080.

The `distributionManagement` setting will be used in a later example to deploy the site from your continuous integration environment. This assumes that you are still running the repository Web server on `localhost:8081`, from the directory `C:\mvnbook\repository`. If you haven't done so already, refer to section 7.3 for information on how to set this up.

Once these settings have been edited to reflect your setup, commit the file with the following command:

```
C:\mvnbook\proficio> svn ci -m 'my settings' pom.xml
```

You should build all these modules to ensure everything is in order, with the following command:

```
C:\mvnbook\proficio> mvn install
```

You are now ready to start using Continuum.

If you return to the `http://localhost:8080/continuum/` location that was set up previously, you will see an empty project list. Before you can add a project to the list, or perform other tasks, you must either log in with the administrator account you created during installation, or with another account you have since created with appropriate permissions. The login link is at the top-left of the screen, under the Continuum logo.

Once you have logged in, you can now select *Maven 2.0+ Project* from the *Add Project* menu. This will present the screen shown in figure 7-2. You have two options: you can provide the URL for a POM, or upload from your local drive. While uploading is a convenient way to configure from your existing check out, in Continuum 1.0.3 this does not work when the POM contains modules, as in the Proficio example. Instead, enter the `file://` URL as shown. When you set up your own system later, you will enter either a HTTP URL to a POM in the repository, a ViewCVS installation, or a Subversion HTTP server.

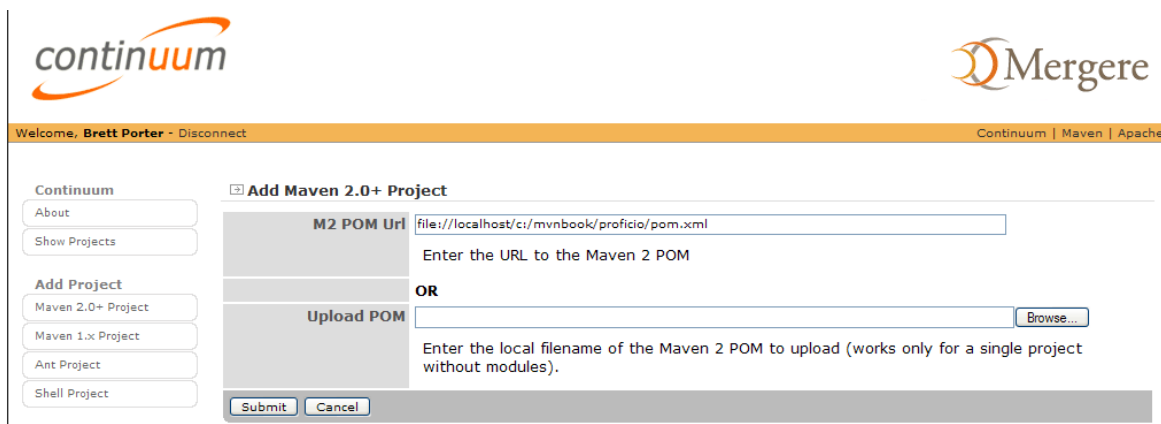


Figure 7-2: Add project screen shot

This is all that is required to add a Maven 2 project to Continuum. After submitting the URL, Continuum will return to the project summary page, and each of the modules will be added to the list of projects. Initially, the builds will be marked as *New* and their checkouts will be queued. The result is shown in figure 7-3.

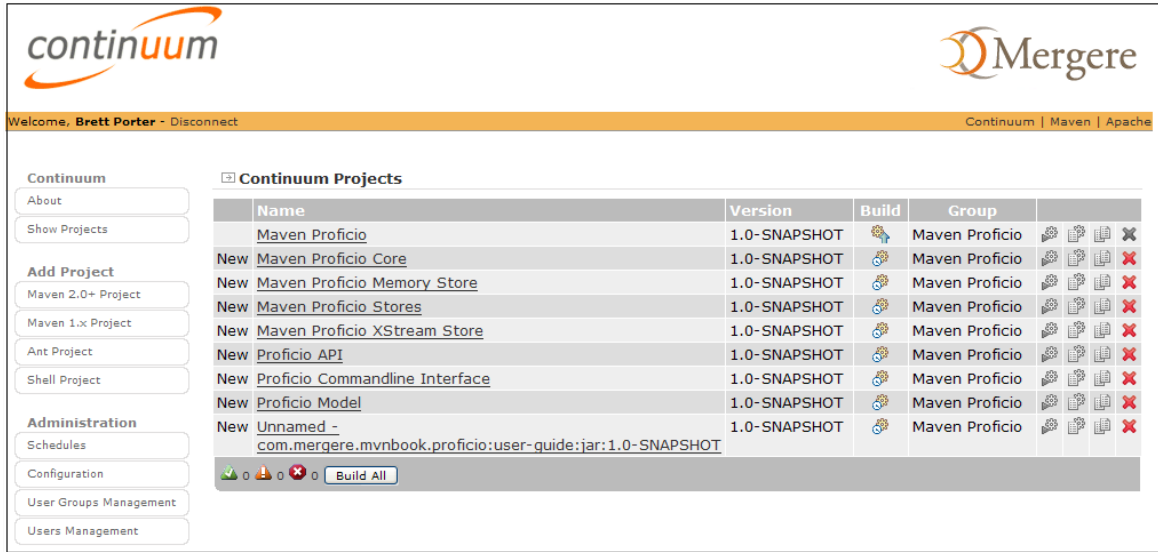


Figure 7-3: Summary page after projects have built

Continuum will now build the project hourly, and send an e-mail notification if there are any problems. If you want to put this to the test, go to your earlier checkout and introduce an error into `Proficio.java`, for example, remove the interface keyword:

```
[...]
public Proficio
[...]
```

Now, check the file in:

```
C:\mvnbook\proficio\proficio-api> svn ci -m 'introduce error' \
src/main/java/com/mergere/mvnbook/proficio/Proficio.java
```

Finally, press **Build Now** on the Continuum web interface next to the *Proficio API* module. First, the build will show an “In progress” status, and then fail, marking the left column with an “!” to indicate a failed build (you will need to refresh the page using the *Show Projects* link in the navigation to see these changes). In addition, you should receive an e-mail at the address you configured earlier. The *Build History* link can be used to identify the failed build and to obtain a full output log.

