Basic introduction to maven

Attila Balogh-Biró

Topics

- Project Object Model (POM)
- Inheritance and Modules
- Dependencies
- Build Configuration
- Whirlwind Tour of Plugins
- Lifecycles
- Build Profiles
- Sometimes Maven Lets You Down

Maven Terminology

- With maven,
  - you execute goals in plugins over the different phases of the build lifecycle,
  - to generate artifacts
  - Examples of artifacts are jars, wars, and ears
  - These artifacts have an
    - artifactId, a groupId, and a version. Together, these are called the artifact's "coordinates."
    - The artifacts stored in repositories.
    - A POM (Project Object Model) describes a project.

Create a Project Directory

- Maven has a command
- for starting a project:

```
mvn archetype:create \
  -DgroupId=city.ui \
  -DartifactId=empty-connector \
  -DpackageName=city.ui.connector \
  -Dversion=1.0
```

Create a Project Directory

The **Maven** Way

Plugin Name

```
mvn archetype:create \
  -DgroupId=city.ui \
  -DartifactId=empty-connector \
  -DpackageName=city.ui.connector \
  -Dversion=1.0
```

Create a Project Directory

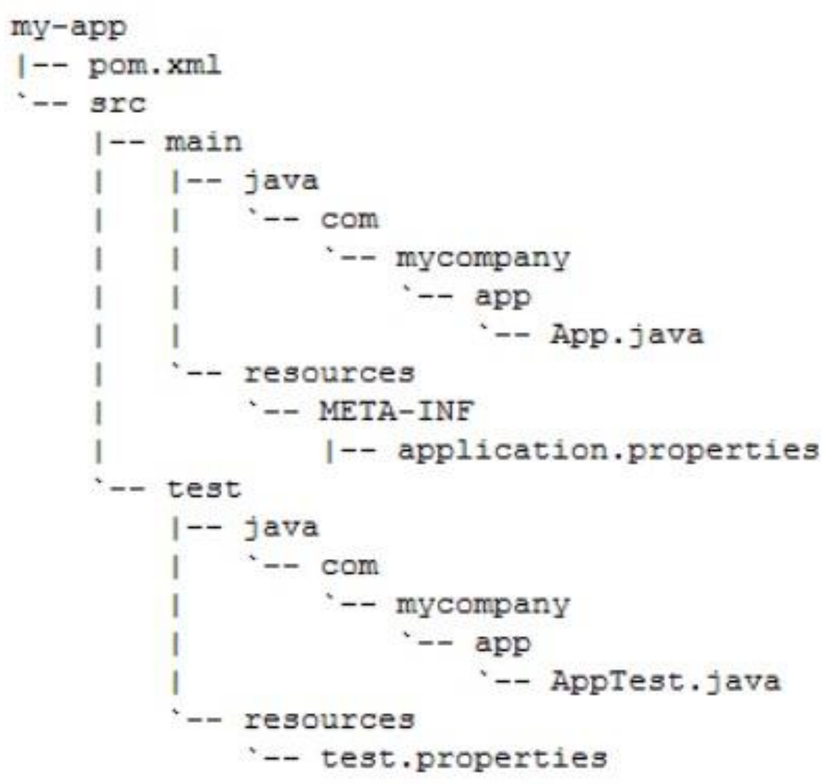
The Maven Way

Plugin Name

Goal

```
mvn archetype:create \
  -DgroupId=city.ui \
  -DartifactId=empty-connector \
  -DpackageName=city.ui.connector \
  -Dversion=1.0
```

Standard Maven Directory Layout



```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   `-- com
    |   |       `-- mycompany
    |   |           `-- app
    |   |               `-- App.java
    |   `-- resources
    |       `-- META-INF
    |           |-- application.properties
    `-- test
        |-- java
        |   `-- com
        |       `-- mycompany
        |           `-- app
        |               `-- AppTest.java
        `-- resources
            `-- test.properties
```

Set up the dependencies

Open pom.xml. We need to tell maven that we have a some dependency:

```
<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>city.ui</groupId>
  <artifactId>empty-connector</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <name>demo-mvn</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

We'll add the dependency here.

Set up the dependencies

This is all we need to add:

