

Maven

A Developer's Notebook™

Vincent Massol
& Timothy M. O'Brien

- Organizing Projects
- Discovering Plugins
- Project Reporting
- Team Collaboration
- Reliable builds

O'REILLY®

Tracking Project Activity

Tracking a project's activity is very important for end users/stakeholders. Several good indicators are available that can aid in tracking:

- The number of commits per day
- The number of additional unit tests added every day
- The evolution of test percentage coverage
- The mailing list activity
- The number of existing books on the project (mostly for open source projects)
- The number of committers
- The number of open issues in the project's bug tracker
- The number of years the project has been in existence

It would be very nice to have the equivalent of the Dashboard plug-in, but for project activity. This could provide a global project activity score à la SourceForge project activity percentage (e.g., http://sourceforge.net/project/stats/?group_id=15278).* Unfortunately, such a comprehensive plug-in does not yet exist! Instead, you have several ad hoc possibilities:

- Use the StatCVS-XML plug-in to analyze a CVS repository and generate statistics.† Unfortunately, such a tool doesn't exist for Subversion yet, but it won't be long before one does.
- Use the Developer-Activity and File-Activity Maven plug-ins that respectively report on developer commits and files containing the most changes, within a given date range.

Let's discover how to use these plug-ins.

How do I do that?

To use the StatCVS-XML plug-in you must install it, as it's not part of the default Maven distribution. Install the plug-in from <http://statcvs-xml.berlios.de/>, following the installation steps described in the and in Chapter 6. You'll also need to ensure you have a command-line CVS cli-

* SourceForge's current ranking formula is: $\log(3 * \# \text{ of forum posts for that week}) + \log(4 * \# \text{ of tasks ftw}) + \log(3 * \# \text{ bugs ftw}) + \log(10 * \text{ patches ftw}) + \log(5 * \text{ tracker items ftw}) + \log(\# \text{ commits to CVS ftw}) + \log(5 * \# \text{ file releases ftw}) + \log(.3 * \# \text{ downloads ftw})$.

† Unfortunately, no such tool exists for Subversion—but it won't be long before one does.

ent installed, as the plug-in is using the Ant cvs task to gather CVS logs for your project.

Using the plug-in cannot be simpler: add the maven-statcvs-plugin report to the reports section of your project's POM and run `maven site`. Myriad reports are generated, but here are a few that you should know more about in order to understand a project's activity. Figure 4-10 shows the commit activity per author and the aggregated activity over the whole lifetime of the Jakarta Cactus project.

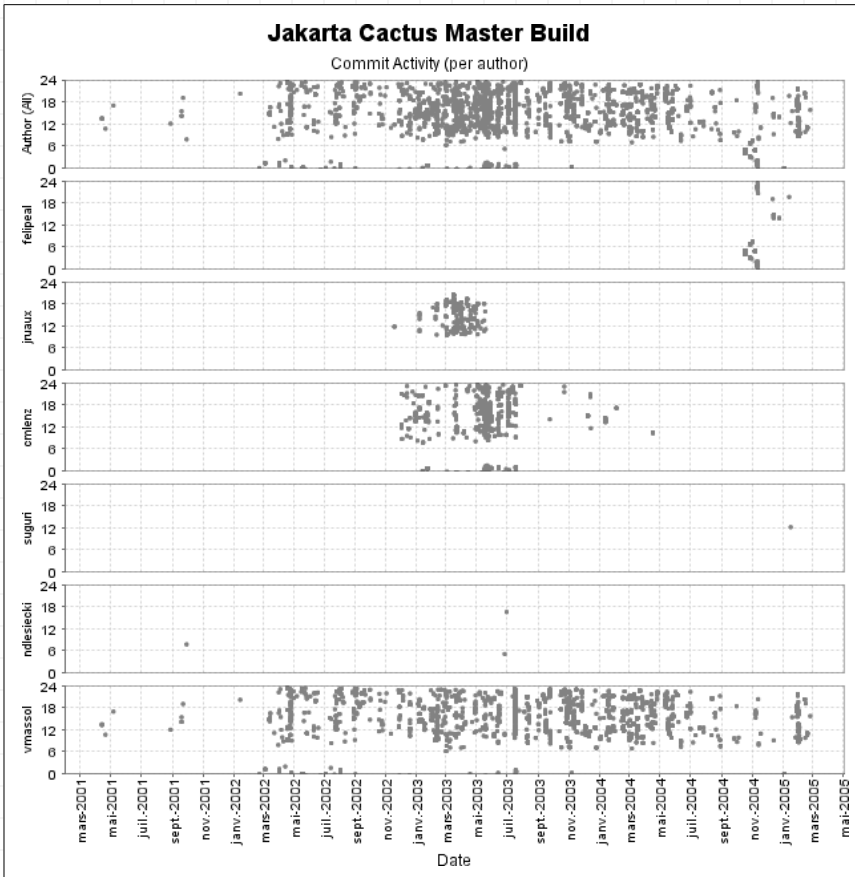


Figure 4-10. Commit activity over time

Another very interesting report shows changes brought to a project over time, module by module, as shown in Figure 4-11 for the StatCVS-XML project itself.

