Public/Project Versioning - Best Practices

From TLC Projects Wiki

Contents

Overview

This page will try to explain one particular process that can be used to version your projects, as a developer. While the process covered here will use one example of how to accomplish effective versioning, the concepts can be used anywhere.

Why do you need to version your project? The answer is multifaceted. As a developer you typically have several deliverables. These can be for either internal use or external use, depending on the nature of the project. It really doesn't matter which. You really need to version all of these deliverables. While you can often get away with ignoring versioning internally, it will eventually catch up with you. For external projects, it has an even greater importance.

Deliverables include the project itself, which may be a web application intended for deployment on a server somewhere. It could also be a desktop application. Along with that, you may need to deliver documentation, source code, and potentially more artifacts. It might just be a library for use in multiple projects. You have all of these

files assembled on your machine, and hopefully all in a source control management system. You've also got a website which allows people to download these artifacts as well. You also have an issue management system where people can report bugs. And yet while you've released one version and all of its artifacts to the world, you're still working to improve your product. Maybe you're even deploying incremental improvements as you make them. How do you keep this all synchronized? How do you make sure that you're using the same version that a client is using, so that you can verify their reports? What if you don't have a versioning system, but just a download page, where people just download the documentation as it appears? Is their documentation the same as your updated documentation? Is their problem just because the documentation they downloaded isn't the latest documentation? What if the libraries your project uses forges ahead with new versions, but these break your project? How do you keep your project going while also giving yourself a chance to keep up to date with the latest and greatest libraries available? What if new developers come along and want to work on your project, or need to fix a problem in the deployed version of your project. Is the latest version from SVN the same as the deployed version? If not, what revision do you use? What dependencies does it work properly with? All of these issues, and many more can be minimized by practising sound versioning technique.

Steps in the process

1. Standard directory structure
2. Prepare and release a version of your product
3. Tag the release.
4. Make a branch from the trunk.
5. Work on the branch to fix bugs in the released version.
6. Work on the trunk to add new features.
7. Merge fixes from the branch into the trunk as required.
8. Release the branch as minor dot increments of your product.
9. Release the trunk as major dot revisions of your product.

The Process

It's probably best to go through an example. I will use maven as a build system and subversion as our scm. Imagine that you have a web application that you have been working on, and you're now ready to release it to the world. In maven, where the only addition to a regular set of files is a project object model, which is an XML file which describes your project (pom.xml), you would probably have the version set at 1.0-SNAPSHOT. As you made builds with maven (it could very well have been ant, or anything else), you would have been creating artifacts such as pmgt-1.0-SNAPSHOT.war. You would have been deploying these to your test server, not worrying particularly about the version, because this is identified as a snapshot, which implies that it relects the latest state of your code. But now you're ready to make it version 1.0.

At the same time, you have a big wishlist, which you would like to work on for version 2. You also have some known bugs and some minor features that you would like to

release as version 1.1, but you need to get the working version 1.0 out there.

Requirements

To follow along, you will need to download maven (http://maven.apache.org) and install it. It's a command line tool, though it is integrated into most IDE's as well. We will stick to the command line. You will also need to have access to a subversion repository, a server capable of sftp, as well as a java compiler. We will use J2SE 1.4.

Standard directory structure.

One thing that varies wildly among developers is the directory structure of a project. Where does the source code go? HTML? Images? Distributions? Dependencies? And so on. It's often wrapped together with an build script. Over the years, developers have converged towards a fairly standard structure. The advantage of a standard structure is that new developers don't need to go and figure out where everything is, when joining a project, or how to build it, or where to put new files that they create. Plus they tend not to create new folders (other than subfolders), because there is already a place for everything. The standard directory structure we'll be using is documented here:

Standard Directory Structure (http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html)

Now in addition to the structure of the project, it is prudent to accomodate your source control management system. For subversion, we need to have a trunk, a branches and a tags folder.

Prepare and release a version of your product.