```
<dependency>
   <groupId>city.ui</groupId>
   <artifactId>UPSecurityRealm</artifactId>
   <version>1.0</version>
</dependency>
```

We don’t need to tell Maven about any of the jars on which the Security realm depends; Maven takes care of all of the transitive dependencies for us!

Configuring Maven

- Settings Files (settings.xml)
    - In ~/.m2 (per-user settings) and in Maven's install directory, under conf (per-system settings)
    - Alternate location for repository
    - Proxy Configuration
    - Per-server authentication settings
    - Mirrors
    - Download policies, for plugins and repositories; snapshots and releases.

Configuring Maven

- Project Object Model (pom.xml)
    - Inherited – individual projects inherit POM attributes from parent projects, and ultimately inherit from the “Super POM”
    - The Super POM is in Maven’s installation directory, embedded in the uber jar.
    - The Super POM defines, among lots of other things, the default locations for the plugin and jar repositories, which is http://repo1.maven.org/maven2

Repositories

- Local - in ~/.m2/repository
- Remote - e.g., http://repo1.maven.org/maven2 or another internal company repository (any directory reachable by sftp will do).
- Contains dependencies and plugins
- Can be managed by a “Repository Manager” like Nexus

The POM

- Describes the project, declaratively
- General Information - Project Coordinates (groupId, artifactId, Version)
- Build Settings – Configuration of the plugins
- Build Environment – We can configure different profiles that can be activated programatically
- POM Relationships – Dependencies on other projects

Anatomy of a POM File

Let’s check an existing POM file..

General Information

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
   <modelVersion>4.0.0</modelVersion>
   <name>Super Project</name>
   <packaging>jar</packaging>
   <groupId>city.ui</groupId>
   <artifactId>super</artifact>
   <version>1.0.0</version>
      . . .

</project>
```

Coordinates

Project Inheritance

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
   <modelVersion>4.0.0</modelVersion>
   <name>Super Project</name>
   <packaging>jar</packaging>
   <groupId>city.ui</groupId>
   <artifactId>super</artifact>
   <version>1.0.0</version>
   <parent>
     <!-- Parent POM stuff if applicable -->
   </parent>
   <modules>
          . . .
</project>
```

Project Inheritance

- What is inherited?
  - Identifiers (groupId, artifactId, one must be different)
  - Dependencies
  - Plugin, Report Lists
  - Plugin Configurations

- Why Inherit?
  - Don’t repeat yourself, e.g., several projects use the same version of logback.
  - Enforce plugin version across projects

Multimodule Projects

- *Not the same thing as POM inheritance!*
- A multimodule project builds submodules, but rarely produces an artifact itself
- Directory structure mimics module layout (e.g., if B is a submodule of A, then B will be a subdirectory of A).

Multimodule: Reactor

- When Maven encounters a multimodule project, it pulls all of the POMs into the “Reactor”
- The Reactor analyzes module inter-dependencies to ensure proper ordering.
- If no changes need to be made, the modules are executed in the order they are declared.
- Maven then runs the goals on each module in the order requested.

User-Defined Properties

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
   <modelVersion>4.0.0</modelVersion>
   <name>Super Duper Amazing Project</name>
   <packaging>jar</packaging>
   <groupId>city.ui</groupId>
   <artifactId>demo</artifact>
   <version>1.0.0</version>
   <parent>
     <!-- Parent POM stuff if applicable -->
   </parent>
   <modules>
     <!-- Sub-modules of this project -->
   </modules>
   <properties>
     <!-- Ad-hoc properties used in the build -->
   </properties>
```

User-Defined Properties

- User-Defined properties are like ant properties:

```
<properties>

  <vertx.version>3.0</vertx.version>

</properties>

...

<dependencies>

  <dependency>

    <groupId>vertx</groupId>

    <artifactId>vertx.core</artifact>

    <version>${vertx.version}</version>

  </dependency>

</dependencies>
```

Other Properties

- Maven Properties, project.*
  - `${project.version}`
- Settings Properties, settings.*
  - `${settings.interactiveMode}`
- Environment Variables, env.*
  - `${env.JAVA_HOME}`
- Java System Properties
  - `${java.version}, ${os.arch}, ${user.dir}`

Dependencies

```
<dependencies>
    <dependency>
        <groupId>vertx</groupId>
        <artifactId>vertx.core</artifact>
        <version>3.0</version>
        <scope>compile</scope>
        <optional>false</optional>
    </dependency>
</dependencies>
```

