Demystifying maven

By Mike Desjardins

Topics

- Installation
- Shock and Awe: Comparison with ant
- Project Object Model (POM)
- Inheritance and Modules
- Dependencies
- Build Configuration
- Whirlwind Tour of Plugins
- Lifecycles
- Build Profiles
- Sometimes Maven Lets You Down

About Me

- Developed Software professionally for about 15 years, the last 5 with Java
- Independent Contractor/Consultant
- Live in Portland, Maine
- For the past several years, primary focus has been on web services
  - Wireless Telecommunications
  - Financial Services
  - Online Music
- Website: http://mikedesjardins.us
- Twitter: @mdesjardins

Installation

- Requires JDK 1.4
- Download Maven from http://maven.apache.org/
- Unzip the archive
- Add the M2_HOME environment variable and point it to your installation.
- Make sure JAVA_HOME is set and java is in your PATH.
- Make sure $M2_HOME/bin is in your PATH.

Ant vs. Maven

- Best illustrated by Example
- We will create a new trivial Hibernate Application that reads rows in a database table (shown below) and outputs them to the console.

greeting_id	greeting_text
1	Hello
2	Bon jour
3	Buenos dias
4	Kon ni chi wa

Example – Source Code

First, we will need a class to represent a greeting:

```
@Entity
public class Greeting {
  @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="greeting_id",nullable=false,unique=true )
  private Integer id;

  @Column(name="greeting_text")
  private String text;

  @Column(name="version") @Version
  private Integer version;

  public Integer getId() { return this.id; }
  public void setId(Integer id) { this.id = id; }

  public String getText() { return this.text; }
  public void setText(String text) { this.text = text; }
}
```

Example – Source Code

Next, we will need a class with a static main method to actually do stuff...

Example – Source Code

```
public class App {
  private static SessionFactory sessionFactory;
  static {
    try {
    sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory();
  } catch (Throwable t) {
    System.out.println("Something terrible has happened.");
    t.printStackTrace();
  }
 }

  public static SessionFactory getSessionFactory() { return App.sessionFactory; }

  public void run() {
   System.out.println("==========================================================");
   Criteria c = App.getSessionFactory().openSession().createCriteria(Greeting.class );
   List<Greeting> greetings = c.list();

    for (Greeting greeting : greetings) {
    System.out.println(greeting.getId() + ": " + greeting.getText());
   }

   System.out.println("==========================================================");
  }

  public static void main(String args[]) {
   App app = new App().run());
  }
}
```

Ant: Let's Get Started!

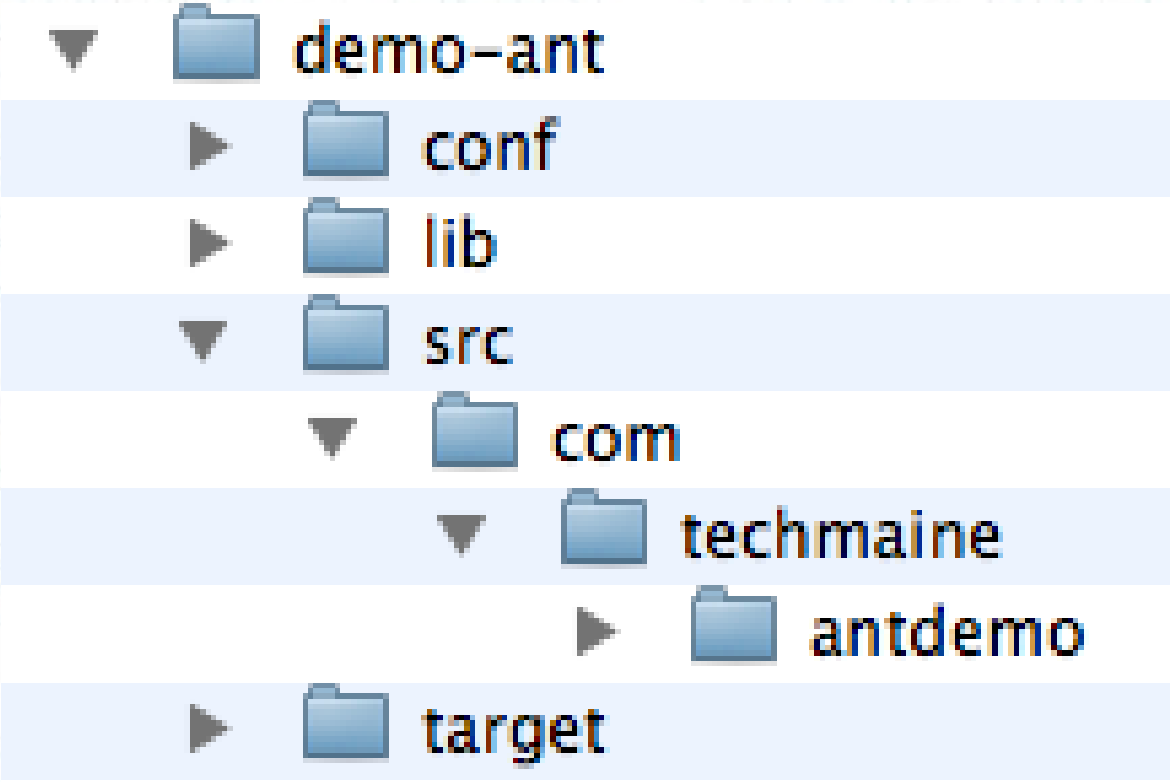
First, let's create the directories for our project:

```
Mac:~/_work$ mkdir demo-ant
Mac:~/_work$ cd demo-ant
Mac:~/_work/demo-ant$  mkdir src lib conf target
Mac:~/_work/demo-ant$  cd src
Mac:~/_work/demo-ant/src$  mkdir com
Mac:~/_work/demo-ant/src$  cd com
Mac:~/_work/demo-ant/src/com$  mkdir techmaine
Mac:~/_work/demo-ant/src/com$  cd techmaine
Mac:~/_work/demo-ant/src/com/techmaine$  mkdir antdemo
Mac:~/_work/demo-ant/src/com/techmaine$
```

Ant: Let's Get Started!