Create a Project

You can create a simple project very quickly using maven, that adheres to the standard directory structure.

```
mvn archetype:create -DarchetypeGroupId=<archetype-groupId> -DarchetypeArtifactId=<archetype-artifac
 -DarchetypeVersion=<archetype-version> -DgroupId=<my.groupid> -DartifactId=<my-artifactId>
```

How about an example - that looks horrible. You don't need to specify archetype parameters unless you want to use the non-default archetype, which is just a quick-start application.

```
mvn archetype:create -DgroupId=ca.ucalgary.commons -DartifactId=my-project
```

Now in that same folder, make branches, trunk and tags folders. Then put the pom.xml and the src folder into the trunk. You should have a folder structure similar to that shown below.

```
my-project
|-- trunk
|  |-- pom.xml
|  `-- src
|      |-- main
|      |   |-- java
|      |   |  `-- ca
|      |   |     `-- ucalgary
|      |   |        `-- commons
|      |   |           `-- App.java
|      |   `-- resources
|      |-- test
|      |   |-- java
|      |   |  `-- ca
|      |   |     `-- ucalgary
|      |   |        `-- commons
|      |   |           `-- AppTest.java
|      |   `-- resources
|      |      `-- users.xml
|      `-- site
|          |-- site.xml
|          |-- changes.xml
|          |-- apt
|          |  `-- intro.apt
|          |-- fml
|          |  `-- faq.fml
|          `-- resources
|              |-- css
|              |  `-- site.css
|              |-- documents
|              |  `-- srs.doc
|              `-- images
|                 `-- logo.png
|-- tags
`-- branches
```

Source Control

Now you need to get your project into Subversion (http://wiki.ucalgary.ca/page/LearningCommons/Documentation/Subversion) , to track changes, to give you a back up, and to let others share in the work. You will also need to let maven know about your scm. You do this by modifying the project object model, or POM, which is contained in pom.xml. So add the following scm elements to the project element in your POM, changing them as required for your scm. You also need to let maven know about the tags area of your project, which is specified in the build area of your pom. It is good practice to make sure that your artifactId and your root folder in scm have the same name. That way, you can just use ${project.artifactId} to keep things consistent.

```
  <scm>
    <connection>scm:svn:http://apollo.ucalgary.ca:8800/${project.artifactId}/trunk</connection>
    <developerConnection>scm:svn:http://woodj@apollo.ucalgary.ca:8800/${project.artifactId}/trunk</de
    <url>http://apollo.ucalgary.ca/websvncommons/listing.php?repname=${project.artifactId}&rev=0&sc=0
  </scm>
```

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-release-plugin</artifactId>
      <configuration>
        <tagBase>http://woodj@apollo.ucalgary.ca:8800/${project.artifactId}/tags</tagBase>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Prepare a Release

Now let's assume you have been writing code, creating unit tests and adding resources. You will of course be wanting to make interim releases, either for your own purposes or to share with your colleagues, or maybe even just to develop your release process. How do you version these releases? Maybe you do 10 releases a day at the height of development - do they all need their own version numbers? In most cases, the answer is no. These are still pre-release releases, for internal use, and thus don't need to be versioned formally. One way to keep track is to version all these releases as 1.0-SNAPSHOT, or something similar. The snapshot simply indicates that it is the latest state of your code. If you look in the pom.xml of our example project, you'll see that the version is already set to 1.0-SNAPSHOT.

Now let's say you're ready to release version 1. You already have tons of ideas for version 2, you have some bugs to take care of, but they can wait until version 1.0.1. You might even want to make an interim feature release at version 1.1. How do you accomodate this?

First let's release version 1. To do that you'll first want to make sure all of your files are committed to your SCM. Then we'll create a tagged version in scm, from which we'll do a build and all our tests. Finally when all is good, we'll make that build and all of it's artifacts available to the world for download from a website.

To release your project using maven, which automates the process somewhat, issue this command in your trunk:

```
mvn install release:prepare
```

First maven will install all your artefacts into your local repository, which will be required during the release process. Then maven will check to make sure your scm is up to date. If it's not, the maven command will fail and tell you to commit any files that have been changed. When this part of the process is satisfied, maven will build the project and run any unit tests. Again, if any of this fails, you need to go fix it and run the command again. Finally, when all of the preceding works satisfactorily, maven will ask you for the release version. The default will be 1.0, which it obtains from the specified release version (in the POM) minus the SNAPSHOT suffix. Usually the defaults will suffice. Then you will be asked for the tag name. The tag name will be the name of the directory which goes under the tags folder in scm. It represents the source code

that was used to build this version, and will always be available, no matter what further development is performed. Something like my-project-1.0 is a good tag name. Maven will then ask you for the next development release version, which in our case will be 2.0-SNAPSHOT.