To avoid receiving this error every hour, restore the file above to its previous state and commit it again. The build in Continuum will return to the successful state.

This chapter will not discuss all of the features available in Continuum, but you may wish to go ahead and try them. For example, you might want to set up a notification to your favorite instant messenger – IRC, Jabber, MSN and Google Talk are all supported.



Regardless of which continuous integration server you use, there are a few tips for getting the most out of the system:

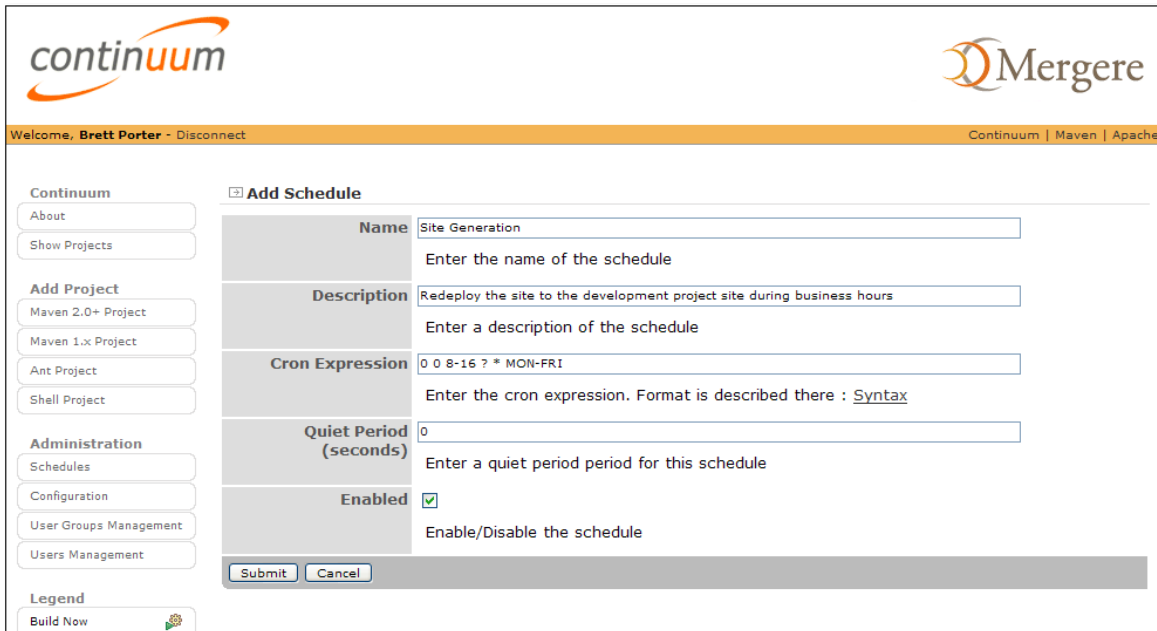
- **Commit early, commit often.** Continuous integration is most effective when developers commit regularly. This doesn't mean committing incomplete code, but rather keeping changes small and well tested. This will make it much easier to detect the source of an error when the build does break.
- **Run builds as often as possible.** This will be constrained by the length of the build and the available resources on the build machine, but it is best to detect a failure as soon as possible, before the developer moves on or loses focus. Continuum can be configured to trigger a build whenever a commit occurs, if the source control repository supports post-commit hooks. This also means that builds should be fast – long integration and performance tests should be reserved for periodic builds.
- **Fix builds as soon as possible.** While this seems obvious, it is often ignored. Continuous integration will be pointless if developers repetitively ignore or delete broken build notifications, and your team will become desensitized to the notifications in the future.
- **Establish a stable environment.** Avoid customizing the JDK, or local settings, if it isn't something already in use in other development, test and production environments. When a failure occurs in the continuous integration environment, it is important that it can be isolated to the change that caused it, and independent of the environment being used.
- **Run clean builds.** While rapid, iterative builds are helpful in some situations, it is also important that failures don't occur due to old build state. Consider a regular, clean build. Continuum currently defaults to doing a clean build, and a future version will allow developers to request a fresh checkout, based on selected schedules.
- **Run comprehensive tests.** Continuous integration is most beneficial when tests are validating that the code is working as it always has, not just that the project still compiles after one or more changes occur. In addition, it is beneficial to test against all different versions of the JDK, operating system and other variables, periodically. Continuum has preliminary support for system profiles and distributed testing, enhancements that are planned for future versions.
- **Build all of a project's active branches.** If multiple branches are in development, the continuous integration environment should be set up for all of the active branches.
- **Run a copy of the application continuously.** If the application is a web application, for example, run a servlet container to which the application can be deployed from the continuous integration environment. This can be helpful for non-developers who need visibility into the state of the application, separate from QA and production releases.

In addition to the above best practices, there are two additional topics that deserve special attention: automated updates to the developer web site, and profile usage.

In Chapter 6, you learned how to create an effective site containing project information and reports about the project's health and vitality. For these reports to be of value, they need to be kept up-to-date. This is another way continuous integration can help with project collaboration and communication. Though it would be overkill to regenerate the site on every commit, it is recommended that a separate, but regular schedule is established for site generation.