The ant way

(If you're more of a visual person)



Let's go get our jars!

The ant way

1.) Visit the Hibernate Website, eventually find their download page…

hibernate.org – Download Overview

http://hibernate.org/6.html

HIBERNATE

Support | Products | Partners

Login Register Careers Contact Us Search...

Download » Download Overview

- HIBERNATE Core
- HIBERNATE Annotations
- HIBERNATE EntityManager
- HIBERNATE Shards
- HIBERNATE Validator
- HIBERNATE Search
- HIBERNATE Tools
- NHIBERNATE
- News
- Documentation
- Download
- Forum & Mailinglists
- Support & Training
- JIRA Issue Tracking
- Wiki Community Area

Download Overview

- Binary Releases
- Compatibility Matrix
- Source Control Repositories
- Compiling Hibernate
- Compiling Hibernate Annotations, EntityManager, Search, Validator or Shards
- Compiling Hibernate Tools
- Compiling NHibernate

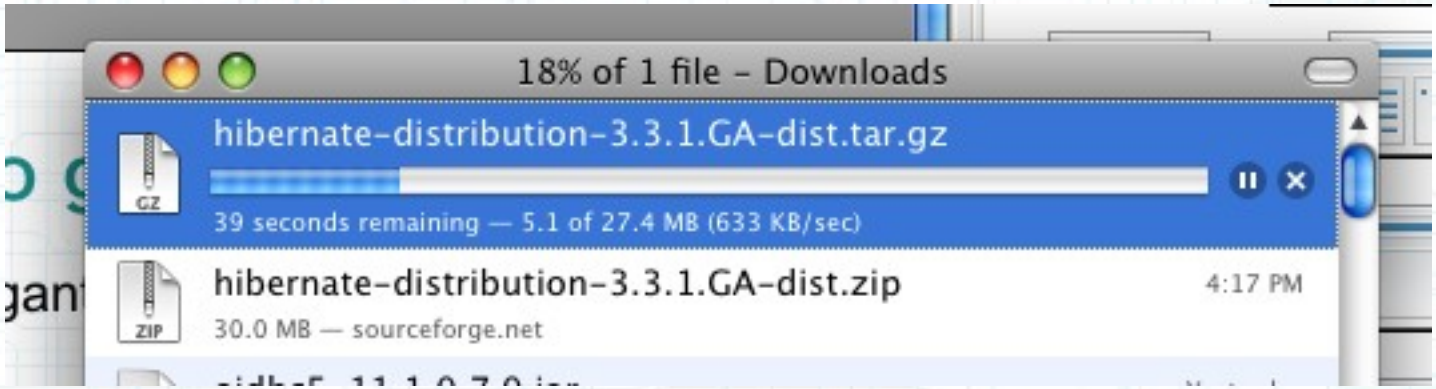
Binary Releases

Package	Version	Release date	Category	
Hibernate Core	3.3.1.GA	11.09.2008	Production	Download
Hibernate Annotations	3.4.0 GA	20.08.2008	Production	Download
Hibernate EntityManager	3.4.0 GA	20.08.2008	Production	Download
Hibernate Validator	3.1.0 GA	10.09.2008	Production	Download

Let's go get our jars!

The ant way

2.) Download gigantic ZIP file



ZZZZZZZZ….

Let's go get our jars!

The ant way

3.) Unpack the ZIP file

```
      0  09-10-08 12:27   hibernate-distribution-3.3.1.GA/
  26428  09-10-08 12:21   hibernate-distribution-3.3.1.GA/legal.txt
   1456  09-10-08 12:21   hibernate-distribution-3.3.1.GA/hibernate_logo.gif
 152761  09-10-08 12:21   hibernate-distribution-3.3.1.GA/readme.txt
2766130  09-10-08 12:27   hibernate-distribution-3.3.1.GA/hibernate3.jar
  31493  09-10-08 12:22   hibernate-distribution-3.3.1.GA/hibernate-testing.jar
      0  09-10-08 12:27   hibernate-distribution-3.3.1.GA/lib/
      0  09-10-08 12:27   hibernate-distribution-3.3.1.GA/lib/antlr/
 443432  06-13-08 12:[illegible]   hibernate-distribution-3.3.1.GA/lib/antlr/antlr-2.7.6.jar
 559366  06-13-08 12:09   hibernate-distribution-3.3.1.GA/lib/commons-collections-3.1.jar
 313898  06-13-08 12:0[illegible]   hibernate-distribution-3.3.1.GA/lib/dom4j/dom4j-1.6.1.jar
  13236  06-13-08 12:0[illegible]   hibernate-distribution-3.3.1.GA/lib/required/jta-1.1.jar
  17384  08-19-08 19:40   hibernate-distribution-3.3.1.GA/lib/required/slf4j-api-1.5.2.jar
 471005  06-13-08 12:10   hibernate-distribution-3.3.1.GA/lib/required/javassist-3.4.GA.jar
      0  09-10-08 12:27   hibernate-distribution-3.3.1.GA/lib/optional/
      0  09-10-08 12:27   hibernate-distribution-3.3.1.GA/lib/optional/c3p0/
 608376  06-13-08 12:12   hibernate-distribution-3.3.1.GA/lib/optional/c3p0/c3p0-0.9.1.jar
      0  09-10-08 12:27   hibernate-distribution-3.3.1.GA/lib/optional/proxool/
 475943  06-13-08 12:12   hibernate-distribution-3.3.1.GA/lib/optional/proxool/proxool-0.8.3.jar
  .
  .
  .
  .
  .
  .
```