Dependencies

groupId and artifactId: must be unique

```
<dependencies>
        <dependency>
        <groupId>vertx</groupId>
        <artifactId>vertx.core</artifact>
        <version>3.0</version>
              <scope>compile</scope>
              <optional>false</optional>
        </dependency>
</dependencies>
```

Dependencies

```
<dependencies>
      <dependency>
            <groupId>vertx</groupId>
            <artifactId>vertx.core</artifact>
            <version>3.0</version>
            <scope>compile</scope>
            <optional>false</optional>
      </dependency>
</dependencies>
```

- ***3.0*** - prefer version 3.0, newer version is acceptable to resolve conflicts
- ***(3.0,3.3)*** - Any version between 3.0 and 3.3, exclusive
- ***[3.0,3.3]*** - Any version between 3.0 and 3.3 inclusive
- ***[,3.3]*** - Any version up to, and including, 3.3
- ***[3.3]*** - Only version 3.3, do not use a newer version.

Dependencies

```
<dependencies>
    <dependency>
        <groupId>vertx</groupId>
        <artifactId>vertx.core</artifact>
        <version>3.0</version>
        <scope>compile</scope>
        <optional>false</optional>
    </dependency>
</dependencies>
```

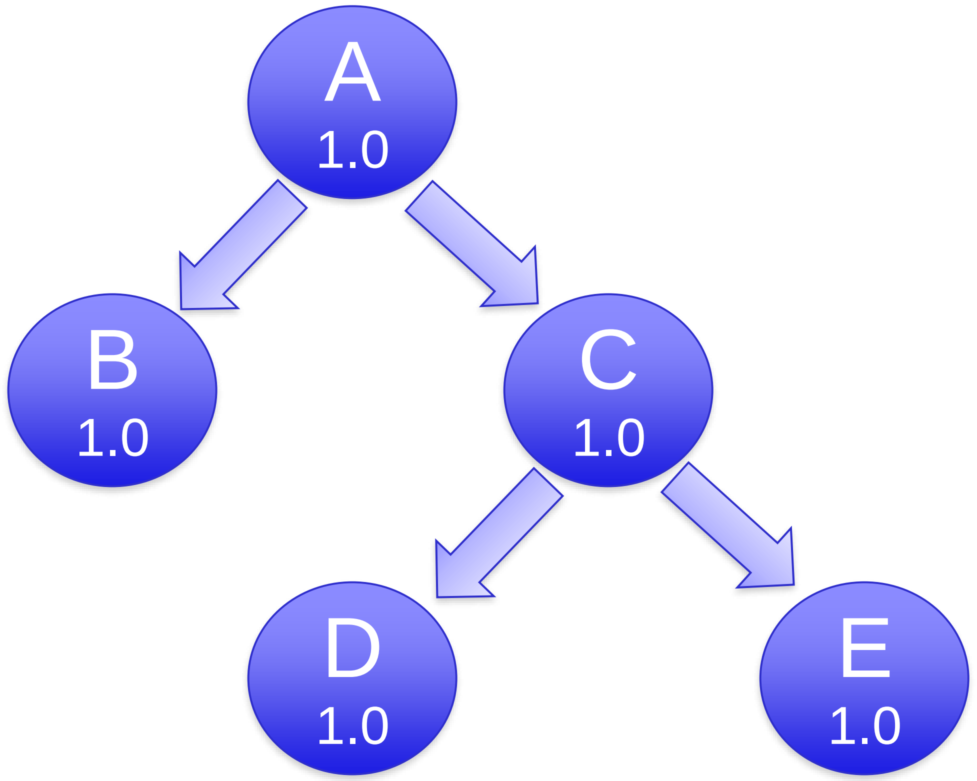
- ***compile*** - default, packaged. Available on compile-time and runtime CLASSPATH.
- ***provided*** - you expect the JVM or app container to provide the library. Available on compile-time CLASSPATH.
- ***runtime*** - needed to run, but not compilation (e.g., a JDBC driver)
- ***test*** - only needed during test execution (e.g., JUnit)

Dependencies

```
<dependencies>
    <dependency>
    <groupId>vertx</groupId>
    <artifactId>vertx.core</artifact>
    <version>3.0</version>
        <scope>compile</scope>
        <optional>false</optional>
    </dependency>
</dependencies>
```