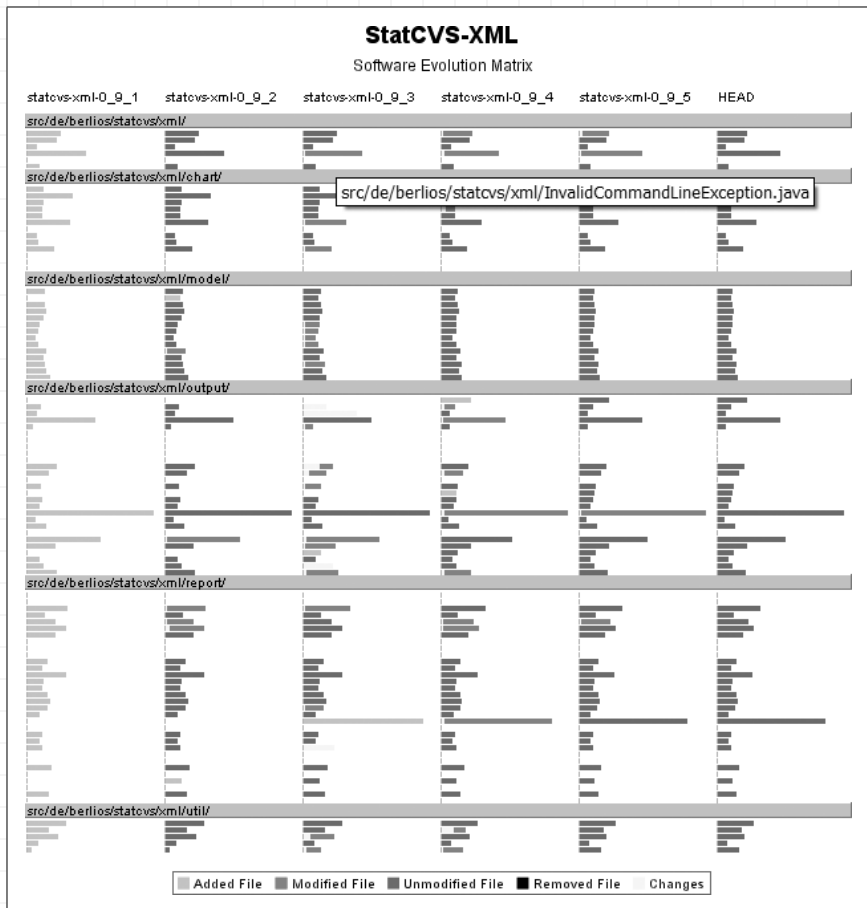


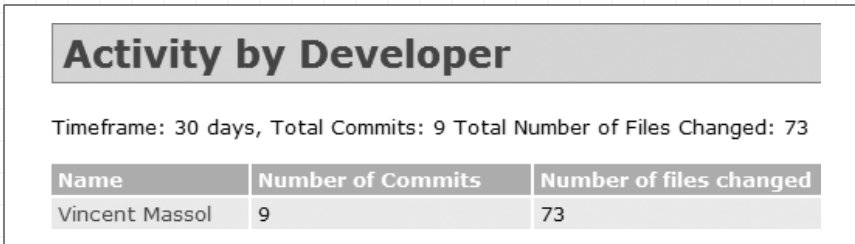
Figure 4-11. Changes to the StatCVS-XML project over time, module by module

The goal here is not to take you through a full-length tutorial of StatCVS-XML, but rather to show you its power and how you can integrate it in a Maven project. You are strongly encouraged to explore it on your own at <http://statcv-sxml.berlios.de/>.

Although StatCVS-XML generates developer and file activity reports, it's still interesting to find out how to use the Developer-Activity and File-Activity plug-ins, as they also support Subversion. To use them, simply add them as usual to your POM. Let's do that on `qotd/core/project.xml`:

```
<reports>
  <report>maven-developer-activity-plugin</report>
  <report>maven-file-activity-plugin</report>
  [...]
</reports>
```

Running `maven site` generates the reports shown in Figure 4-12 and Figure 4-13.