Holy Shnikeys!

Do I really need <u>all</u> of this stuff?!?

Next Steps

- Copy the jar files to your lib directory
  - Option 1: Copy Everything (Lazy)
  - Option 2: Copy only what you need (Tedious)
- Create your build.xml file

Create the build.xml

Start with an empty file

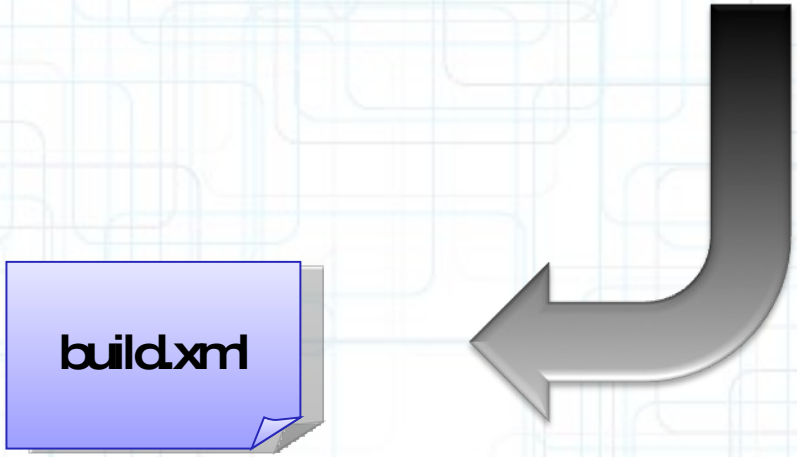
```
<?xml version="1.0"?>
<project name="antdemo" default="compile"
   basedir="." >

</project>
```

Create the build.xml

Create some variables

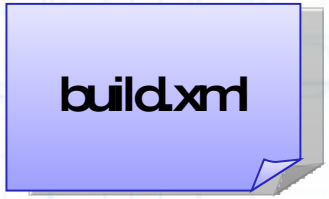
```
<!-- set global properties for this build -->
<property name="lib" value="lib"/>
<property name="src" value="src"/>
<property name="conf" value="conf"/>
<property name="target" value="target"/>
<property name="classes" value="${target}/classes"/>
```



Create the build.xml

Tell ant about all of your libraries

```
<fileset id="compile.libs" dir="${lib}">
        <include name="antlr-2.7.6.jar"/>
        <include name="asm.jar"/>
        <include name="asm-attrs.jar"/>
        <include name="cglib-2.1.3.jar"/>
        <include name="commons-collections-3.2.jar"/>
        <include name="commons-lang-2.3.jar"/>
        <include name="commons-logging-1.0.4.jar"/>
        <include name="dom4j-1.6.1.jar"/>
        <include name="ejb3-persistence.jar"/>
        <include name="javassist.jar"/>
        <include name="jboss-archive-browsing.jar"/>
        <include name="jdbc2_0-stdext.jar"/>
        <include name="jta.jar"/>
        <include name="xml-apis.jar"/>
        <include name="xercesImpl-2.6.2.jar"/>
        <include name="hibernate3.jar"/>
        <include name="hibernate-annotations.jar"/>
        <include name="hibernate-commons-annotations.jar"/>
        <include name="hibernate-entitymanager.jar"/>
</fileset>
```



Create the build.xml

The ant way

Step 4

Tell ant about compiling

```
<target name="init">
        <tstamp/>
        <mkdir dir="${classes}"/>
</target>

<target name="compile" depends="init">
        <mkdir dir="${basedir}/${classes}"/>
        <javac srcdir="${basedir}/${src}"
              destdir="${basedir}/${classes}"
              debug="yes"
           target="1.5">
                <classpath>
                        <fileset refid="compile.libs"/>
                </classpath>
        </javac>
        <copy file="${basedir}/${conf}/hibernate.cfg.xml"
             todir="${basedir}/${classes}"/>
</target>

<target name="clean">
        <delete quiet="yes" dir="${basedir}/${target}"/>
        <mkdir dir="${basedir}/${target}"/>
</target>
```