Verify that you are still logged into your Continuum instance. Next, from the *Administration* menu on the left-hand side, select *Schedules*. You will see that currently, only the default schedule is available. Click the *Add* button to add a new schedule, which will be configured to run every hour during business hours (8am – 4pm weekdays).

The appropriate configuration is shown in figure 7-4.



The screenshot shows the Continuum web interface. On the left is a sidebar with navigation links: Continuum (About, Show Projects), Add Project (Maven 2.0+ Project, Maven 1.x Project, Ant Project, Shell Project), Administration (Schedules, Configuration, User Groups Management, Users Management), and Legend (Build Now). The main content area is titled 'Add Schedule' and contains a form with the following fields:

Field	Value
Name	Site Generation
Description	Redeploy the site to the development project site during business hours
Cron Expression	0 0 8-16 ? * MON-FRI
Quiet Period (seconds)	0
Enabled	<input checked="" type="checkbox"/>

At the bottom of the form are 'Submit' and 'Cancel' buttons.

Figure 7-4: Schedule configuration

To complete the schedule configuration, you can cut and paste field values from the following list:

Field Name	Value
Name	Site Generation
Description	Redeploy the site to the development project site during business hours
Cron Expression	0 0 8-16 ? * MON-FRI
Quiet Period (seconds)	0

The cron expression entered here is much like the one entered for a Unix crontab and is further described at <http://www.opensymphony.com/quartz/api/org/quartz/CronTrigger.html>. The example above runs at 8:00:00, 9:00:00,..., 16:00:00 from Monday to Friday.

The “quiet period” is a setting that delays the build if there has been a commit in the defined number of seconds prior. This is useful when using CVS, since commits are not atomic and a developer might be committing midway through a update. It is not typically needed if using Subversion.

Once you add this schedule, return to the project list, and select the top-most project, *Maven Proficio*. The project information shows just one build on the default schedule that installs the parent POM, but does not recurse into the modules (the `-N` or `--non-recursive` argument). Since this is the root of the multi-module build – and it will also detect changes to any of the modules – this is the best place from which to build the site. In addition to building the sites for each module, it can aggregate changes into the top-level site as required.

The downside to this approach is that Continuum will build any unchanged modules, as well – if this is a concern, use the non-recursive mode instead, and add the same build definition to all of the modules.



In Continuum 1.0.3, there is no way to make bulk changes to build definitions, so you will need to add the definition to each module individually.

In this example you will add a new build definition to run the site deployment for the entirety of the multi-module build, on the business hours schedule. To add a new build definition, click the *Add* button below the default build definition. The *Add Build Definition* screen is shown in figure 7-5.

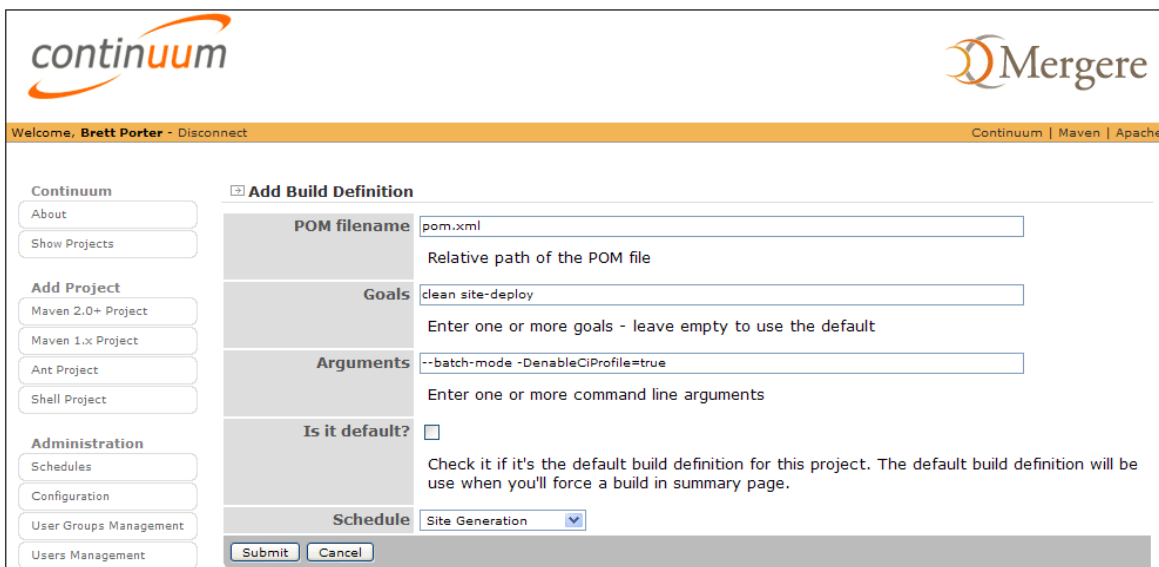


Figure 7-5: Adding a build definition for site deployment

To complete the *Add Build Definition* screen, you can cut and paste field values from the following list:

Field Name	Value
POM filename	<code>pom.xml</code>
Goals	<code>clean site-deploy</code>
Arguments	<code>--batch-mode -DenableCiProfile=true</code>

The goals to run are `clean` and `site-deploy`. The site will be deployed to the file system location you specified in the POM, when you first set up the Subversion repository earlier in this chapter, which will be visible from `http://localhost:8081/sites/proficio/`.

The arguments provided are `--batch-mode`, which is essential for all builds to ensure they don't block for user input, and `-DenableCiProfile=true`, which sets the given system property. The meaning of this system property will be explained shortly. The `--non-recursive` option is omitted.

