Maven

Steven J Zeil

March 17, 2013


Outline


Outline I


Maven

Another Apache project, *Maven* came well after **Ant** had come to dominate the Java open source landscape.

- Initially seen as a comperitor or replacement for Ant
- Maven addresses both
  - build management (as does Ant)
  - and configuration management (which Ant does not)
    - Later, we'll talk about Ivy, which adds configuration mgmt to Ant


Motivations for Maven

Grew out of an observation that many supposedly cooperative, related Apache projects had inconsistent and incompatible **ant** build structures.
Stated goals are

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices development
- Allowing transparent migration to new features


Uniform Build System

- Maven supports *archetype* projects that standardize
  - directory structure
    - Source code kept in separate directory tree from both intermediate and final build products
    - Tests occupy separate subtrees of the source and product trees
  - "Life Cycle"
    - Really, a presumptuous name on their part for a build process
    - A sequence of *goals*
- Archetypes can be obtained form the Maven project or tailored for an organization.


Providing quality project information

- Provides easy access to report tools
- Aids in building & maintaining project web sites (e.g.)


Providing guidelines for best practices development

- Directory structures (already discussed)
- Unit testing
- Encourage familiarity with approved archetypes


Outline I


Maven as a Build Manager

Perhaps the best way to illustrate this is to follow the steps in Maven in 5 Minutes

- Start with the command

```
mvn archetype:generate -DgroupId=edu.odu.cs \
  -DartifactId=codeAnnotation \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DinteractiveMode=false
```

  - Lots of libraries (mainly comprising the latest version of Maven system itself) will be downloaded into your `~/.m2` directory.
  - A directory, `codeAnnotation`, will be created.
- **cd** into the new directory and explore
  - `src` directory structure
  - `pom.xml` is the *build file* for this project


Building with the Sample Source

- Run

```
mvn package
```

  - Sample source code is compiled
  - Sample unit test is run and executed.
  - A jar file is created with the sample source code

- Explore the `target` directory to see what has been placed there


OK, that was fun...

Let's try this with some real source code.

- Delete the `target` directory (or run `mvn clean`)
- Replace the contents of `src/main/java` and `src/test/java` by the corresponding contents from my code Annotation project.
  - Also, copy `src/main/jflex` while we're at it, though we won't use this right away.
- Try `mvn package` again.


What Went Wrong?

- A glance at the code with the error messages won't show anything obvious.
  - But I happen to know that Maven defaults to running the Java compiler in Java 5 compatibility mode
  - This code uses Java 6 features
- Edit the `pom.xml` file and, just above the `<dependencies>` section, add

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.bui
    <maven.compiler.source>1.6</maven.compiler.source
    <maven.compiler.target>1.6</maven.compiler.target
 </properties>
```

- Then try `mvn package` again


That was a little better

- The main source code compiled successfully.
- The error was in the compilation of the unit tests
  - The first error message says
    ```
    [ERROR]
    /home/zeil/cs795SD/Build/lib/codeAnnotation/src/test/java
    package org.junit does not exist
    ```
- Looks to be a problem with the JUnit library


That was a little better

- The main source code compiled successfully.
- The error was in the compilation of the unit tests
  - The first error message says
    ```
    [ERROR]
    /home/zeil/cs795SD/Build/lib/codeAnnotation/src/test/java
    package org.junit does not exist
    ```
- Looks to be a problem with the JUnit library
- In the original project, I was keeping a copy of `junit4.jar` in the project directory.
  - A clumsy solution


That was a little better

- The main source code compiled successfully.
- The error was in the compilation of the unit tests
  - The first error message says
    ```
    [ERROR]
    /home/zeil/cs795SD/Build/lib/codeAnnotation/src/test/java
    package org.junit does not exist
    ```
- Looks to be a problem with the JUnit library
- In the original project, I was keeping a copy of `junit4.jar` in the project directory.
  - A clumsy solution
  - Which brings us to ...


Outline I


Maven as a Configuration Manager

- Edit the `pom.xml` file and look for the `dependencies` section. You should see something like

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

  - This indicates that our project requires the `junit` package.
    - But it's an old version. JUnit changed a *lot* in version 4.


Updating the Dependency

- Change the version number to