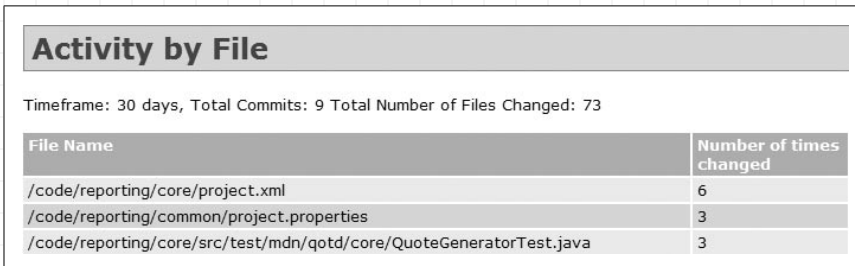


Activity by Developer

Timeframe: 30 days, Total Commits: 9 Total Number of Files Changed: 73

Name	Number of Commits	Number of files changed
Vincent Massol	9	73

Figure 4-12. Developer activity showing number of commits and files modified



Activity by File

Timeframe: 30 days, Total Commits: 9 Total Number of Files Changed: 73

File Name	Number of times changed
/code/reporting/core/project.xml	6
/code/reporting/common/project.properties	3
/code/reporting/core/src/test/mdn/qotd/core/QuoteGeneratorTest.java	3

Figure 4-13. Files which have been changed recently, ordered by change frequency

Both of these reports internally use the Changelog plug-in that you'll learn about in the next lab. You can configure them by using Changelog plug-in properties, such as the `maven.changelog.range` property that controls the report timeframe.

Tracking Project Changes

When you're developing a project, it's useful to provide to your users a list of changes between the previous version of the project and the current version. This allows users who are expecting a feature or a bug fix to verify that the enhancement is in the new version. But even more important is the fact that you can regularly publish the progress report, which will keep your users and/or stakeholders spellbound!

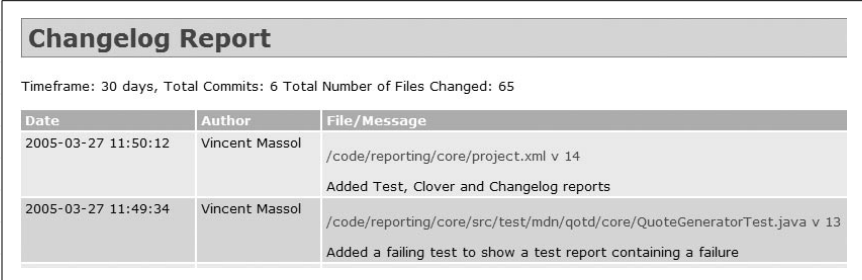
How do I do that?

The first solution that comes to mind is to use the Maven Changelog plug-in. It extracts commit logs from your SCM and generates a report. It has a `maven.changelog.range` property that controls how many days in

the past to look when performing the extraction. To use it, add it to your reports section in *qotd/core/project.xml*:

```
<reports>
  <report>maven-changelog-plugin</report>
  [...]
</reports>
```

After executing maven site you'll get a report similar to the one in Figure 4-14.



The screenshot shows a web browser view of a 'Changelog Report'. At the top, it says 'Timeframe: 30 days, Total Commits: 6 Total Number of Files Changed: 65'. Below this is a table with three columns: 'Date', 'Author', and 'File/Message'. There are two rows of data.

Date	Author	File/Message
2005-03-27 11:50:12	Vincent Massol	/code/reporting/core/project.xml v 14 Added Test, Clover and Changelog reports
2005-03-27 11:49:34	Vincent Massol	/code/reporting/core/src/test/mdn/qotd/core/QuoteGeneratorTest.java v 13 Added a failing test to show a test report containing a failure

Figure 4-14. Sample Changelog report showing the last 30 days' worth of SCM commits

The problem with this solution is that commit logs contain lots of information that is not useful for users of your project. Actually, showing all those logs will make it close to impossible for anyone to find out user-related changes. Automatic generation of relevant changes is still a dream!