You can see also that the schedule is set to use the site generation schedule created earlier, and that it is not the default build definition, which means that *Build Now* from the project summary page will not trigger this build. However, each build definition on the project information page (to which you would have been returned after adding the build definition) has a *Build Now* icon. Click this for the site generation build definition, and view the generated site from <http://localhost:8081/sites/proficio/>.

It is rare that the site build will fail, since most reports continue under failure conditions. However, if you want to fail the build based on these checks as well, you can add the test, verify or integration-test goal to the list of goals, to ensure these checks are run.



Any of these test goals should be listed after the site-deploy goal, so that if the build fails because of a failed check, the generated site can be used as reference for what caused the failure.

In the previous example, a system property called `enableCiProfile` was set to true. In Chapter 6, a number of plugins were set up to fail the build if certain project health checks failed, such as the percentage of code covered in the unit tests dropping below a certain value. However, these checks delayed the build for all developers, which can be a discouragement to using them.

If you compare the example `proficio/pom.xml` file in your Subversion checkout to that used in Chapter 6, you'll see that these checks have now been moved to a profile. Profiles are a means for selectively enabling portions of the build. If you haven't previously encountered profiles, please refer to Chapter 3. In this particular case, the profile is enabled only when the `enableCiProfile` system property is set to true.

```
...
<profiles>
  <profile>
    <id>ciProfile</id>
    <activation>
      <property>
        <name>enableCiProfile</name>
        <value>true</value>
      </property>
    </activation>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>
        <executions>
          ...
        </executions>
      </plugin>
    </plugins>
  </profile>
</profiles>
```

You'll find that when you run the build from the command line (as was done in Continuum originally), none of the checks added in the previous chapter are executed. The checks will be run when you enable the ciProfile using `mvn -DenableCiProfile=true`.

There are two ways to ensure that all of the builds added in Continuum use this profile. The first is to adjust the default build definition for each module, by going to the module information page, and clicking *Edit* next to the default build definition. As you saw before, it is necessary to do this for each module individually, at least in the version of Continuum current at the time of writing.

The other alternative is to set this profile globally, for all projects in Continuum. As Maven 2 is still executed as normal, it reads the `${user.home}/.m2/settings.xml` file for the user under which it is running, as well as the settings in the Maven installation. To enable this profile by default from these settings, add the following configuration to the `settings.xml` file in `<maven home>/conf/settings.xml`:

```
...
<activeProfiles>
...
  <activeProfile>ciProfile</activeProfile>
</activeProfiles>
...
```

In this case the identifier of the profile itself, rather than the property used to enable it, indicates that the profile is always active when these settings are read.

How you configure your continuous integration depends on the culture of your development team and other environmental factors such as the size of your projects and the time it takes to build and test them. The guidelines discussed in this chapter will help point your team in the right direction, but the timing and configuration can be changed depending upon your circumstances. For example, if the additional checks take too much time for frequent continuous integration builds, it may be necessary to schedule them separately for each module, or for the entire multi-module project to run the additional checks after the site has been generated, the `verify` goal may need to be added to the site deployment build definition, as discussed previously.

## 7.6. Team Dependency Management Using Snapshots

Chapter 3 of this book discussed how to manage your dependencies in a multi-module build, and while dependency management is fundamental to any Maven build, the team dynamic makes it critical.

In this section, you will learn about using snapshots more effectively in a team environment, and how to enable this within your continuous integration environment.

So far in this book, snapshots have been used to refer to the development version of an individual module. The generated artifacts of the snapshot are stored in the local repository, and in contrast to regular dependencies, which are not changed, these artifacts will be updated frequently. Projects in Maven stay in the snapshot state until they are released, which is discussed in section 7.8 of this chapter.

Snapshots were designed to be used in a team environment as a means for sharing development versions of artifacts that have already been built. Usually, in an environment where a number of modules are undergoing concurrent development, the build involves checking out all of the dependent projects and building them yourself. Additionally, in some cases, where projects are closely related, you must build all of the modules simultaneously from a master build.

While building all of the modules from source can work well and is handled by Maven inherently, it can lead to a number of problems:

- It relies on manual updates from developers, which can be error-prone. This will result in local inconsistencies that can produce non-working builds
- There is no common baseline against which to measure progress
- Building can be slower as multiple dependencies must be rebuilt also
- Changes developed against outdated code can make integration more difficult

As you can see from these issues, building from source doesn't fit well with an environment that promotes continuous integration. Instead, use binary snapshots that have been already built and tested.

In Maven, this is achieved by regularly deploying snapshots to a shared repository, such as the internal repository set up in section 7.3. Considering that example, you'll see that the repository was defined in `proficio/pom.xml`:

```
...
<distributionManagement>
  <repository>
    <id>internal</id>
    <url>file://localhost/c:/mvnbook/repository/internal</url>
  </repository>
</distributionManagement>
```

Now, deploy `proficio-api` to the repository with the following command:

```
C:\mvnbook\proficio\proficio-api> mvn deploy
```

You'll see that it is treated differently than when it was installed in the local repository. The filename that is used is similar to `proficio-api-1.0-20060211.131114-1.jar`. In this case, the version used is the time that it was deployed (in the UTC timezone) and the build number. If you were to deploy again, the time stamp would change and the build number would increment to 2.

This technique allows you to continue using the latest version by declaring a dependency on `1.0-SNAPSHOT`, or to lock down a stable version by declaring the dependency version to be the specific equivalent such as `1.0-20060211.131114-1`. While this is not usually the case, locking the version in this way may be important if there are recent changes to the repository that need to be ignored temporarily.

Currently, the Proficio project itself is not looking in the internal repository for dependencies, but rather relying on the other modules to be built first, though it may have been configured as part of your settings files. To add the internal repository to the list of repositories used by Proficio regardless of settings, add the following to `proficio/pom.xml`:

```
...
<repositories>
  <repository>
    <id>internal</id>
    <url>http://localhost:8081/internal</url>
  </repository>
</repositories>
...
```



If you are developing plugins, you may also want to add this as a `pluginRepository` element as well.