```
<dependencies>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.10</version>
  <scope>test</scope>
</dependencies>
```

  - Could also say `[4.10,]` to get versions 4.10 or greater

- Try `mvn package` again.


Fetching Dependencies

- Maven does a transitive search over the dependencies for a project
  - Tries to find a mutually compatible set of versions
    - Helps if you give it some flexibility
- Maven then downloads the required libraries automatically
  - Downloaded libraries are cached (e.g., `~/.m2`)


Version Ranges

- Version ranges can be specified as `[`*low*`,`*high*`]`

  and are inclusive. Use (and) for exclusive
  - If *low* or *high* are omitted, taken as $\pm\infty$
- Examples
  - `[4.0,4.10]` inclusive range
  - `[4.10,)` 4.10 or anything more recent
  - `[4.10]` 4.10 and nothing else
  - `4.10` with no brackets means "any version, but prefer 4.10"


Maven Repositories

- By default, Maven searches the ibiblio repository, which can be human-searched here.
- Try searching for junit
  - Notice range of versions available
    - Select one (e.g., 4.10)
  - The "Maven" tab shows what to put into your `<dependencies>` section to request this version.


Transitive Dependencies I

How does Maven know whether `junit` itself depends on other libraries?

- Near the top of the junit 4.10 page, click to "View" the POM file:
  Near the bottom, you will see

```
<dependencies>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-core</artifactId>
  <version>1.3</version>
  <scope>compile</scope>
</dependencies>
```

- This is the same kind of info that we put into our own `pom.xml` file


Transitive Dependencies II

- And is, presumably, taken from the `pom.xml` that the JUnit team used to maintain their builds.
- Publishing the dependency information along with the libraries leads to an accumulated base of information on library dependencies.


Choosing Repositories

- Technically, our project has two repositories
  - a *remote repository*, at ibiblio
  - a *local repository* holding my cache
- Team projects will often employ an intermediate *shared repository*
  - reduce cache duplication
  - provide a mechanism for managing subproject modules
  - provide a common storage area for libraries not available on ibiblio
  - preserve older versions needed for a project baseline


Example: the Forge350 Repository I

A basic shared repository can be found here.
To add access to it, we add this to our `pom.xml`:

```
<repositories>
  <repository>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <id>forge350</id>
    <name>ODU CS Forge350 Repository</name>
    <url>http://forge350.cs.odu.edu/www/mavenrepo1/repo
  </repository>
</repositories>
```


Example: the Forge350 Repository II

- We could then, for example, import early versions of zipdiff and trilead-ssh that were part of the baseline for the Extract project
  - At the time they were added to the project baseline, they were not in the remote Maven repository
  - As time has passed, those libraries were added, but have started with versions later than the project baseline


Publishing to a Repository

Suppose that we want to publish our jar to a repository.

- "Output" repositories are listed similarly to, but separately from, input repositories
  - Necessary because the network protocol may be different

```
<repositories>
  ⋮
  </repository>
</repositories>
<distributionManagement>
  <repository>
    <id>forge350Scp</id>
    <name>ODU CS Forge350 Repository (scp)</name>
    <url>scp://forge350.cs.odu.edu/var/lib/gforge/chro
  </repository>
</distributionManagement>
```

Connection Info

In addition to adding the output repository to the POM, you need to provide your authentication info for the servers that you use in `~/.m2/settings.xml`:

settings.xml.listing

❶ This ID provides a handle for the distributionManagement tag in your POM to use identifying where you want to publish to.

❷ This is a clear security issue.

- Only very recently has an option been added that allows you to use encrypted passwords instead of plain text.
  - I'm still not sure how secure that really is.

❸ Probably a better idea is to use public/private key access.

- The `privateKey` tag might not even be required if you are using a key agent.


We're Not Done Yet

Our repository can be accessed by **scp**

- Of course, you can't actually do something like **scp** without a library that supports it.
  - Unlike the libraries we list in the dependencies, we need something that would become part of the maven process, not simply passed to the compiler or to the


Extending Maven

- Maven is usually extended by adding *plugins*
  - Each plugin is a collection of related *MOJO*s, a Maven plain Old Java Object
    - Each MOJO provides a specific function
    - which can be bound to a specific goal of the life cycle for our selected project archetype

Circling the Wagons

However, for whatever reason, communication with repositories has a separate extension mechanism, called *wagons*.


Circling the Wagons

However, for whatever reason, communication with repositories has a separate extension mechanism, called *wagons*.

- 'Cause they ship things from one place to another, get it?