build.xml

Create the build.xml

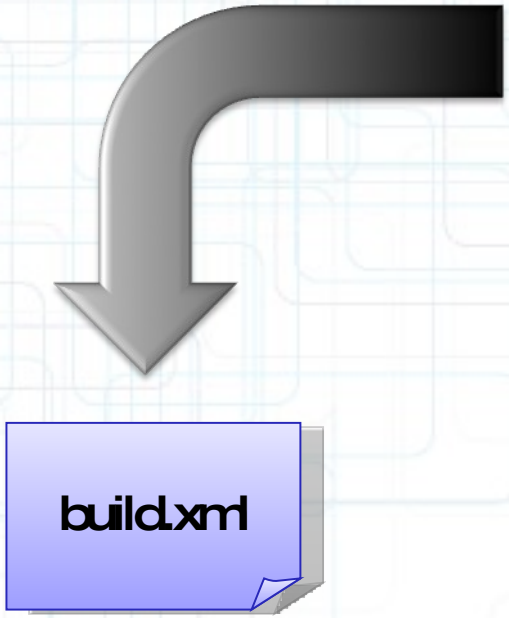
Step 5

The ant way

Tell ant how to run the app

```
<fileset id="runtime.libs" dir="${lib}">
        <include name="postgresql-8.2-507.jdbc3.jar"/>
</fileset>

<target name="run" depends="compile">
        <java classname="com.techmaine.antdemo.App">
                <classpath>
                        <fileset refid="compile.libs"/>
                        <fileset refid="runtime.libs"/>
                        <pathelement location="${basedir}/${classes}"/>
                </classpath>
        </java>
</target>
```



Recap

We downloaded jars and dependencies ourselves

We told ant

- The name of the jar file that we needed (Hibernate)
- All the dependent jar file names
- Where the jar files were located
- That it needed to compile java files
- Where the java files were located
- Where it should put the class files
- Where it should put the Hibernate configuration file
- How to run the application
- Where the jar and class files were located (again, this time for runtime)
- build.xml is 75 lines, but who's counting?

Next:

The

maven

way

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. Artifacts are *deployed* to remote repositories and *installed* into local 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=com.techmaine \
 -DartifactId=demo-mvn \
 -DpackageName=com.techmaine.mvndemo \
 -Dversion=1.0
```

Create a Project Directory

Plugin Name

```
mvn archetype:create \
 -DgroupId=com.techmaine \
 -DartifactId=demo-mvn \
 -DpackageName=com.techmaine.mvndemo \
 -Dversion=1.0
```

Create a Project Directory

Plugin Name

Goal

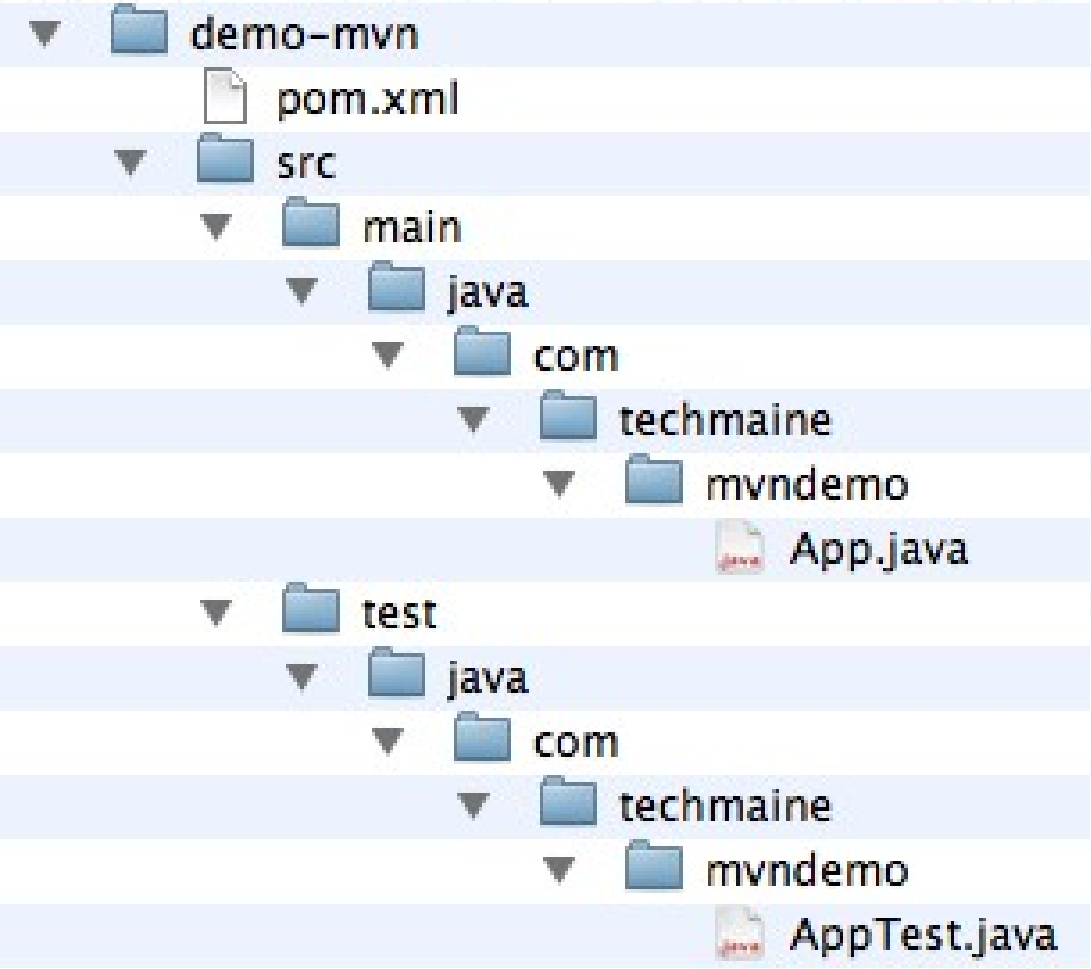
```
mvn archetype:create \
 -DgroupId=com.techmaine \
 -DartifactId=demo-mvn \
 -DpackageName=com.techmaine.mvndemo \
 -Dversion=1.0
```

Create a Project Directory

Step 1

The Maven Way

Voila! Look what maven has done for you:



Set up the dependencies

The Maven Way

Step 2

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