Now, to see the updated version downloaded, build `proficio-core` with the following command:

```
C:\mvnbook\proficio\proficio-core> mvn -U install
```

During the build, you will see that some of the dependencies are checked for updates, similar to the example below (note that this output has been abbreviated):

```
...
proficio-api:1.0-SNAPSHOT: checking for updates from internal
...
```

The `-U` argument in the prior command is required to force Maven to update all of the snapshots in the build. If it were omitted, by default, no update would be performed. This is because the default policy is to update snapshots daily – that is, to check for an update the first time that particular dependency is used after midnight local time.

You can always force the update using the `-U` command, but you can also change the interval by changing the repository configuration. To see this, add the following configuration to the repository configuration you defined above in `proficio/pom.xml`:

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

In this example, any snapshot dependencies will be checked once an hour to determine if there are updates in the remote repository. The settings that can be used for the update policy are `never`, `always`, `daily` (the default), and `interval:minutes`.



Whenever you use the `-U` argument, it updates both releases and snapshots. This causes many plugins to be checked for updates, as well as updating any version ranges.

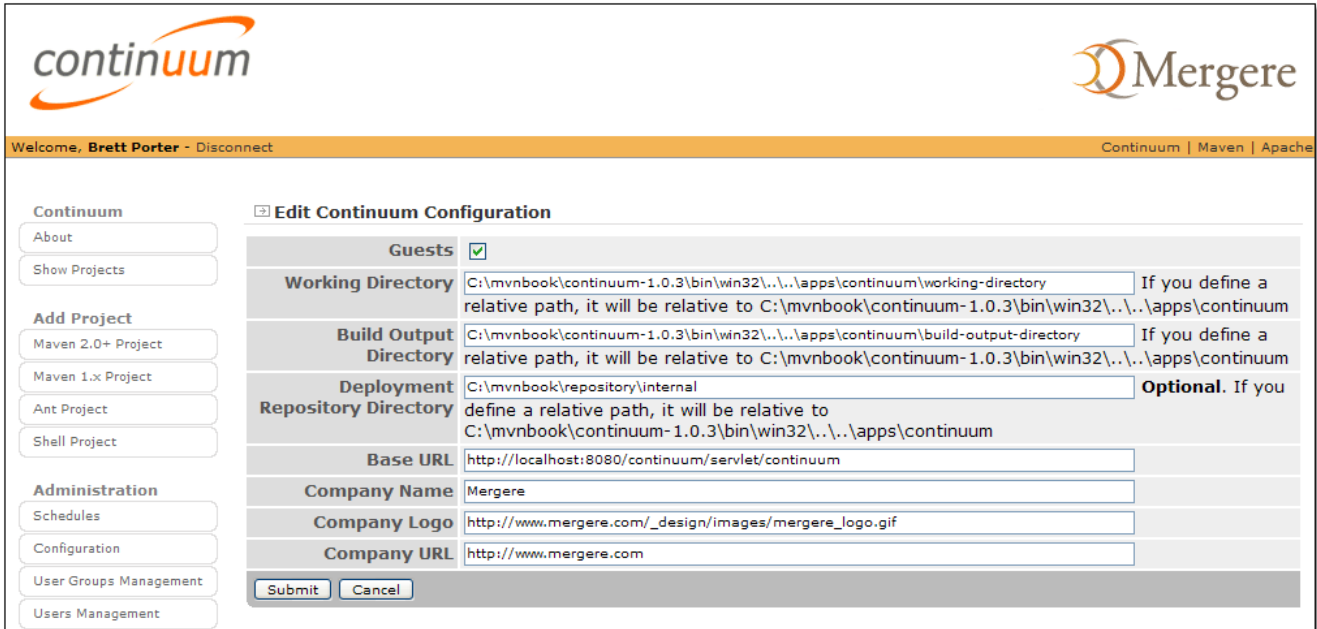
This technique can ensure that developers get regular updates, without having to manually intervene, and without slowing down the build by checking on every access (as would be the case if the policy were set to `always`). However, the updates will still occur only as frequently as new versions are deployed to the repository.

It is possible to establish a policy where developers do an update from the source control management (SCM) system before committing, and then deploy the snapshot to share with the other team members. However, this introduces a risk that the snapshot will not be deployed at all, deployed with uncommitted code, or deployed without all the updates from the SCM, making it out-of-date. Several of the problems mentioned earlier still exist – so at this point, all that is being saved is some time, assuming that the other developers have remembered to follow the process.



A much better way to use snapshots is to automate their creation. Since the continuous integration server regularly rebuilds the code from a known state, it makes sense to have it build snapshots, as well.

How you implement this will depend on the continuous integration server that you use. To deploy from your server, you must ensure that the `distributionManagement` section of the POM is correctly configured. However, as you saw earlier, Continuum can be configured to deploy its builds to a Maven snapshot repository automatically. If there is a repository configured to which to deploy them, this feature is enabled by default in a build definition. So far in this section, you have not been asked to apply this setting, so let's go ahead and do it now. Log in as an administrator and go to the following Configuration screen, shown in figure 7-6.