```
</dependencies>
<build>
  <extensions>
    <extension>
      <groupId>org.apache.maven.wagon</groupId>
       <artifactId>wagon-ssh</artifactId>
       <version>2.2</version>
    </extension>
  </extensions>
</build>
```


If all is well...

- `mvn deploy` should then publish your project to the designated repository.


Modification via Plugin

- Much of the source code in this project is generated by **jflex**
- Currently, we have replied on that generated code already being in our `src/main/java`
- Let's fix that.
  - First, delete the `*Scanner.java` files
- Maven has a `jflex` plugin that will convert jflex inputs in `src/main/jflex` to Java source code in the `target` directory and will include that generated code in the normal compilation.


Adding the JFlex Plugin

- Search the Maven ibiblio for the plugin
  - "jflex-plugin", not the "jflex" library
    - the library will be obtained as well, but as a transitive dependency
- Modify the POM:
  pom.xml.jflex.listing


Outline I


Maven Report Generation

- Report generation is considered a key feature of Maven


Generating a Website

`mvn site` will generate a project website.

- Most of the content will be highly generic, however.
- The missing details have to be added to the POM **pom.xml.website.listing**


Adding Reports

- A variety of report generators can then be run and have their reports linked in to the generated site
  pom.xml.reports.listing
  - Some tools need to have plugins loaded to run the tool itself at the appropriate point in the build process. ❶
    - E.g., Test coverage tools ❷ typically need to modify the compilation settings.
  - Most are simply listed as plugins in a new `reporting` section ❸
    - ❹ Javadoc
    - ❺ Syntax-highlighted source listings
    - ❻ Unit test reports
    - ❼ PMD static analysis/style check
    - ❽ Test coverage
    - ❾ General report handling


Outline I


Further Modifying Maven Projects I

Maven is wonderful when your project matches the archetype.
When it doesn't...

- Look for a plug-in
  - Plugging it in to the build model requires familiarity with both the mojos provided by the plugin and the archetype's life cycle model

```
<plugin>
  <groupId>de.jflex</groupId>
  <artifactId>maven-jflex-plugin</artifactI
  <version>1.4.3</version>
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
```


Further Modifying Maven Projects II

```
            </execution>
          </executions>
        </plugin>
```

- How did I even know that "generate" was available as a goal?
- Maven plugin documentation is often atrocious


Sometimes Doing the Easiest Things

. . . will be the hardest thing to accomplish in Maven

- Example
  Basically, Maven is wonderful when your projects matches an archetype, and tortuous when it does not.


Maven and Ant

One of the more popular ways to "escape" when Maven is getting in the way of a simple step is the `antrun` plugin

- For example, Maven's built-in process for deciding what goes into a .jar file is to include the compiled classes and anything provided in `src/main/resources`
- But what if I want to include some data files that were constructed during the build process?
  - First, I need to run the programs to build those files.
    - This definitely requires escaping from the normal life cycle
  - Then I need to get them included into the Jar file
    - The built-in mechanism for this is incredibly cumbersome
- Easiest to handle both via a couple of embedded **ant** tasks

Maven and Eclipse

- Integrating Maven into **eclipse** is more than just allowing you run **mvn** as the build command
- The Eclipse built-in compiler also needs to know what libraries Maven has pulled into your configuration
  - Otherwise it will issue bogus error messages and inappropriate help suggestions
- Can also hope for convenience functions
  - In particular, POM editors
- Two plugins currently
  - m2eclipse, older plugin
  - Eclipse IAM