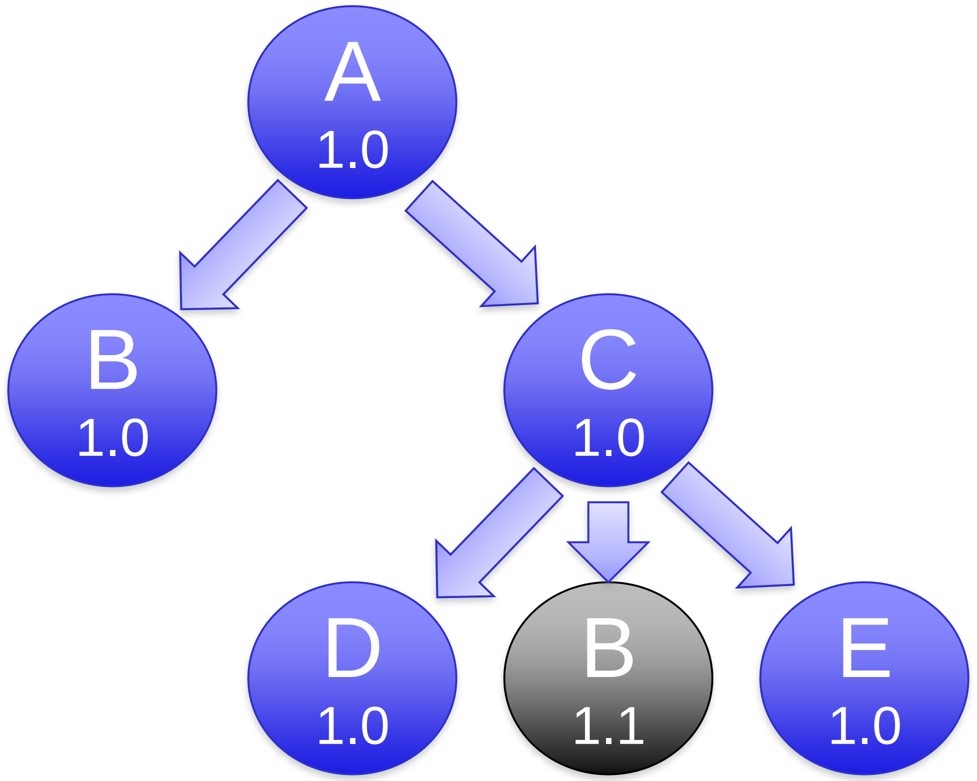
Prevents this dependency from being included as a transitive dependency if some other project depends on this project.

Transitive Dependencies



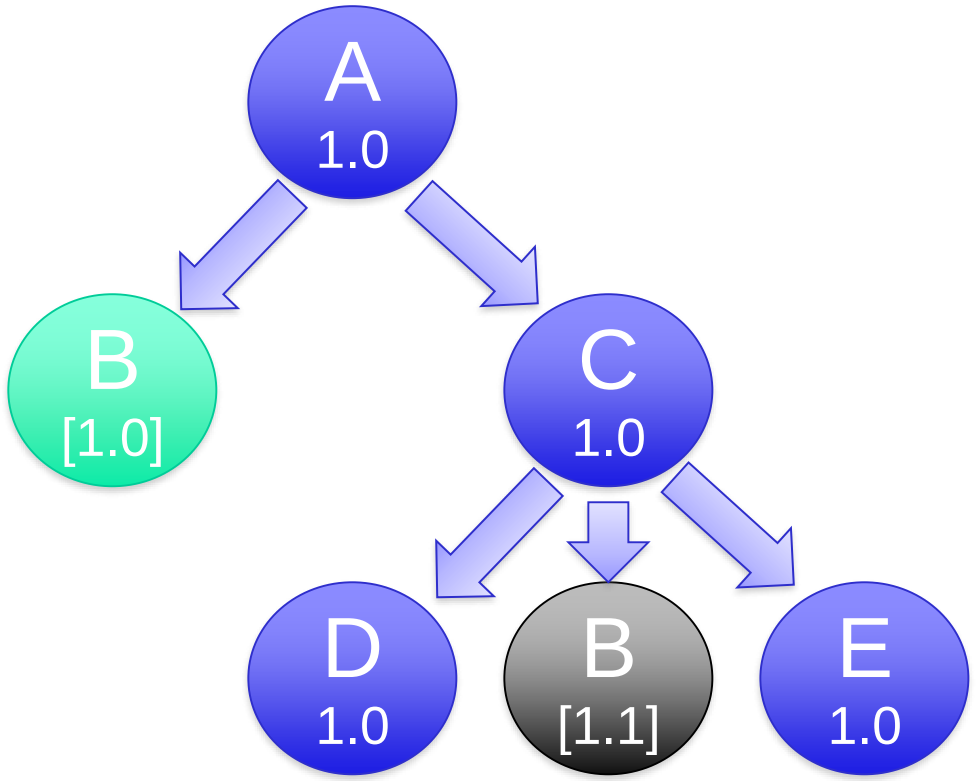
Our project (Project “A”) depends on B and C. Project C depends on projects D and E. Thus, our project depends on B, C, D, and E, and Maven will fetch and use these artifacts appropriately.