Thus, you need to hand-edit the changes to make them nice and clean. Maven has a nice Changes plug-in that helps you to do this. To use it, create a *qotd/core/xdocs/changes.xml* file with the following format (this is an example):

```
<?xml version="1.0" encoding="UTF-8"?>

<document>
  <properties>
    <title>Changes</title>
  </properties>
  <body>
    <release version="1.0-SNAPSHOT" date="in SVN">
      <action dev="vmassol" type="update">
        Restricted the changelog report to only show the changes for the
        last week.
      </action>
    </release>
    <release version="0.9" date="2005-03-22" description="Initial release">
      <action dev="vmassol" type="fix" issue="QOTDCORE-1">
        The changelog report now works with Subversion.
      </action>
      <action dev="vmassol" type="add">
```

```
        Initial release. See the features page for more details.
    </action>
</release>
</body>
</document>
```

Each release tag corresponds to a version release and each change is described using the action tag. An action can be an addition (type="add"), a removal (type="delete"), an update (type="update"), or a bug fix (type="fix"). As you can see from the preceding code snippet, with the QOTDCORE-1 issue it's also possible to link an action to an issue URL from your issue tracker. The plug-in computes the URL by evaluating the `maven.changes.issue.template` property which, by default, points to:

```
maven.changes.issue.template = %URL%/ISSUE%
```

The `%URL%` token is the value of the `issueTrackingUrl` tag in *project.xml*, but without the last element in the path. For example, for the core project using Jira as its issue tracker, you have:

```
<project>
  [...]
  <issueTrackingUrl>
    http://www.mavenbook.org/jira/browse/QOTDCORE
  </issueTrackingUrl>
  [...]
</project>
```

The `%ISSUE%` token is the issue number defined in the action tag of *changes.xml* (QOTDCORE-1 here). Thus, the full URL that is computed by the Changes plug-in is:

```
http://www.mavenbook.org/jira/browse/QOTDCORE-1
```

TIP

If you're using a bug tracker that doesn't match the default issue template, you'll need to modify the value of `maven.changes.issue.template`. For example, for Bugzilla you would use:

```
maven.changes.issue.template =
  %URL%/show_bug.cgi?id=%ISSUE%
```

And you would use:

```
maven.changes.issue.template =
  http://sourceforge.net/support/tracker.php?aid=%ISSUE%
```

for the SourceForge tracker.

Add the `maven-changes-plugin` report to your POM reports section and execute `maven site` to generate the report shown in Figure 4-15.

Release History

Version	Date	Description
1.0-SNAPSHOT	in SVN	
0.9	2005-03-22	Initial release

Get the RSS feed of the last changes [RSS](#)

Release 1.0-SNAPSHOT - in SVN

Type	Changes	By
	Restricted the changelog report to only show the changes for the last week.	vmassol

Release 0.9 - 2005-03-22

Type	Changes	By
	The changelog reports now works with Subversion. Fixes QOTDCORE-1	vmassol
	Initial release. See the features page for more details.	vmassol

Figure 4-15. Sample Changes report

One nice additional bonus is that the Changes report generates an RSS feed of the changes. Thus, your project users can add this feed in their favorite Feed Aggregator (SharpReader, FeedDemon, BlogLines, Newz Crawler, etc.), and they'll know right away whenever a change is made!*

What about...

...using issue tracking software for change logs?

This is a very good solution that more and more projects are adopting. You'll need to practice what we (the authors) call Issue Driven Development (IDD) to make it work. IDD goes like this: when a task is done, and just before the code is checked in, ensure that a corresponding issue exists in your issue tracker. If there's no issue for this task, create one (unless the modification is a really minor one that the user should not be concerned with). Then, check in the code mentioning the issue number in the check-in comment, and close/resolve the issue. The result of using IDD is that you can ask your issue tracker to generate a representative change log because all the important changes have an associated issue.

* See Vincent's blog post about Source Code Communication at <http://tinyurl.com/exbsb>.

Publishing Maven Artifacts

The previous labs covered how to add project visibility on quality and progress. Now let's add visibility on deliverables by publishing a project's artifacts. Let's consider the QOTD project. It has several artifacts: a JAR in *qotd/core*, a WAR in *qotd/web*, and a zip in *qotd/packager*. Imagine you want to deploy them to a Maven remote repository.

How do I do that?

Several Maven plug-ins—including the JAR, WAR, EAR, and RAR plug-ins—deploy the artifact they generate. Thus, to deploy a JAR you use `jar:deploy`, to deploy a WAR you use `war:deploy`, to deploy an EAR you use `ear:deploy`, etc.

Under the hood all these deploy goals use the Artifact plug-in's `artifact:deploy` Jelly tag to perform the actual deployment. Thus, to properly deploy an artifact you need to find out how to configure the Artifact plug-in.

Let's practice by deploying the *qotd/core* JAR. The first thing to decide is what deployment protocol you're going to use. The Artifact plug-in supports several deployment protocols: SCP, file copy, FTP, and SFTP (see <http://maven.apache.org/reference/plugins/artifact/protocols.html> for more details).

