Sponsored by

CHARIOT SOLUTIONS

Taming Maven

Keith D Gregory
Philly JUG
14 November 2012

Reasons to Love Maven

- Dependency management
- Consistent project structure
- 97% of the time, It Just Works

Reasons to Hate Maven

- Verbose and Repetitive
- “Repeatable Builds” are a myth
- It's the Maven Way or the Highway
- 0.01% of the time, welcome to the Pit of Despair

Alternatives

- Gradle
  - Mix of Declarative and Imperative
  - Ready for Prime Time?
- Ant/Maven Ant Tasks
  - Dependency Management and Deployment for existing Ant scripts
- Ant/Ivy
  - Dependency Management for Ant
  - YABF (Yet Another Build File)
- Other (typically non-Java JVM projects)

What is Maven?

- A framework to run plugins
  - Compiler / Test-Runner / Jar / &c
  - Will download plugin versions as needed
  - Each plugin has one or more goals
- Build process consists of phases
  - clean / compile / test / package / &c
- Arbitrary goals may be bound to arbitrary phases
- Default life-cycle covers most cases
  - Bindings defined by Maven, based on artifact type
  - mvn clean install

Hello World – POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example.hello</groupId>
  <artifactId>hello</artifactId>
  <version>1.0-SNAPSHOT</version>

</project>
```

Standard Project Layout

- pom.xml
- src/main/java
- src/main/resources
- src/test/java
- src/test/resources
- src/main/webapp
- src/site

Dependencies

- Specifies the JARs you need to build and run

```
<dependencies>
   <dependency>
      <groupId>net.sf.practicalxml</groupId>
      <artifactId>practicalxml</artifactId>
      <version>1.1.14</version>
   </dependency>
   <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.10</version>
      <scope>test</scope>
   </dependency>
</dependencies>
```

- Scopes: compile, test, runtime, provided

Transitive Dependencies

- The dependencies that your dependencies depend upon
- Beware: transitive dependencies are indistinguishable from direct … until the direct dependency disappears
- Transitive dependencies may also introduce unexpected JARs, duplicate classes

Archetypes

- Generates project directories / files
  - Slightly less effort than copying existing project
- Value is in the files created (eg: web.xml)
- Allows consistent customization

Parent POM

- ***Not*** just used for multi-module projects
- POMs form a hierarchy
  - Every POM descends from the “super POM”
  - Children inherit settings from their parents, can override
- POMs are versioned, just like other artifacts

Plugin Management

- Parent POM specifies common plugin config
  - Example: Source/Target compiler versions
- Child POM can override

Dependency Management

- Parent POM can specify version of dependencies used by children
  - This is usually a Bad Thing
  - Version ranges make it less of a Bad Thing
- Also allows consistent exclusion of transitive dependencies
  - This can be a Good Thing
- Use sparingly!

Version Properties

- Defined in <properties>
  - Can be overridden on command line
  - Or in child POM
- Used in <dependency>
- Properties not limited to versions

Extract Dependencies

- Useful when many projects have the same set of dependencies
  - Example: Spring MVC Web-Apps
- Create a project that contains just a POM with a <dependencies> section
- Your projects depend on this “dependency” project
  - And they get the transitive dependencies

Local Repository Server

- Local server for artifacts
  - Nexus
  - Artifactory
  - Apache
- Proxies Maven Central, other public repos
- Allows upload of restricted 3rd-party artifacts
- A place to deploy corporate artifacts

Continuous Build/Deploy

- Deploy snapshots as soon as they're built
  - Can also configure to deploy production builds
- Servers that understand Maven will automatically build dependent projects
  - Flushes out incompatible changes

The Goal: More, Smaller Projects

- Reduces amount of work in each build
- Projects should be self-contained modules
- Path to OSGi

Tools

- M2Eclipse (m2e)
- Maven Dependency Plugin
  - dependency:analyze
  - dependency:tree
  - dependency:build-classpath
- PomUtil (http://github.com/kdgregory/pomutil)
  - Normalize dependencies and add version props
  - Check for used/unused/mis-scoped dependencies
  - Inter-dependencies, co-dependencies, build order
  - Generate a parent POM

Sponsored by

CHARIOT SOLUTIONS

Taming Maven

Keith D Gregory
Philly JUG
14 November 2012

Reasons to Love Maven

- Dependency management
- Consistent project structure
- 97% of the time, It Just Works

Dependency Management: no project lives in vacuum; repository of open-source software; encourages moudlarization (use GSI “JAR” as an example

Reasons to Hate Maven

- Verbose and Repetitive
- “Repeatable Builds” are a myth
- It's the Maven Way or the Highway
- 0.01% of the time, welcome to the Pit of Despair

Repeatable builds: an interesting idea, but taken to an extreme; Maven will complain if you don't specify the versions of all your plugins – and the developers will close bug reports if you don't, even if the bug is clearly their fault (eg: uploading snapshot plugins to the central repository).

For a truly repeatable build, you'd need to guarantee the same JDK, possibly the same OS; plugin versions don't even come close

Summary: “if Winston Churchill were alive today, he might say that 'Maven is the worst build system, except for all the others'” (segue to the alternatives)

Alternatives

- Gradle
  - Mix of Declarative and Imperative
  - Ready for Prime Time?
- Ant/Maven Ant Tasks
  - Dependency Management and Deployment for existing Ant scripts
- Ant/Ivy
  - Dependency Management for Ant
  - YABF (Yet Another Build File)
- Other (typically non-Java JVM projects)

What is Maven?

- A framework to run plugins
  - Compiler / Test-Runner / Jar / &c
  - Will download plugin versions as needed
  - Each plugin has one or more goals
- Build process consists of phases
  - clean / compile / test / package / &c
- Arbitrary goals may be bound to arbitrary phases
- Default life-cycle covers most cases
  - Bindings defined by Maven, based on artifact type
  - mvn clean install

Hello World – POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example.hello</groupId>
  <artifactId>hello</artifactId>
  <version>1.0-SNAPSHOT</version>

</project>
```