Transitive Dependencies



Now, let's say project C has a dependency on project B, but requires version 1.1. If project A's POM doesn't explicitly require version 1.0 or earlier, then Maven will choose version 1.1.

Transitive Dependencies



Uh oh. Now Project A is saying that it must use version 1.0 of B, and only version 1.0, and project C needs version 1.1 of project B.

Dependency Exclusions

One way to deal with conflicts is with exclusions

```
<dependencies>
    <dependency>
        <groupId>city.ui</groupId>
        <artifactId>project-b</artifactId>
        <version>[1.0]</version>
    </dependency>
    <dependency>
        <groupId>city.ui</groupId>
        <artifactId>project-c</artifactId>
        <exclusions>
            <exclusion>
                <groupId>city.ui</groupId>
                <artifactId>project-b</artifactId>
             </exclusion>
         </exclusions>
```

Dependency Management

Parent POM

```
<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring</artifactId>
   <version>2.5.5</version>
  </dependency>
 </dependencies>
</dependencyManagement>
```

Child POM

```
<dependencies>
 <dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring</artifactId>
 </dependency>
</dependencies>
```

The `dependencyManagement` element allows you to specify version numbers of dependencies in child POMs without making all children dependent on a particular library.

SNAPSHOT Versions

- SNAPSHOT is a literal string appended to a version number, e.g., 1.2.3-SNAPSHOT
- Indicates that a version is “under development”
- Use if you need Maven to keep checking for the latest version
- Maven replaces SNAPSHOT with a UTC time stamp before putting it into the repository.

Build Configuration Plugins

- All work in Maven is performed by plugins
- Like Dependencies, are Downloaded from a repository
- Because they are shared, you often benefit from the fact that someone else has already built a plugin for whatever function you may need

Plugin Configuration

- The plugin section of the POM has a configuration element where you can customize plugin behavior:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-clean-plugin</artifactId>
      <version>2.2</version>
      <configuration>
        <!-- Configuration details go here -->
      </configuration>
    </plugin>
  </plugins>
</build>
```

Core Plugins

Maven Plugin Whirlwind Tour

- clean - has only one goal, clean. Deletes the target directory, can be configured to delete other stuff
- compiler - compiles sources, uses javac compiler by default.
  - Has a compile and testCompile goal.
  - Can be configured to use any executable as the compiler
- deploy - uploads artifacts to a remote repository

Maven Plugin Whirlwind Tour

Core Plugins, cont.

- install - installs the artifact into the local repository.
  - install goal, install this project's artifact
  - install-file goal, install a specific file into local repo (good for third-party stuff)
- surefire - runs all of the unit tests in the test source directory, and generates reports.
- resources - copies resources to be packaged

Packaging Plugins

Maven Plugin Whirlwind Tour

- ear, ejb, jar, war
- assembly - builds a binary distribution including runtime dependencies
  - supports zip, tar.gz, tar.bz2, jar, dir, and war formats
  - uses “assembly descriptors” to configure (although several pre-fab ones are available)
  - one of the pre-fab descriptors builds executable jar files with all depenencies embedded

Utility Plugins

- archetype - builds skeleton of a working project for many different frameworks
  - Wicket, Tapestry, JSF, JPA, tons of others
- help - even the help is a plugin! Use the describe goal to learn what a plugin can do, e.g.,