Field Name	Value
Guests	<input checked="" type="checkbox"/>
Working Directory	C:\mvnbook\continuum-1.0.3\bin\win32\...\apps\continuum\working-directory
Build Output Directory	C:\mvnbook\continuum-1.0.3\bin\win32\...\apps\continuum\build-output-directory
Deployment Repository Directory	C:\mvnbook\repository\internal
Base URL	http://localhost:8080/continuum/servlet/continuum
Company Name	Mergere
Company Logo	http://www.mergere.com/_design/images/mergere_logo.gif
Company URL	http://www.mergere.com

Figure 7-6: Continuum configuration

To complete the Continuum configuration page, you can cut and paste field values from the following list:

Field Name	Value
Working Directory	C:\mvnbook\continuum-1.0.3\bin\win32\...\apps\continuum\working-directory
Build Output Directory	C:\mvnbook\continuum-1.0.3\bin\win32\...\apps\continuum\build-output-directory
Deployment Repository Directory	C:\mvnbook\repository\internal
Base URL	http://localhost:8080/continuum/servlet/continuum
Company Name	Mergere
Company Logo	http://www.mergere.com/_design/images/mergere_logo.gif
Company URL	http://www.mergere.com



---

The Deployment Repository Directory field entry relies on your internal repository and Continuum server being in the same location. If this is not the case, you can enter a full repository URL such as `scp://repositoryhost/www/repository/internal`.

---

To try this feature, follow the *Show Projects* link, and click *Build Now* on the *Proficio API* project. Once the build completes, return to your console and build `proficio-core` again using the following command:

```
C:\mavenbook\proficio\proficio-core> mvn -U install
```

You'll notice that a new version of `proficio-api` is downloaded, with an updated time stamp and build number.

With this setup, you can avoid all of the problems discussed previously. Better yet, while you get regular updates from published binary dependencies, when necessary, you can either lock a dependency to a particular build, or build from source.

Another point to note about snapshots is that it is possible to store them in a separate repository from the rest of your released artifacts. This can be useful if you need to clean up snapshots on a regular interval, but still keep a full archive of releases.

If you are using the regular deployment mechanism (instead of using Continuum), this separation is achieved by adding an additional repository to the `distributionManagement` section of your POM. For example, if you had a snapshot-only repository in `/www/repository/snapshots`, you would add the following:

```
...
<distributionManagement>
...
  <snapshotRepository>
    <id>internal.snapshots</id>
    <url>file://localhost/www/repository/snapshots</url>
  </snapshotRepository>
</distributionManagement>
...
```

This will deploy to that repository whenever the version contains `SNAPSHOT`, and deploy to the regular repository you listed earlier, when it doesn't.

Given this configuration, you can make the snapshot update process more efficient by not checking the repository that has only releases for updates. The replacement repository declarations in your POM would look like this:

```
...
<repositories>
  <repository>
    <id>internal</id>
    <url>http://localhost:8081/internal</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>internal.snapshots</id>
    <url>http://localhost:8081/snapshots</url>
    <snapshots>
      <updatePolicy>interval:60</updatePolicy>
    </snapshots>
  </repository>
</repositories>
...
```

## 7.7. Creating a Standard Project Archetype

Throughout this book, you have seen the archetypes that were introduced in Chapter 2 used to quickly lay down a project structure. While this is convenient, there is always some additional configuration required, either in adding or removing content from that generated by the archetypes. To avoid this, you can create one or more of your own archetypes.

Beyond the convenience of laying out a project structure instantly, archetypes give you the opportunity to start a project in the right way – that is, in a way that is consistent with other projects in your environment. As you saw in this chapter, the requirement of achieving consistency is a key issue facing teams.

Writing an archetype is quite like writing your own project, and replacing the specific values with parameters. There are two ways to create an archetype: one based on an existing project using `mvn archetype:create-from-project`, and the other, by hand, using an archetype. To get started with the archetype, run the following command:

```
C:\mvnbook\proficio> mvn archetype:create \
-DgroupId=com.mergere.mvnbook \
-DartifactId=proficio-archetype \
-DarchetypeArtifactId=maven-archetype-archetype
```

The layout of the resulting archetype is shown in figure 7-7.

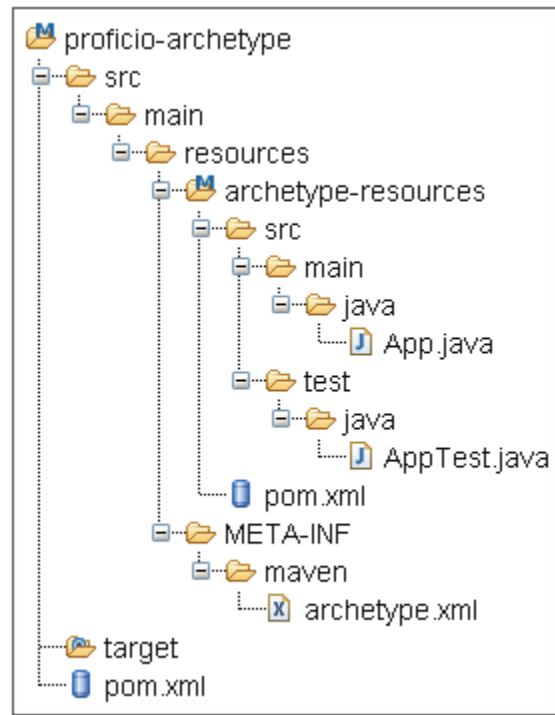


Figure 7-7: Archetype directory layout

If you look at `pom.xml` at the top level, you'll see that the archetype is just a normal JAR project – there is no special build configuration required. The JAR that is built is composed only of resources, so everything else is contained under `src/main/resources`. There are two pieces of information required: the archetype descriptor in `META-INF/maven/archetype.xml`, and the template project in `archetype-resources`.

The archetype descriptor describes how to construct a new project from the archetype-resources provided. The example descriptor looks like the following:

```
<archetype>
  <id>proficio-archetype</id>
  <sources>
    <source>src/main/java/App.java</source>
  </sources>
  <testSources>
    <source>src/test/java/AppTest.java</source>
  </testSources>
</archetype>
```

Each tag is a list of files to process and generate in the created project. The example above shows the sources and test sources, but it is also possible to specify files for resources, `testResources`, and `siteResources`.