You are going to use the SCP method to deploy the JAR artifact, as it is one of the most commonly used protocols, and it's secure. You also need to tell the Artifact plug-in where to deploy to. Let's imagine you'd like to publish to *www.mavenbook.org*, for example.

As these properties are true for any subproject in QOTD, add the following Artifact plug-in properties to *common/project.properties*:

```
maven.repo.list = mavenbook  
  
maven.repo.mavenbook = scp://www.mavenbook.org  
maven.repo.mavenbook.directory = /var/www/html/mavenbook/maven
```

Deployment properties are defined for the mavenbook deployment repository using the following syntax: `maven.repo.[repository name].*`, where `[repository name]` is `mavenbook`. `maven.repo.list` is a comma-separated list containing all the repositories to deploy to, and in this case you are publishing to only one remote repository—`mavenbook`. The `maven.repo.mavenbook` property defines both the protocol to use (SCP here) and the deployment host. `maven.repo.mavenbook.directory` specifies the deployment directory on the host server.

Unfortunately, you won't be able to publish to mavenbook.org. If you want to try this, you'll need your own SSH server. Sorry!

You also need to specify deployment credentials. It's best to define those in a *build.properties* file, as this file is not meant to be checked in your SCM and you want your password to remain secret. If your deployment server uses username/password authentication, you'll define:

```
maven.repo.mavenbook.username = vmassol  
maven.repo.mavenbook.password = somepassword
```

TIP

Simply define the following properties:

```
maven.repo.mavenbook = ftp://www.mavenbook.org  
maven.repo.mavenbook.directory = /var/www/html/mavenbook/maven  
maven.repo.mavenbook.username = vmassol  
maven.repo.mavenbook.password = somepassword
```

to publish using FTP.

If your SSH server supports private key authentication, you can use the `maven.repo.mavenbook.privatekey` and `maven.repo.mavenbook.passphrase` properties instead of a password. A more secure approach is to configure an authentication key for SSH in a user account on the machine you want to deploy from.

TIP

The Artifact plug-in uses the JSch framework (<http://www.jcraft.com/jsch/>) under the hood for supporting the SSH protocol. You'll get the following kind of stack trace error:

```
com.jcraft.jsch.JSchException: Auth fail  
    at com.jcraft.jsch.Session.connect(Unknown Source)  
    at org.apache.maven.deploy.deployers.  
    GenericSshDeployer...
```

if you have not configured authentication correctly.

Publish the core JAR artifact:

```
C:\dev\mavenbook\code\reporting\core>maven jar:deploy  
[...]  
jar:deploy:  
    [echo] maven.repo.list is set - using artifact deploy mode  
Will deploy to 1 repository(ies): mavenbook  
Deploying to repository: mavenbook  
Deploying: C:\dev\mavenbook\code\reporting\core\project.xml-->mdn/poms/qotd-  
core-1.0.pom  
Executing command: mkdir -p /var/www/html/mavenbook/maven/mdn/poms
```

```
Executing command: chmod g+w /var/www/html/mavenbook/maven/mdn/poms/qotd-
core-1.0.pom

Deploying: C:\dev\mavenbook\code\reporting\core\project.xml.md5-->mdn/poms/
qotd-core-1.0.pom.md5
Executing command: mkdir -p /var/www/html/mavenbook/maven/mdn/poms

Executing command: chmod g+w /var/www/html/mavenbook/maven/mdn/poms/qotd-
core-1.0.pom.md5

Will deploy to 1 repository(ies): mavenbook
Deploying to repository: mavenbook
Deploying: C:\dev\mavenbook\code\reporting\core\target\qotd-core-1.0.jar-->
mdn/jars/qotd-core-1.0.jar
Executing command: mkdir -p /var/www/html/mavenbook/maven/mdn/jars

Executing command: chmod g+w /var/www/html/mavenbook/maven/mdn/jars/qotd-
core-1.0.jar

Deploying: C:\dev\mavenbook\code\reporting\core\target\qotd-core-1.0.jar.
md5-->mdn/jars/qotd-core-1.0.jar.md5
Executing command: mkdir -p /var/www/html/mavenbook/maven/mdn/jars

Executing command: chmod g+w /var/www/html/mavenbook/maven/mdn/jars/qotd-
core-1.0.jar.md5
```

BUILD SUCCESSFUL

What just happened?

The `artifact:deploy` tag is executing commands on the remote machine using SSH. The structure of the repository is being created, and all directories and files are made group-writable with `chmod`. As you can see from the console, the `artifact:deploy` tag has deployed not only the core JAR, but also the core project's POM. This is because the POM is the identity of a Maven project and it may be useful for a user browsing the repository to know more about the project producing the artifacts he's looking for. In practice the POMs will also enable Maven 2 to support *transitive dependencies*. This means that in the future you'll be able to specify only the direct dependencies your project is depending upon, and Maven will auto-discover the dependencies of your dependencies.