At this point, maven has actually committed an entire version of your project back into scm, under the tags directory, with the tag name specified during prepare. You can update your working copy to see the changes. Maven has also updated the pom in this tagged version so that it uses the new development versions you specified during prepare. Maven has also created a release.properties file. Maven documentation has this to say:

The release.properties file is created while preparing the release. After performing the release the file remains within the project root directory until the maven user deletes it. The release.properties file can be given to any developer within the team and by simply excuting the release:perform goal can create and deploy a new instance of the project artifact time and again.

But what do you actually do with this file? Commit it into the head? Delete it? Remember, we want the tagged release to be exactly what was used to create the deployment, so I'm not sure it belongs in scm. I keep a folder, locally, full of my release.properties files. That sort of gives you a quick idea of what has been released and when, plus if you need to, you can re-release a project quite easily. There is no other way to release the version of the project specified by this release.properties file, once it is gone! You can check out the tagged version of the project, which is the same as the released version, but if you try to release that tagged version, you should release to a point version.

Perform a Release

Now to actually release the project, you need to do a few more things to facilitate deployment. We want the artifacts to go to a server, so we are going to let maven use sftp to connect to that server. We do this by adding a distributionManagement section to your pom.

```
<distributionManagement>
  <repository>
    <id>commons.ucalgary.ca</id>
    <name>Learning Commons Server</name>
    <url>sftp://commons.ucalgary.ca/Library/WebServer/Documents/pub/m2</url>
  </repository>
</distributionManagement>
```

Then, you need to put passwords into a file outside of version control. The easiest is to make a settings.xml file in your .m2 directory in your home folder (~/.m2/settings.xml). It should look something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<settings>
  <servers>
    <server>
      <id>commons.ucalgary.ca</id>
      <username>woodj</username>
      <privateKey>~/.ssh/id_rsa</privateKey>
      <passphrase>***</passphrase>
      <password>***</password>
    </server>
  </servers>
</settings>
```

Now you can issue the release command to maven:

```
mvn release:perform
```

Now maven checks out the tagged version of your project from scm, builds it, tests it, and creates artifacts for distribution. These artifacts are placed onto the server, along with a developer website which wraps all of this information together into one place for easy perusal.

Reverting a Release

If the release process fails, most likely because something in your dev environment is different than what you get when you freshly check out your project from scm, you'll need to fix the problem and do the prepare over again. This is a key feature to maven builds - it makes sure that the project you have in scm has everything it needs to build and test, which is critical to reproducibility and to project sharing. Maven has likely already modified your pom.xml and committed it, or has left it in some in between state. Maven wants to change your pom so that it reflects the new location under the tags folder in scm, and that this is a 1.0 release version, as well as create a new pom which reflects the fact that the trunk is now version 2.0, for instance. To revert, you will need to:

1. delete release.properties in your trunk
2. change to your top level directory under scm control
3. svn update
4. svn delete tags/my-project-1.0
5. change the version in trunk/pom.xml from 1.1-SNAPSHOT to 1.0-SNAPSHOT
6. make sure the scm element paths in trunk/pom.xml reflect the trunk of your project (the way it was to begin with), not the failed tag
7. svn commit -m "Redoing version 1 release"
8. fix the problem
9. svn commit -m "The fixes"
10. change to your trunk directory
11. mvn release:prepare
12. mvn release:perform

Now you can continue to work on your code in the trunk, committing to scm as always,

knowing that you can easily get back to a 1.0 version of the code if necessary. In the next section, we'll see how to deal with version 1.0.1 and version 1.1. You *don't* want to work on the code in the tagged release.

As an aside, you will notice that maven will have placed artifacts on the server you specified in distributionManagement. These include a binary release, a source release, and the javadocs. Each release is thus characterized by a synchronized set of artifacts, as well as source code in the scm that can be immediately analyzed, regardless of the state of your code in the trunk.

Make a branch from the trunk.

Any time you want to divert away from your main line of code, you create what is usually called a branch. Your main line of code, of course, is the trunk. Each released version of code is a tag, hence the folders we created at the beginning of this document. It is good practice to create a sub version right after you make a release, so that you can make dot releases (eg 1.0.1) while your major work proceeds in the trunk of a larger increment (eg 2.0-SNAPSHOT).

To make a branch in subversion, you can issue this command (while at the top level directory):

```
>svn copy trunk branches/my-project-1.0.1
>svn commit -m "Creating the 1.0.1 branch of my-project"
```

You can do the same for version 1.1. Now each of these 2 branches and the trunk represent their own independent line. They are, of course, now in your branches directory. To accommodate maven, you would want to edit the version element in the pom such that it was 1.0.1-SNAPSHOT or 1.1-SNAPSHOT, instead of the 2.0-SNAPSHOT we specified previously. You also need to fix the scm element:

```
<scm>
  <connection>scm:svn:http://apollo.ucalgary.ca:8800/my-project/branches/my-project-1.0.1</connectio
  <developerConnection>scm:svn:http://woodj@apollo.ucalgary.ca:8800/my-project/branches/my-project-1
  <url>http://apollo.ucalgary.ca/websvncommons/listing.php?repname=my-project&rev=0&sc=0&path=/branc
</scm>
```

These changes need to be committed into SCM.

You can release versions from these branches as described previously, into the tags directory. This also allows you to move off bugs and feature requests in your issue management system to specific versions. After releasing from a branch, you would probably want to do an svn move, to move your branch to the next release, and again, fix the scm element as required.

```
>cd branches
>svn mv my-project-1.0.1 my-project-1.0.2
>svn commit -m "Changed branch from 1.0.1 to 1.0.2 (1.0.1 released)" my-project-1.0.2
>svn commit -m "Released 1.0.1"
```

You can merge desirable changes from the 1.1 or 1.0.1 branches back into the trunk, or vice-versa. The key is to know at which revision the branch was made. Let's say you changed pom.xml in the trunk at revision 150 and you want those changes merged into the 1.0.1 branch, which was made at revision 145.

```
>cd branches/my-project-1.0.1
>svn diff -r 145:150 ../../trunk/pom.xml
>svn merge -r 145:150 ../../trunk/pom.xml
```

Note that you're changing to the directory in the branches where the file in question resides. If you want to do an entire directory, the process is the same. If you want to merge changes from a branch back into the trunk:

```
>cd branches/my-project-1.0.1
>svn diff -r 150:145 .
>svn merge -r 150:145 . ../../trunk
```

Think of it as calculating the changes from when the branch was made up until now, and then applying those changes to the trunk. Note that if you have made changes in a branch, and in the trunk, on the same file, these commands won't show that merge to you. More than likely, when you merge, you will get a conflict, and you'll need to go resolve it manually.

Refer to the svn redbook (http://svnbook.red-bean.com/en/1.0/ch04.html) for more information on how to do this with subversion.

If you have more complex bugs then repository versions might not be enough and you will want to tag both before and after working on the bug in order to obtain the changeset that can be applied to other branches. See page 116 of Pragmatic Version Control Using Subversion by Mike Mason ((http://www.amazon.com/gp/product/0974514063/ref=ed_oe_p/103-6269884-3551010?%5Fencoding=UTF8) ISBN 0-974510-6-3) (http://www.amazon.com/gp/product/0974514063/ref=ed_oe_p/103-6269884-3551010?%5Fencoding=UTF8) for the complete example.

Snapshots

Sometimes you need to deploy versions of your software that can be used by others during development, but you are not ready for a release. You don't want to go through the hassle of making a point release for every small change, or even a bundle of changes. This is solved by deploying a version referred to as snapshot release of your code. They are a snapshot of say, the 1.1 development branch of your project. As such, it is typically versioned 1.1-SNAPSHOT. Now there can be many 1.1-SNAPSHOT releases, which you also need to keep track of, if you want to be able to match issues with code. To do that, we need to include a build number, which we will touch on

later. But for now, how do we deploy snapshots so they can be used?

You will first need to make a snapshot repository. It is just like your regular repository, and in fact can be your regular repository if desired, but most people prefer to keep released artifacts separate from snapshots. You will refer to it like thus:

```
<distributionManagement>
  ...
  <snapshotRepository>
    <id>tlc-snapshots</id>
    <name>TLC Snapshot Development Repository</name>
    <url>sftp://commons.ucalgary.ca/Library/WebServer/Documents/pub/m2-snapshots</url>
  </snapshotRepository>
</distributionManagement>
```

Again, make sure you put your credentials in settings.xml for your snapshot repository, just as you did for your artifact repository.

Now, instead of releasing your snapshot, you will deploy it.

```
mvn install deploy:deploy
```

You need to make sure the snapshot is installed before deploying it. If you know it is installed, then you can omit the install step from the above command. Deploying will place the artifact in your snapshot repository.

To use it as a dependency, you refer to it as you would any other dependency, but make sure you specify the snapshot repository in your pom.