The files within the archetype-resources section are Velocity templates. These files will be used to generate the template files when the archetype is run. For this example, the `pom.xml` file looks like the following:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>$groupId</groupId>
  <artifactId>$artifactId</artifactId>
  <version>$version</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

As you can see, the `groupId`, `artifactId` and `version` elements are variables that will be substituted with the values provided by the developer running `archetype:create`.

From here, you need to populate the template with the content that you'd like to have applied consistently to new projects. For more information on creating an archetype, refer to the documentation on the Maven web site.

Once you have completed the content in the archetype, Maven will build, install and deploy it like any other JAR. Continuing from the example in section 7.3 of this chapter, you will use the “internal” repository. Since the archetype inherits the Proficio parent, it has the correct deployment settings already, so you can run the following command:

```
C:\mvnbook\proficio\proficio-archetype> mvn deploy
```

The archetype is now ready to be used. To do so, go to an empty directory and run the following command:

```
C:\mvnbook> mvn archetype:create -DgroupId=com.mergere.mvnbook \
-DartifactId=proficio-example \
-DarchetypeGroupId=com.mergere.mvnbook \
-DarchetypeArtifactId=proficio-archetype \
-DarchetypeVersion=1.0-SNAPSHOT
```

Normally, the `archetypeVersion` argument is not required at this point. However, since the archetype has not yet been released, if omitted, the required version would not be known (or if this was later development, a previous release would be used instead). Releasing a project is explained in section 7.8 of this chapter.

You now have the template project laid out in the `proficio-example` directory. It will look very similar to the content of the `archetype-resources` directory you created earlier, now however, the content of the files will be populated with the values that you provided on the command line.

## 7.8. Cutting a Release

Releasing software is difficult. It is usually tedious and error prone, full of manual steps that need to be completed in a particular order. Worse, it happens at the end of a long period of development when all everyone on the team wants to do is get it out there, which often leads to omissions or short cuts. Finally, once a release has been made, it is usually difficult or impossible to correct mistakes other than to make another, new release.

Once the definition for a release has been set by a team, releases should be consistent every time they are built, allowing them to be highly automated. Maven provides a release plugin that provides the basic functions of a standard release process. The release plugin takes care of a number of manual steps in updating the project POM, updating the source control management system to check and commit release related changes, and creating tags (or equivalent for your SCM).

The release plugin operates in two steps: *prepare* and *perform*. The prepare step is run once for a release, and does all of the project and source control manipulation that results in a tagged version. The perform step could potentially be run multiple times to rebuild a release from a clean checkout of the tagged version, and to perform standard tasks, such as deployment to the remote repository.

To demonstrate how the release plugin works, the Proficio example will be revisited, and released as 1.0. You can continue using the code that you have been working on in the previous sections, or check out the following:

```
C:\mvnbook> svn co \  
file://localhost/C:/mvnbook/svn/proficio/tags/proficio-final \  
proficio
```

To start the release process, run the following command:

```
c:\mvnbook\proficio> mvn release:prepare -DdryRun=true
```

This simulates a normal release preparation, without making any modifications to your project. You'll notice that each of the modules in the project is considered. As the command runs, you will be prompted for values. Accept the defaults in this instance (note that running Maven in “batch mode” avoids these prompts and will accept all of the defaults).

---

In this project, all of the dependencies being used are releases, or part of the project. However, if you are using a dependency that is a snapshot, an error will appear.

The prepare step ensures that there are no snapshots in the build, other than those that will be released as part of the process (that is, other modules). This is because the prepare step is attempting to guarantee that the build will be reproducible in the future, and snapshots are a transient build, not ready to be used as a part of a release.

In some cases, you may encounter a plugin snapshot, even if the plugin is not declared in the POM. This is because you are using a locally installed snapshot of a plugin (either built yourself, or obtained from the development repository of the Maven project) that is implied through the build life cycle. This can be corrected by adding the plugin definition to your POM, and setting the version to the latest release (But only after verifying that your project builds correctly with that version!).

---

To review the steps taken in this release process:

1. Check for correct version of the plugin and POM (for example, the appropriate SCM settings)
2. Check if there are any local modifications
3. Check for snapshots in dependency tree
4. Check for snapshots of plugins in the build
5. Modify all POM files in the build, as they will be committed to the tag
6. Run `mvn clean integration-test` to verify that the project will successfully build
7. Describe other preparation goals (none are configured by default, but this might include updating the metadata in your issue tracker, or creating and committing an announcement file)
8. Describe the SCM commit and tag operations
9. Modify all POM files in the build, as they will be committed for the next development iteration
10. Describe the SCM commit operation

You might like to review the POM files that are created for steps 5 and 9, named `pom.xml.tag` and `pom.xml.next` respectively in each module directory, to verify they are correct. You'll notice that the version is updated in both of these files, and is set based on the values for which you were prompted during the release process. The SCM information is also updated in the tag POM to reflect where it will reside once it is tagged, and this is reverted in the next POM.

However, these changes are not enough to guarantee a reproducible build – it is still possible that the plugin versions will vary, that resulting version ranges will be different, or that different profiles will be applied. For that reason, there is also a `release-pom.xml.tag` file written out to each module directory. This contains a resolved version of the POM that Maven will use to build from if it exists. In this POM, a number of changes are made:

- the explicit version of plugins and dependencies that were used are added
- any settings from `settings.xml` (both per-user and per-installation) are incorporated into the POM.
- any active profiles are explicitly activated, including profiles from `settings.xml` and `profiles.xml`

---

You may have expected that inheritance would have been resolved by incorporating any parent elements that are used, or that expressions would have been resolved. This is not the case however, as these can be established from the other settings already populated in the POM in a reproducible fashion.