Because you can never be too security conscious, the `artifact:deploy` tag creates MD5 signatures for every deployed artifact. Maven currently does not use them when downloading artifacts, but they'll certainly be implemented in the future.

Transitive dependencies are going to be a huge timesaver.

What about...

...publishing the packager's zip file?

Publishing the JAR was easy because there's an existing `jar:deploy` goal. However, there's no zip plug-in, and thus no `zip:deploy` goal! The solution is to write a custom goal in your *maven.xml* file. Edit the *qotd/packager/maven.xml* file and add the code in bold:

```
<?xml version="1.0"?>

<project default="qotd:build"
  xmlns:ant="jelly:ant"
  xmlns:artifact="artifact">

  <goal name="qotd:build">
    <ant:mkdir dir="{maven.build.dir}"/>
    <ant:zip destfile=
      "${maven.build.dir}/${pom.artifactId}-${pom.currentVersion}.zip">
      <ant:fileset file="{pom.getDependencyPath('mdn:qotd-web')}/>
      <ant:fileset dir="{maven.src.dir}/conf"/>
    </ant:zip>
  </goal>

  <goal name="qotd:deploy" prereqs="qotd:build">
    <artifact:deploy
      artifact="{maven.build.dir}/${pom.artifactId}-${pom.currentVersion}.zip"
      type="zip"
      project="{pom}"/>
  </goal>

</project>
```

The `qotd:build` goal creates the QOTD zip (see Chapter 3 for a refresher). You've now added a `qotd:deploy` goal that uses the `artifact:deploy` Jelly tag to deploy the zip file, and then passed in the `artifact` attribute. The `type` attribute corresponds to the artifact extension that is used to decide in which directory to put the artifact in the remote repository. You need to pass a reference to the current project's POM in the `project` attribute. Running `qotd:deploy` deploys the zip to the remote repository, in `[REMOTE_REPO_ROOT]/mdn/zips/qotd-packager-1.0.zip`, following the standard repository mapping rule, `[REPO_ROOT]/<groupId>/<type>s/<artifactId>-<currentVersion>.<type>`, discussed in Chapter 1.

Announcing a Project Release

You have deployed your artifacts to a Maven remote repository. Now you need to announce this fact to your project stakeholders.

How do I do that?

Use the Announcement plug-in. If you've been following the practice of using a *changes.xml* file to describe your project changes, as demonstrated earlier, this will be very easy. If not, you're out of luck, as the Announcement plug-in works hand in hand with the Changes plug-in.

In this lab you'll generate a release announcement for the *qotd/core* subproject, which will allow you to reuse the *changes.xml* file you added in a previous lab.

Decide for what release you want to generate the announcement. If you don't configure anything, an announcement for the current version defined in the POM will be generated. You can control this through the `maven.announcement.version` property, which defaults to:

```
maven.announcement.version = ${pom.currentVersion}
```

The Announcement plug-in makes one check: it verifies that there is a `version` tag in the POM that matches the release for which you wish to generate the announcement. A `version` tag indicates that a release has been made, and it's also meant to link the release with the `SCM` tag that you've used. It's indeed good practice to tag your `SCM` when performing a software release. Let's imagine that you've already released version 0.9 of the *core* project. Add the following version information to *qotd/core/project.xml*:

```
[...]
</repository>

<versions>
  <version>
    <id>0.9</id>
    <name>0.9</name>
    <tag>QOTD_CORE_0_9</tag>
  </version>
</versions>
[...]
```

Note that the `name` element is simply a friendly name for the `id` tag. It can be whatever you like.

Generate the announcement for release 0.9 by running `maven announcement:generate` (note that we're passing a property on the command line):

```
C:\dev\mavenbook\code\reporting\core>maven announcement:generate ^
More? -Dmaven.announcement.version=0.9
[...]
announcement:generate:
```

```
[echo] Generating announcement for release 0.9 in C:\dev\mavenbook\code\
reporting\core\target\generated-xdocs\announcements\
announcement-0.9.txt...
```

```
[echo] Using stylesheet located at file:C:\Documents and Settings\
Vincent Massol\.maven\cache\maven-announcement-plugin-1.3\pl
ugin-resources\announcement.jsl and UTF-8 encoding
BUILD SUCCESSFUL
```

You can find the generated announcement in *target/generated-xdocs/announcements/announcement-0.9.txt*:

The mdn team is pleased to announce the QOTD Core 0.9 release!