```
<repositories>
  ...
  <repository>
    <id>snapshots</id>
    <name>Maven Snapshot Development Repository</name>
    <url>http://snapshots.maven.codehaus.org/maven2</url>
  </repository>
</repositories>

<dependencies>
  <dependency>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-release-plugin</artifactId>
    <version>2.0-beta-4-SNAPSHOT</version>
  </dependency>
</dependencies>
```

Build Versioning

Especially for snapshots, you might want to keep better track of what build makes it into the release. So you have deployed version 1.1-SNAPSHOT several dozen times. Which is which? Even for a released version, what build made up the release? Of course, you can tell by looking in the log of your subversion repository when you made

the release, and use the revision number as a build number. You could also use a timestamp, or you could use a sequence. In each case, you want to make these build numbers easily available, so that they can be referred to, for instance, when a bug report is filed. The build number could be in the filename, or in some metadata, or in a properties file. There are advantages to each. Let's deal with using the revision number from svn as a build number, and let's make sure that number goes into a metadata file, which is easily retrieved and displayed by the resultant application.

To facilitate our build versioning, we are going to use a build number plugin for maven (http://commons.ucalgary.ca/projects/maven-buildnumber-plugin/index.html) which retrieves the current revision from svn, and allows us to use it in our pom. We are also assuming that we are packaging a war, with java code inside.

First, add the repository to your pom:

```
<pluginRepositories>
   <pluginRepository>
       <id>tlc</id>
       <name>TLC Repository</name>
       <url>http://commons.ucalgary.ca/pub/m2</url>
   </pluginRepository>
</pluginRepositories>
```

Then add the plugin:

```
<build>
  ...
  <plugins>
    ...
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>maven-buildnumber-plugin</artifactId>
      <version>0.9.4</version>
      <executions>
        <execution>
          <phase>validate</phase>
          <goals>
            <goal>create</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

And finally, add the resultant build number to your war manifest:

```
<build>
  ...
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <version>2.0</version>
      <configuration>
        <archive>
          <manifestEntries>
            <Implementation-Build>${buildNumber}</Implementation-Build>
          </manifestEntries>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Finally to refer to this in code, you just need to read in the manifest file. Something like this would do:

```
String appServerHome = getServletContext().getRealPath("/");

File manifestFile = new File(appServerHome, "META-INF/MANIFEST.MF");

Manifest mf = new Manifest();
mf.read(new FileInputStream(manifestFile));

Attributes atts = mf.getMainAttributes();

System.out.println("Version: " + atts.getValue("Implementation-Version"));
System.out.println("Build: " + atts.getValue("Implementation-Build"));
```

Creating a Developer Site

Besides versioning your project and making artifacts available on your server for other projects to use, it is a good idea to make a developer web site. This site is intended to pull all parts of the project together for easy perusal. Items such as Functional Requirements, Software Requirements Specifications and Design documents can go here. You can also interactively link with your SCM and your Issue Management System, provide change histories to accompany versions, produce javadocs, report on developer activity, list project members, provide downloads, and include just about anything that might have something to do with your project.

Maven can help out an awful lot in the production of this website. Because some of the reports interact with SCM, the best time to create a site is *just before you are ready to release it*. If one were to create the site from the tagged release, or from a branch, or even from the target/checkout produced by the release, all of the SCM information would be missing.

To create a site, simply switch to your trunk directory and issue the site command.

```
mvn site
```

As you are experimenting with the site, you might want to turn off tests and stop any plugins from working on the internet:

```
mvn site -Dmaven.test.skip=true -o
```

The tutorial on the maven site (http://maven.apache.org/guides/mini/guide-site.html) is a good place to learn how to enable reports and write files in APT or FML.

Definitions and Abbreviations

archetype
: a template for a project, consisting of default directories and files

artifact
: the product of a particular build (eg. pmgt-1.0-src.tar.gz)

snapshot
: a version of your product that reflects the latest state of your source code

build
: the act of creating an artifact, which may include compiling of source files, copying in resource files, and packaging them up

subversion (svn)
: a source control managements system (see CVS as well)

cvs
: concurrent versioning system - a source control management system

scm
: source control management

dependency
: a file required for the artifact to build, run or be tested. (eg. commons-lang-1.0.jar)

POM
: project object model - pom.xml; describes your project, it's dependencies, it's scm, and most importantly, how to build it

Retrieved from "http://apollo.ucalgary.ca/tlcprojectswiki/index.php/Public/Project_Versioning_-_Best_Practices"

Categories: Documentation | Public