---

When the final run is executed, this file will be `release-pom.xml` in the same directory as `pom.xml`. This is used by Maven, instead of the normal POM, when a build is run from this tag to ensure it matches the same circumstances as the release build.

Having run through this process you may have noticed that only the unit and integration tests were run as part of the test build. Recall from Chapter 6 that you learned how to configure a number of checks – so it is important to verify that they hold as part of the release. Also, recall that in section 7.5, you created a profile to enable those checks conditionally. To include these checks as part of the release process, you need to enable this profile during the verification step. To do so, use the following plugin configuration:

```
[...]
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-release-plugin</artifactId>
  <configuration>
    <arguments>-DenableCiProfile=true</arguments>
  </configuration>
</plugin>
[...]
```

Try the dry run again:

```
C:\mvnbook\proficio> mvn release:prepare -DdryRun=true
```

Now that you've gone through the test run and are happy with the results, you can go for the real thing with the following command:

```
C:\mvnbook\proficio> mvn release:prepare
```

You'll notice that this time the operations on the SCM are actually performed, and the updated POM files are committed.

You won't be prompted for values as you were the first time – since by the default, the release plugin will resume a previous attempt by reading the `release.properties` file that was created at the end of the last run. If you need to start from the beginning, you can remove that file, or run `mvn -Dresume=false release:prepare` instead.

Once this is complete, you'll see in your SCM the new tag for the project (with the modified files), while locally, the version is now `1.1-SNAPSHOT`.

However, the release still hasn't been generated yet – for that, you need to deploy the build artifacts. This is achieved with the `release:perform` goal. This is run as follows:

```
C:\mvnbook\proficio> mvn release:perform
```

No special arguments are required, because the `release.properties` file still exists to tell the goal the version from which to release. To release from an older version, or if the `release.properties` file had been removed, you would run the following:

```
C:\mvnbook\proficio> mvn release:perform -DconnectionUrl=\
scm:svn:file://localhost/c:/mvnbook/svn/proficio/tags/proficio-1.0
```

If you follow the output above, you'll see that a clean checkout was obtained from the created tag, before running Maven from that location with the goals `deploy site-deploy`. This is the default for the release plugin – to deploy all of the built artifacts, and to deploy a copy of the site.

If this is not what you want to run, you can change the goals used with the `goals` parameter:

```
C:\mvnbook\proficio> mvn release:perform -Dgoals="deploy"
```

However, this requires that you remember to add the parameter every time. Since the goal is for consistency, you want to avoid such problems. To do so, add the following goals to the POM:

```
[...]
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-release-plugin</artifactId>
  <configuration>
    <goals>deploy</goals>
  </configuration>
</plugin>
[...]
```

You may also want to configure the release plugin to activate particular profiles, or to set certain properties. Refer to the plugin reference at <http://maven.apache.org/plugins/maven-release-plugin/> for more information. It is important in these cases that you consider the settings you want, before you run the `release:prepare` goal, though. To ensure reproducibility, the release plugin will confirm that the checked out project has the same release plugin configuration as those being used (with the exception of goals).

When the release is performed, and the built artifacts are deployed, you can examine the files that are placed in the SCM repository. To do this, check out the tag:

```
C:\mvnbook> svn co \
file:///localhost/mvnbook/svn/proficio/tags/proficio-1.0
```

You'll notice that the contents of the POM match the `pom.xml` file, and not the `release-pom.xml` file. The reason for this is that the POM files in the repository are used as dependencies and the original information is more important than the release-time information – for example, it is necessary to know what version ranges are allowed for a dependency, rather than the specific version used for the release. For the same reason, both the original `pom.xml` file and the `release-pom.xml` files are included in the generated JAR file.

Also, during the process you will have noticed that Javadoc and source JAR files were produced and deployed into the repository for all the Java projects. These are configured by default in the Maven POM as part of a profile that is activated when the release is performed.

You can disable this profile by setting the `useReleaseProfile` parameter to `false`, as follows:

```
[...]
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-release-plugin</artifactId>
  <configuration>
    <useReleaseProfile>false</useReleaseProfile>
  </configuration>
</plugin>
[...]
```



Instead, you may want to include additional actions in the profile, without having to declare and enable an additional profile. To do this, define a profile with the identifier `release-profile`, as follows:

```
[...]
<profiles>
  <profile>
    <id>release-profile</id>

    <build>
      <!-- Extra plugin configuration would be inserted here -->
    </build>

  </profile>
</profiles>
[...]
```

After the release process is complete, the only step left is to clean up after the plugin, removing `release.properties` and any POM files generated as a result of the dry run. Simply run the following command to clean up:

```
C:\mvnbook\proficio> mvn release:clean
```

## 7.9. Summary

As you've seen throughout this chapter, and indeed this entire book, Maven was designed to address issues that directly affect teams of developers. All of the features described in this chapter can be used by any development team. So, whether your team is large or small, Maven provides value by standardizing and automating the build process.

There are also strong team-related benefits in the preceding chapters – for example, the adoption of reusable plugins can capture and extend build knowledge throughout your entire organization, rather than creating silos of information around individual projects. The site and reports you've created can help a team communicate the status of a project and their work more effectively. And all of these features build on the essentials demonstrated in chapters 1 and 2 that facilitate consistent builds.

Lack of consistency is the source of many problems when working in a team, and while Maven focuses on delivering consistency in your build infrastructure through patterns, it can aid you in effectively using tools to achieve consistency in other areas of your development. This in turn can lead to and facilitate best practices for developing in a community-oriented, real-time engineering style, by making information about your projects visible and organized.