```
<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>com.techmaine</groupId>
  <artifactId>demo-mvn</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>3.8.1</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>org.hibernate</groupId>
    <artifactId>hibernate-annotations</artifactId>
    <version>3.3.1.ga</version>
</dependency>
```

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

Set up the compiler

STUPIDITY ALERT!

Maven assumes a default source version of 1.3. We need to tell it if we want 1.5. Here's a preview of plugin configuration:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.0.2</version>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Set up Hibernate Configuration

The Maven Way

Step 4

Create a resources directory beneath the main (and, optionally, test) directory, and put the Hibernate configuration file there.

- Files in the resources directory get copied to the root of the classpath when packaging occurs as part of the resource:resource goal (more on that later)
- The resources directories are automatically created for many of the archetypes, but not for the quickstart archetype that we used.

Package

Step 5

Next, package everything up before we run it.

The Maven Way

To do this, invoke maven thusly:

```
mvn package
```

This is an alternate way to invoke maven. Instead of specifying a plugin and goal, you specify a phase (in this case, package is the phase). A phase is a sequenced set of goals. The package phase compiles the java classes and copies the resources

Execute

Step 6

Next, use the exec plugin to run our application:

```
mvn exec:exec \
 -DmainClass=com.techmaine.mvndemo.App
```

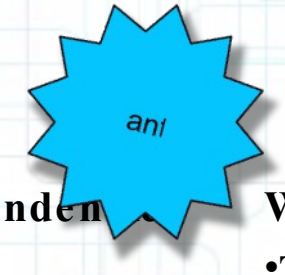
The Maven Way

Recap

We told maven

- That we were making a “quickstart” project.
- That we depended on Hibernate Annotations.
- That we needed Java 1.5
- pom.xml was 35 lines (would have been 22 if maven defaulted to Java 1.5 instead of 1.3)

Recap – Why Maven is Cool



We downloaded jars and dependen
ourselves

We told ant

- The name of the jar file that we needed (Hibernate)
- All the dependent jar file names
- Where the jar files were located
- That it needed to compile java files
- Where the java files were located
- Where it should put the class files
- Where it should put the Hibernate configuration file
- How to run the application
- Where the jar and class files were located (again, this time for runtime)
- build.xml is 75 lines, but who's counting?

We told maven

- That we were making a "quickstart" project.
- That we depended on Hibernate Annotations.
- That we needed Java 1.5
- pom.xml was 35 lines (would have been 22 if maven defaulted to Java 1.5 instead of 1.3)

What can Maven do?

When you first download it, almost nothing!

- Run goals
- Run phases (collections of goals)
- Download Dependencies*
- Download Plugins

* Actually, dependency downloads are done by a plugin, too.

But... from where?

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

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</artifact>
  <packaging>jar</packaging>
  <version>1.0.0</version>
  <name>Super Duper Amazing Deluxe Project</name>
  <modules>
    <!-- Sub-modules of this project -->
  </modules>
  <parent>
    <!-- Parent POM stuff if applicable -->
  </parent>
  <properties>
    <!-- Ad-hoc properties used in the build -->
  </properties>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```

General Information

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <name>Super Duper Amazing Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</artifact>
  <version>1.0.0</version>
  <modules>
    <!-- Sub-modules of this project -->
  </modules>
  <parent>
    <!-- Parent POM stuff if applicable -->
  </parent>
  <properties>
    <!-- Ad-hoc properties used in the build -->
  </properties>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```

Coordinates

Project Inheritance

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <name>Super Duper Amazing Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</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>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</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 log4j.
- Enforce plugin version across projects

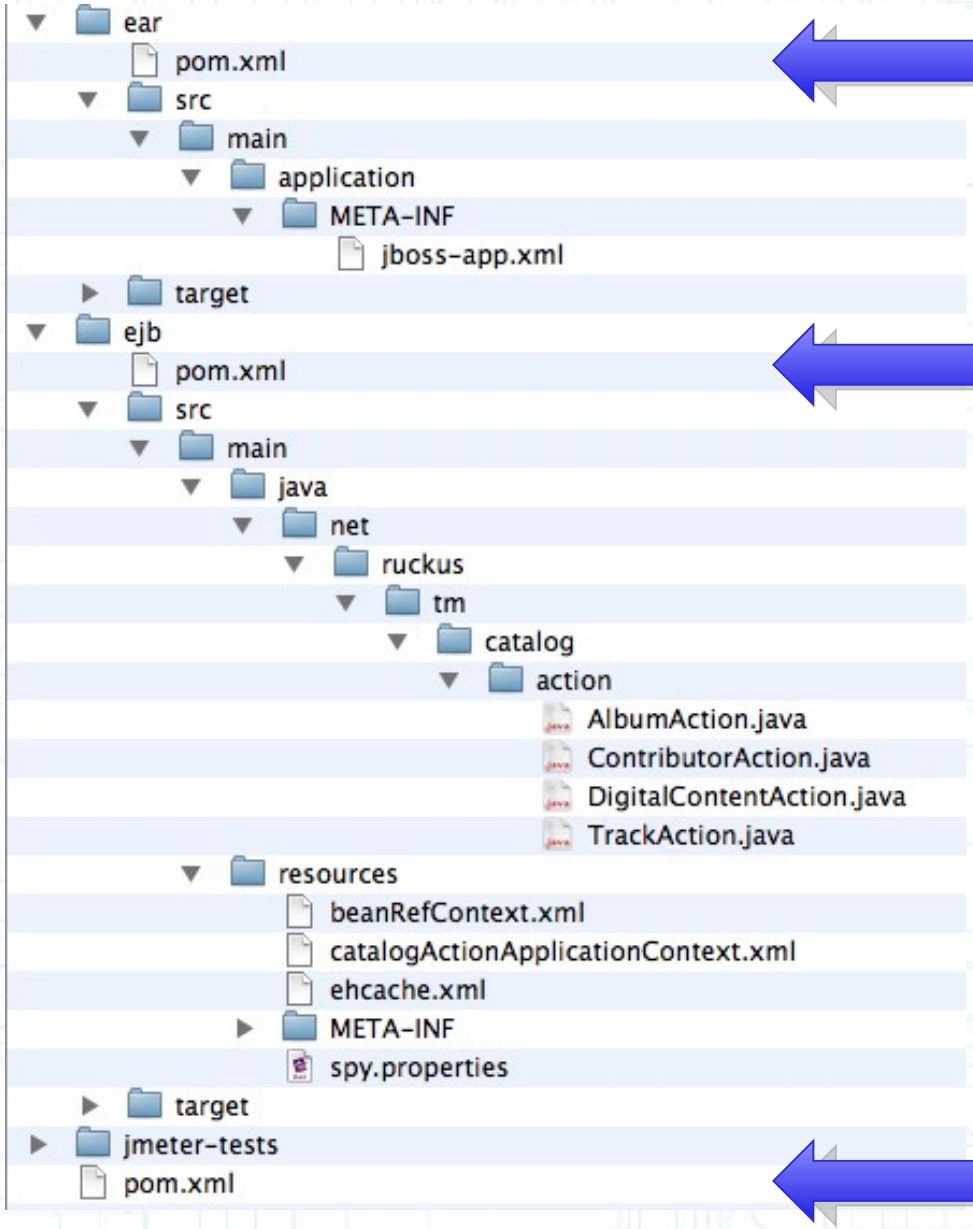
Multimodule Projects

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <name>Super Duper Amazing Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</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>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```

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 Projects



Depends on the EJB artifact

Creates EJB jar