```
     mvn help:describe -
Dplugin=compiler
```

- scm - source control stuff

Build Lifecycle

- Usually, an artifact is built by executing a sequence of goals
- For example, to generate a WAR:
  - Clean the build area
  - Copy the resources
  - Compile the code
  - Copy the test resources
  - Compile the test code
  - Run the test
  - Package the result

Maven's Lifecycles

- Maven supports three standard lifecycles
- clean - as you might expect, starts us fresh
- default - the lifecycle that builds the code
- site - a lifecycle for building other related artifacts (e.g., reports and documentation)

Clean Lifecycle

- The Clean Lifecycle has three phases:
- pre-clean
- clean
- post-clean

- Only clean is "bound" by default, to the clean goal of the clean plugin. You can bind other tasks using executions.

Executions

- Let’s say you have a whizz-bang plugin named *mp3*, and it has a goal named *play* that lets you play an arbitrary audio clip, and you’d like to play a clip during pre-clean:

```
<plugin>
   <groupId>city.ui</groupId>
   <artifactId>mp3</artifactId>
   <version>1.0</version>
   <executions>
     <execution>
       <phase>pre-clean</phase>
       <goals>
         <goal>play</goal>
       </goals>
       <configuration>
         <audioClipFile>toilet-flush.mp3</audioClipFile>
       </configuration>
     </execution>
   </executions>
 </plugin>
```

Maven's Default Lifecycle

- Maven models the software build process with the 21 step "default lifecycle"

validate	generate-test-sources	package
generate-sources	process-test-sources	pre-integration-test
process-sources	generate-test-resources	integration-test
generate-resources	process-test-resources	post-integration-test
process-resources	test-compile	verify
compile	test	install
process-classes	prepare-package	deploy

Package-Specific Lifecycles

- Maven automatically binds goals to the phases on the previous slide based on the packaging

Lifecycle Phase	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	war:war
install	install:install
deploy	deploy:deploy

Build Profiles

```
<project>
     . . .
     <!-- Build Configuration  -->
     <plugins>
       <!-- plugin configuration -->
     </plugins>
   </build>
        <profiles>
          <!-- build profiles -->
        </profiles>
</project>
```

Profiles: Customized Builds

- Sometimes our artifacts need to be tweaked for different “customers”
- The Development version has different logging or database configuration than QA or Production
- There might be slight differences based on target OS or JDK version

How to declare a profile

- In the POM itself, in an external profiles.xml file, or even in settings.xml

```
<project>
  ...
  <profiles>
    <profile>
      <id>appserverConfig-dev</id>
      <properties>
        <appserver.home>/path/to/dev/appserver</appserver.home>
      </properties>
    </profile>

    <profile>
      <id>appserverConfig-dev-2</id>
      <properties>
        <appserver.home>/path/to/another/dev/appserver2</appserver.home>
      </properties>
    </profile>
  </profiles>
  ...
</project>
```

Build Configuration in a Profile

- You can even configure plugins based on a profile:

```
<project>
     ...
     <profiles>
          <profile>
               <id>production</id>
                    <build>
                    <plugins>
                         <plugin>
                              ...
                              </plugin>
                    </plugins>
               </build>
          </profile>
...
```

Activating a Profile

- On the command-line:

```
    mvn package -Pmyprofile1,myprofile2
```

- In your settings.xml file:

```
<settings>
...
  <activeProfiles>
    <activeProfile>dev</activeProfile>
  </activeProfiles>
...
</settings>
```

- Activation elements

Activation Elements

```
<project>
…
  <profiles>
    <profile>
      <id>dev</id>
      <activation>
        <activeByDefault>false</activeByDefault>
        <jdk>1.8</jdk>
        <os>
          <name>Windows XP</name>
          <family>Windows</famliy>
          <arch>x86</arch>
          <version>5.1.2600</version>
        </os>
        <      …
```

Activation Elements

```
property>
          <name>mavenVersion</name>
          <value>2.0.9</value>
        </property>
        <file>
          <exists>file2.properties</exists>
          <missing>file1.properties</missing>
        </file>
      </activation>
</profile>
  </profiles>
</project>
```

Common Criticisms

- Poor Documentation - Lots of Maven's online documentation is automatically generated and is generally pretty horrible
- Simple things are sometimes counterintuitive with Maven - E.g., copying a file
- Maven adds to the number of places you need to look when something breaks - both your source repository, and the maven repository
- Everything breaks if someone changes an artifactId or groupId
- Doesn't work well if your network connectivity is unreliable or unavailable
- Gets tangled and confused if one of your transitive dependencies isn't available in a maven repository

THANK YOU FOR YOUR ATTENTION
PLEASE CLAP AND DON'T ASK TOUGH QUESTIONS
memegenerator.net