<http://www.mavenbook.org/projects/mdn/qotd-core>

QOTD Core library

Changes in this version include:

New Features:

- o Initial release. See the features page for more details.

Fixed bugs:

- o The changelog report now works with Subversion. Issue: QOTDCORE-1.

Have fun!

-The mdn team

As you can see, the Announcement plug-in generated this text by gathering information from the POM and from the *core/changes.xml* file:

```
[...]
<release version="0.9" date="2005-03-22" description="Initial release">
  <action dev="vmassol" type="fix" issue="QOTDCORE-1">
    The changelog reports now works with Subversion.
  </action>
  <action dev="vmassol" type="add">
    Initial release. See the features page for more details.
  </action>
</release>
</body>
</document>
```

What about...

...sending this announcement by email?

You can do that using the `announcement:mail` goal. At a minimum you'll need to tell the plug-in what SMTP server to use and to whom to send the email. You specify the SMTP server by adding the following property

in your *build.properties* file (this is indeed a property that depends on your environment and may or may not be shared with others):

```
maven.announcement.mail.server = my_smtp_server
```

The recipients are defined using the `maven.announcement.mail.to` property. For example:

```
maven.announcement.mail.to = users@mavenbook.org,dev@mavenbook.org
```

The plug-in will compute the email sender by looking at the developers section of the POM and will use the email of the first developer it finds, unless you have specified a sender using the `maven.announcement.mail.from` property. Several other optional properties exist; you can find them online at <http://maven.apache.org/reference/plugins/announcement/properties.html>.

Reporting Project Releases

You have published project releases. Now would be a good time to discover how to let your users know where to find them. Let's add a release report to the project web site.

How do I do that?

The XDoc plug-in has a nice feature: it can generate a download report listing all past releases, with a link to the artifacts and the release announcements. Let's use it on the QOTD packager project. The packager is the project that generates the main QOTD distributable, so it makes sense to have it generate the download report.

First, tell the XDoc plug-in where the artifacts are published. You do this by using the `maven.xdoc.distributionUrl` property. Add it to the *qotd/packager/project.properties* file:

```
maven.xdoc.distributionUrl=http://www.mavenbook.org/maven/mdn/zips/
```

The distribution URL is an important piece of information missing from the POM. It will be added in the near future. The XDoc plug-in needs information about the releases that have been made. It gets this information from the POM by scanning the version tags. Let's imagine that you've already released versions 0.7, 0.8, and 0.9 of the packager project. Add the following version information to *qotd/packager/project.xml*:

```
[...]  
</repository>
```

```

<versions>
  <version>
    <id>0.7</id>
    <name>0.7</name>
    <tag>QOTD_PACKAGER_0_7</tag>
  </version>
  <version>
    <id>0.8</id>
    <name>0.8</name>
    <tag>QOTD_PACKAGER_0_8</tag>
  </version>
  <version>
    <id>0.9</id>
    <name>0.9</name>
    <tag>QOTD_PACKAGER_0_9</tag>
  </version>
</versions>
[...]
```

You saw in the previous lab that the Announcement plug-in can generate announcement reports. It would be nice to add them to the download report so that users who want to download your software can know what the download contains. The download report will automatically add the announcement reports if the XDoc plug-in can find a *changes.xml* file with releases matching the version tags of the POM. Add one to *qotd/packager/xdocs/changes.xml*:

```

<?xml version="1.0" encoding="UTF-8"?>

<document>
  <properties>
    <title>Changes</title>
  </properties>
  <body>
    <release version="1.0-SNAPSHOT" date="in SVN">
      [...]
    </release>
    <release version="0.9" date="2005-03-22">
      [...]
    </release>
    <release version="0.8" date="2005-03-10">
      [...]
    </release>
    <release version="0.7" date="2005-03-01">
      [...]
    </release>
  </body>
</document>
```

You're all set! Reap the fruits of your hard labor by typing `maven site`. It'll generate a download report, as shown in Figure 4-16.

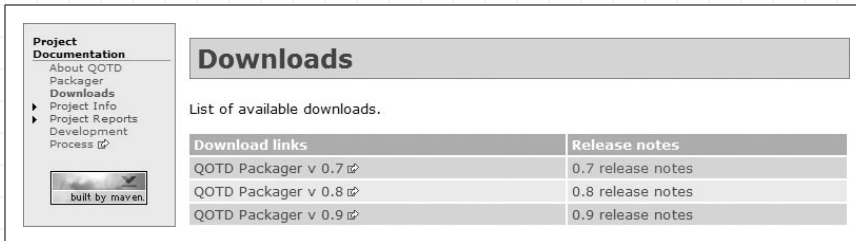


Figure 4-16. Download report showing QOTD packager’s releases, including release notes

Note that there is a visible Downloads link in the left menu that draws the attention of visitors.