```
<packaging>pom<packaging>
<modules>

    <module>ejb</module>
```

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 Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</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>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```

User-Defined Properties

- **User-Defined properties are like ant properties:**

```
<properties>
  <hibernate.version>3.3.0.ga</hibernate.version>
</properties>
...
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifact>
    <version>${hibernate.version}</version>
  </dependency>
</dependencies>
```

Example from *Maven: The Definitive Guide*, Sonatype, O'Reilly p.266

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

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <name>Super Duper Amazing Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</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>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```

Dependencies

Maven's ***pièce de résistance***

```
<dependencies>
  <dependency>
    <groupId>com.techmaine</groupId>
    <artifactId>awesome-lib</artifactId>
    <version>1.3.5</version>
    <scope>compile</scope>
    <optional>false</optional>
  </dependency>
</dependencies>
```

Dependencies

groupId and artifactId must be unique

```
<dependencies>
  <dependency>
    <groupId>com.techmaine</groupId>
    <artifactId>awesome-lib</artifactId>
    <version>1.3.5</version>
    <scope>compile</scope>
    <optional>false</optional>
  </dependency>
</dependencies>
```

Dependencies

```
<dependencies>
  <dependency>
    <groupId>com.techmaine</groupId>
    <artifactId>awesome-lib</artifactId>
    <version>1.3.5</version>
    <scope>compile</scope>
    <optional>false</optional>
  </dependency>
</dependencies>
```

- *1.3.5* - prefer version 1.3.5, newer version is acceptable to resolve conflicts
- *(1.3.4,1.3.9)* - Any version between 1.3.2 and 1.3.9, exclusive
- *[1.3.4,1.3.9]* - Any version between 1.3.2 and 1.3.9, inclusive
- *[,1.3.9]* - Any version up to, and including, 1.3.9
- *[1.3.5]* - Only version 1.3.5, do not use a newer version.

Dependencies