Describe elements

Describe process for picking group and version

“It's not this easy in the real world”

Standard Project Layout

- pom.xml
- src/main/java
- src/main/resources
- src/test/java
- src/test/resources
- src/main/webapp
- src/site

Not all of these directories are used for every project; can add as necessary

If you really, really want to change the directories, you can (mostly)

Dependencies

- Specifies the JARs you need to build and run

```
<dependencies>
   <dependency>
      <groupId>net.sf.practicalxml</groupId>
      <artifactId>practicalxml</artifactId>
      <version>1.1.14</version>
   </dependency>
   <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.10</version>
      <scope>test</scope>
   </dependency>
</dependencies>
```

- Scopes: compile, test, runtime, provided

Incorrect scopes: test-only artifacts that aren't marked “test” and appear in released project (eg: hsqldb); scopes marked as “provided” when they aren't (eg: spring-test dependency on spring-core)

Examples: test = Junit; runtime = javassist; provided = servlet

Transitive Dependencies

- The dependencies that your dependencies depend upon
- Beware: transitive dependencies are indistinguishable from direct … until the direct dependency disappears
- Transitive dependencies may also introduce unexpected JARs, duplicate classes

Examples: practicalxml depends on kdgcommons; spring-core depends on ???

Unexpected dependencies: commons-lang vs commons-lang3; org.apache.ant vs ant

Note that Maven is smart about picking highest version for same group/artifact

Archetypes

- Generates project directories / files
  - Slightly less effort than copying existing project
- Value is in the files created (eg: web.xml)
- Allows consistent customization

Parent POM

- ***Not*** just used for multi-module projects
- POMs form a hierarchy
  - Every POM descends from the “super POM”
  - Children inherit settings from their parents, can override
- POMs are versioned, just like other artifacts

Examples: PathFinder (multi-module parent), KDGCommons (Sonatype parent)

Plugin Management

- Parent POM specifies common plugin config
  - Example: Source/Target compiler versions
- Child POM can override

Dependency Management

- Parent POM can specify version of dependencies used by children
  - This is usually a Bad Thing
  - Version ranges make it less of a Bad Thing
- Also allows consistent exclusion of transitive dependencies
  - This can be a Good Thing
- Use sparingly!

Explanation: child POM refers to dependency using just group/artifact; *need an example*

Definition of Bad Thing: if one of your dependencies requires a transitive dependency that's newer than your DM allows, your artifact is broken

Exclusion example: spring-core dependency on commons-logging; note that spring-test relies on this dependency

Version Properties

- Defined in <properties>
  - Can be overridden on command line
  - Or in child POM
- Used in <dependency>
- Properties not limited to versions

Extract Dependencies

- Useful when many projects have the same set of dependencies
  - Example: Spring MVC Web-Apps
- Create a project that contains just a POM with a <dependencies> section
- Your projects depend on this “dependency” project
  - And they get the transitive dependencies

Local Repository Server

- Local server for artifacts
  - Nexus
  - Artifactory
  - Apache
- Proxies Maven Central, other public repos
- Allows upload of restricted 3rd-party artifacts
- A place to deploy corporate artifacts

Pre-configure Nexus (GSI doesn't have network); show repository list

Show settings.xml, run build

Upload artifact? Or show deploy?

Continuous Build/Deploy

- Deploy snapshots as soon as they're built
  - Can also configure to deploy production builds
- Servers that understand Maven will automatically build dependent projects
  - Flushes out incompatible changes

Install Jenkins from distribution (run WAR)

Add BCELX as a project (using SVN repository); show default build configuration

Add PomUtil as a project, show how Jenkins recognizes it as a dependency of BCELX

Make change to BCELX that will break Jenkins (remove method to find dependencies); show how build propagates – *and that, without clean the default build breaks unit tests only*

Update build to clean before building, show that it now breaks fully

The Goal: More, Smaller Projects

- Reduces amount of work in each build
- Projects should be self-contained modules
- Path to OSGi

Show pathfinder and pomutil as examples

Note the project naming conventions (originated at Navteq): lib-XXX, app-XXX, test-XXX, web-XXX

OSGi example from NFJS: using Glassfish server, individual modules can be replaced; similar to early Java development, where individual classes could be shipped to customers

Tools

- M2Eclipse (m2e)
- Maven Dependency Plugin
  - dependency:analyze
  - dependency:tree
  - dependency:build-classpath
- PomUtil (http://github.com/kdgregory/pomutil)
  - Normalize dependencies and add version props
  - Check for used/unused/mis-scoped dependencies
  - Inter-dependencies, co-dependencies, build order
  - Generate a parent POM