What about...

...having a Downloads link on the master QOTD web site?

Yes, you’re right, the Downloads link in the packager project is nice, but it isn’t very visible for visitors who will arrive on the top-level site. The solution is to write a *qotd/xdocs/downloads.xml* XDoc document linking to the different subprojects’ download reports. Indeed, it’s possible that several subprojects produce distributable artifacts and this solution allows you to list them all on the main *downloads.xml* page.

Publishing a Project Web Site

You have generated lots of reports in this chapter. These reports will not do any good if they’re not made available by publishing them. Let’s remedy this!

How do I do that?

Publishing to a single project web site is different from publishing to a multiproject web site only in terms of the goal to call: you call `multiproject:site-deploy` for multiprojects and `site:deploy` for single projects. In practice, the `multiproject:site-deploy` goal simply starts by calling the `multiproject:site` goal that we saw in Chapter 3, and then it calls `site:deploy` on it to upload the site to the remote server.

TIP

At the time of this writing the Site plug-in does not use the Artifact plug-in to deploy the web site as an artifact. This will certainly happen in future versions of Maven, and at that time you'll be able to practice your newly acquired knowledge of publishing project artifacts.

The Site plug-in supports four protocols for deploying a web site: File System (fs), SSH (ssh), FTP (ftp), and RSync (rsync). SSH is the default one, but if you wish to choose another one, define the `maven.site.deploy.method` property in your properties file (either *project.properties* or *build.properties*, depending on whether you wish to share this property with other team members):

```
maven.site.deploy.method = ssh
```

Let's publish the QOTD web site to <http://www.mavenbook.org/mdn/qotd> using SSH.

To use the SSH method you need to have SSH and SCP clients installed, and the `maven.ssh.executable` and `maven.scp.executable` properties must point to them. For example, if you're on Windows and have installed PuTTY's Plink and Pscp (<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>) and they're in your PATH, you would write the following in your *build.properties* file:

```
maven.ssh.executable = plink
maven.scp.executable = pscp
```

You also need to specify the username under which you wish to deploy the site. Specify it in your *build.properties* file. For example:

```
maven.username = vmassol
```

TIP

Most of the problems you will have when publishing a web site using SSH will be due to authentication. It's strongly recommended to use a public/private key scheme. If you don't use one, your SSH executable will want to prompt you to enter the password but Maven will not propagate this prompt. It'll thus appear to hang. You should always use your SSH client first to ensure you can connect to the remote host automatically, without having to type anything.

Now you need to tell the Site plug-in where to deploy to by defining the `siteAddress` and `siteDirectory` elements in the QOTD POM. As these

properties are shared by all subprojects, define them in *qotd/common/project.xml*:

```
[...]
<url>http://www.mavenbook.org/projects/${pom.groupId}/${pom.artifactId}</url>

<siteAddress>www.mavenbook.org</siteAddress>
<siteDirectory>
  /var/www/html/mavenbook/projects/${pom.groupId}/${pom.artifactId}
</siteDirectory>
[...]
```

Publish the QOTD web site by typing `multiproject:site-deploy`:

```
C:\dev\mavenbook\code\reporting>maven multiproject:site-deploy
[...]
site:local-deploy-init:
[echo]
  site clean = false
  siteDirectory = /var/www/html/mavenbook/projects/mdn/qotd

site:remote-deploy-init:
[echo]
  siteAddress = www.mavenbook.org
  siteUsername = vmassol

site:sshdeploy:
[tar] Building tar: C:\dev\mavenbook\code\reporting\target\qotd-1.0-
site.tar
[gzip] Building: C:\dev\mavenbook\code\reporting\target\qotd-1.0-site.
tar.gz
[delete] Deleting: C:\dev\mavenbook\code\reporting\target\qotd-1.0-site.
tar
[...]
```

Open a browser and point it to <http://www.mavenbook.org/mdn/qotd>.
Enjoy!

TIP

If you have made a configuration mistake and you do not wish to regenerate the full web site (which `multiproject:site-deploy` does), simply run `site:sshdeploy` in *qotd/*.

What about...

...using another deployment method?

Refer to the Site plug-in reference documentation at <http://maven.apache.org/reference/plugins/site/>. For example, to use the FTP protocol

you would use the same properties as defined earlier, and simply tell the `site:deploy` goal that you're using FTP by setting the following:

```
maven.site.deploy.method = ssh
```