```
<dependencies>
  <dependency>
    <groupId>com.techmaine</groupId>
    <artifactId>awesome-lib</artifactId>
    <version>1.3.5</version>
    <scope>compile</scope>
    <optional>false</optional>
  </dependency>
</dependencies>
```

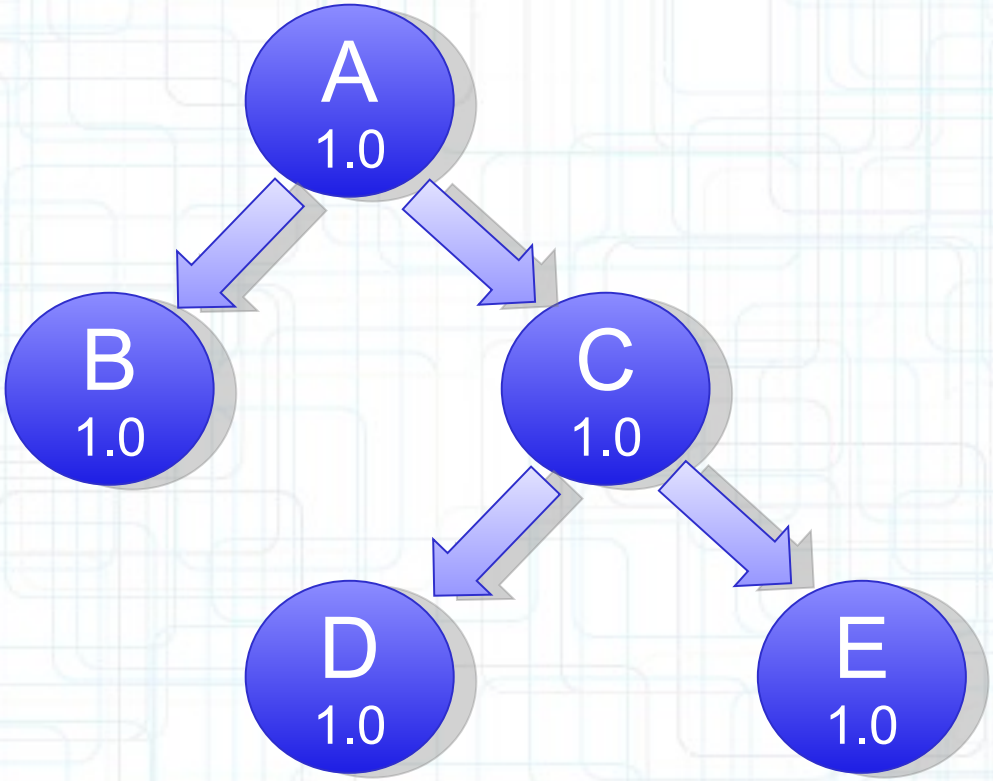
- *compile* - default, packaged. Available on compile-time and runtime CLASSPATH.
- *provided* - you expect the JVM or app container to provide the library. Available on compile-time CLASSPATH.
- *runtime* - needed to run, but not compilation (e.g., a JDBC driver)
- *test* - only needed during test execution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

Dependencies

```
<dependencies>
  <dependency>
    <groupId>com.techmaine</groupId>
    <artifactId>awesome-lib</artifactId>
    <version>1.3.5</version>
    <scope>compile</scope>
    <optional>false</optional>
  </dependency>
</dependencies>
```

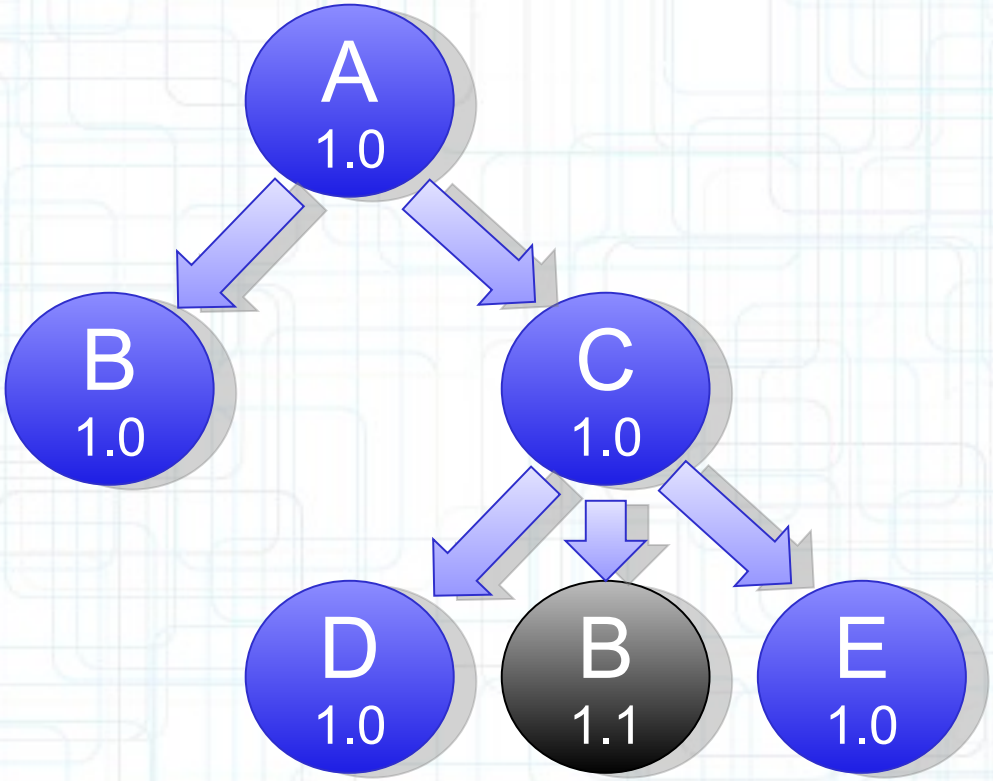
Prevents this dependency from being included as a transitive dependency if some other project depends on this project.

Transitive Dependencies



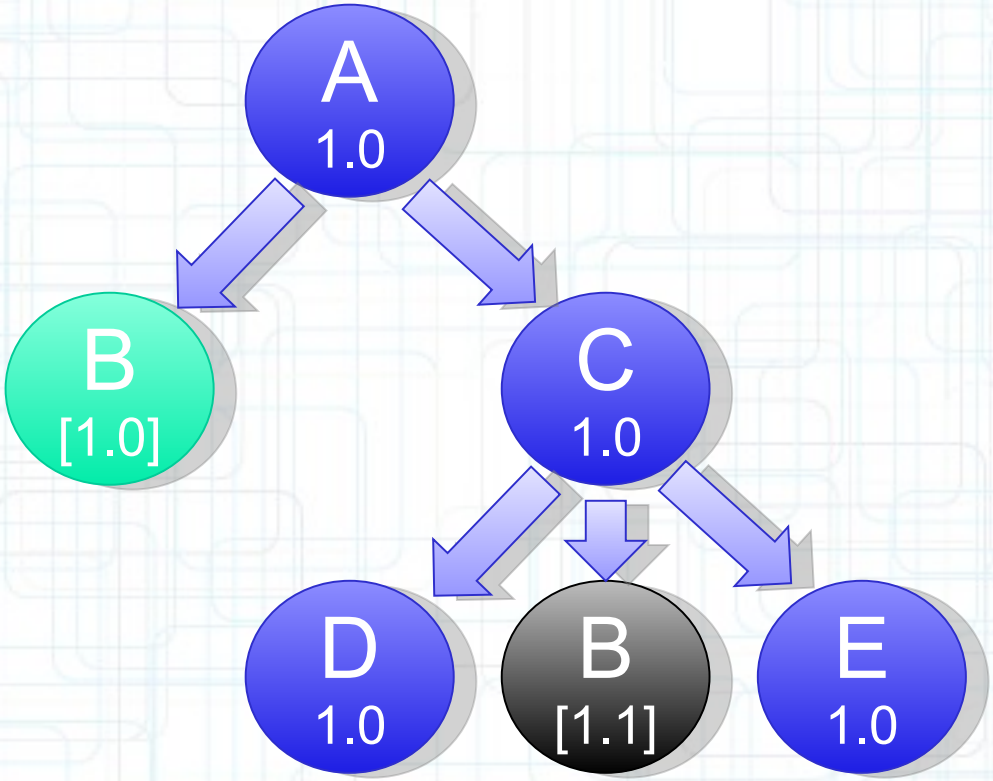
Our project (Project ÒAÓ) depends on B and C. Project C depends on projects D and E. Thus, our project depends on B, C, D, and E, and Maven will fetch and use these artifacts appropriately.

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

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

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <name>Super Duper Amazing Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</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>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```

Build Configuration

The build section of the POM, broken down further:

```
<project>
  ...
  <build>
    <filters>
      <filter>filter/my.properties</filter>
    </filters>
    <resources>
      ...
    </resources>
    <plugins>
      ...
    </plugins>
  </build>
  ...
</project>
```

Build Configuration Filters

```
<project>
  ...
  <build>
    <filters>
      <filter>filter/my.properties</filter>
    </filters>
    <resources>
      ...
    </resources>
    <plugins>
      ...
    </plugins>
  </build>
  ...
</project>
```

Path to a properties file (name=value). When the resources are processed during packaging, maven will substitute any ${name} strings with the corresponding value from the properties file.

Build Configuration Resources

- The resources:resources goal copies files from the resources directory to the output directory
- Can process using filters
- Default location src/main/resources
- Can be further configured:

```
<resources>
  <resource>
    <directory>src/main/scripts</directory>
    <filtering>true</filtering>
    <targetPath>bin</targetPath>
    <includes>
      <include>run.bat</include>
      <include>run.sh</include>
    </includes>
  <resource>
</resources>
```

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>
```

Plugin Configuration: Example

Below, we have configured the clean plugin to delete files ending in .txt from the tmp directory

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-clean-plugin</artifactId>
      <version>2.2</version>
      <configuration>
        <filesets>
          <fileset>
            <directory>tmp</directory>
            <includes>
              <include>**/*.txt</include>
            </includes>
            <followSymlinks>false</followSymlinks>
          </fileset>
        </filesets>
      </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

Core Plugins, cont.

Maven Plugin Whirlwind Tour

- **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

Maven Plugin Whirlwind Tour

- **archetype** - builds skeleton of a working project for many different frameworks
  - Wicket, Tapestry 5, 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>com.techmaine</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 type. E.g., for projects that package WARs:

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 xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <name>Super Duper Amazing Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</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>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- 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.h
    ome>
      </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>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
              <debug>false</debug>
              <optimize>true</optimize>
            </configuration>
          </plugin>
        </plugins>

    <appserver.home>/path/to/dev/appserver</appserver.h
    ome>
      </build>
    </profile>
...
```

Example from *Maven: The Definitive Guide,* Sonatype, O'Reilly p.200

Activating a Profile

- **On the command-line:**

```
mvn package
-P myprofile1,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>1.5</jdk>
        <os>
          <name>Windows XP</name>
          <family>Windows</family>
          <arch>x86</arch>
          <version>5.1.2600</version>
        </os>
        <property>
          <name>mavenVersion</name>
          <value>2.0.9</value>
        </property>
        <file>
          <exists>file2.properties</exists>
          <missing>file1.properties</missing>
        </file>
      </activation>
      ...
    </profile>
  </profiles>
</project>
```

Example from *Maven: The Definitive Guide,* Sonatype, O'Reilly p.204-205

Sometimes Maven Sucks

…returns ÒabOut 128,000Ó results

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

(FIN)