Maven II
(Maven Advanced)

"Code with Passion"

Topics covered in “Maven Advanced”

- Multi-module project
- Grouping Dependencies
- Profiles
- Dependency management
- Site generation

Multi-Module Project

What is a Multi-Module Project?

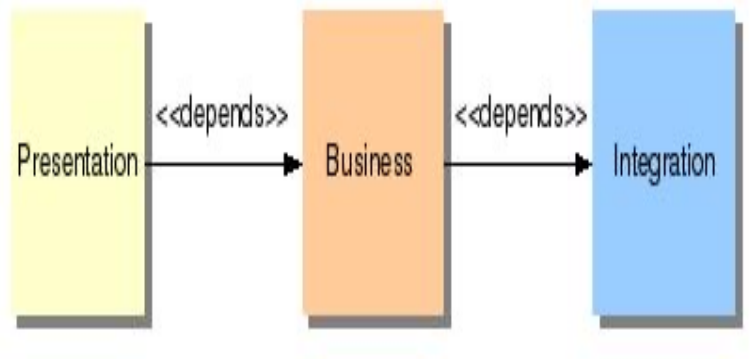
- A multi-modules project is a very particular type of project - it doesn't produce any artifact and is composed of several other projects known as Modules
  - These modules are called child projects while the multi-module project is called parent project

- When you run a Maven command on the multi-module parent project, it will execute it on all of its child projects

- Maven is able, through its reactor component, to discover the correct execution order and to detect circular dependencies

Projects have dependencies

- Imagine you are working on a project based upon a traditional 3 layered architecture, in which the layers are named presentation, business and integration
- Presentation layer has a dependency on business layer, which in turn has a dependency on integration layer – integration layer has to be built before business layer and business layer has to be built before presentation layer



Your project setup

Manual ordering of projects is hard

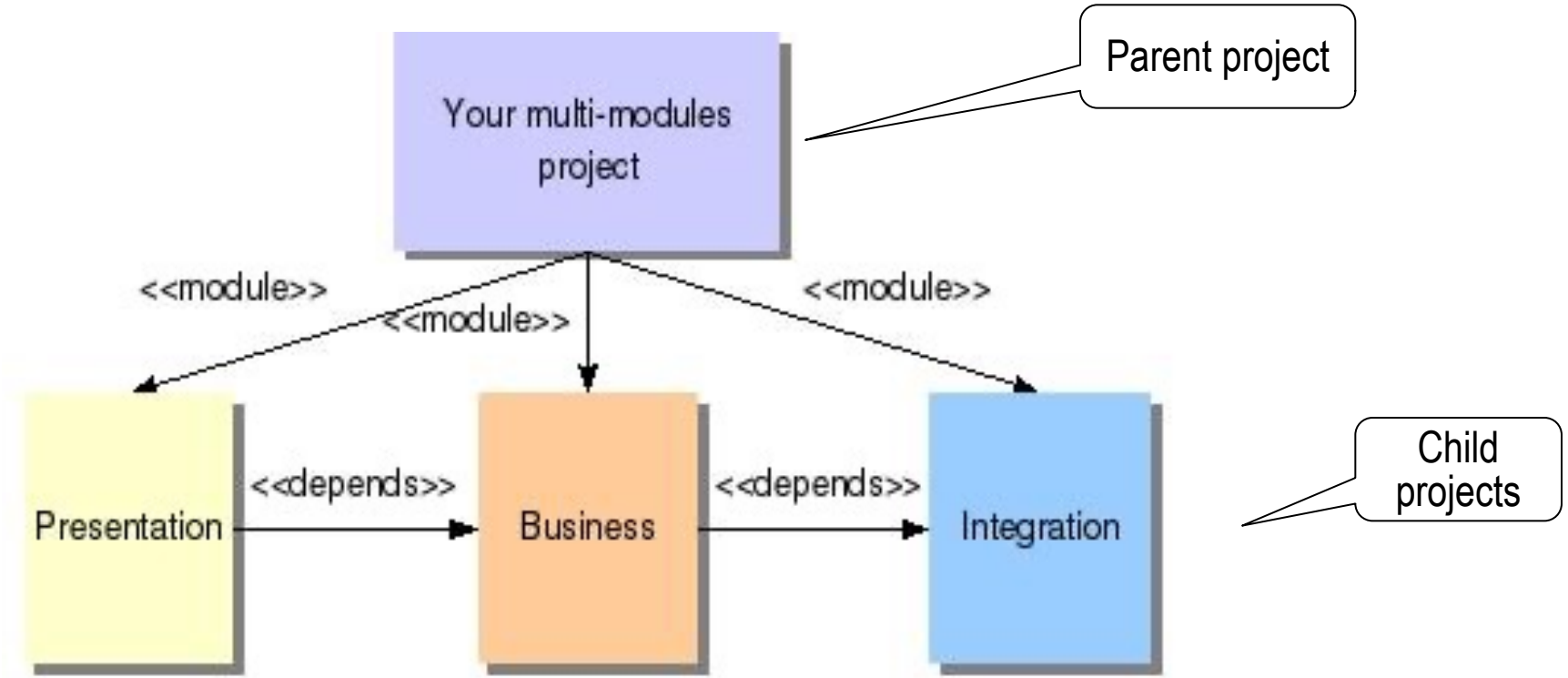
- So to compile your presentation project, you would first need to compile the business project.
- The business project depends upon the integration project so it should be compiled first.
- For example, you would have to control the order yourself as following

  `/root_directory$ cd integration`
  `/root_directory/integration$ mvn compile`
  `/root_directory/integration$ cd ..`
  `/root_directory$ cd business`
  `/root_directory/business$ mvn compile`
  `/root_directory/business$ cd ..`
  `/root_directory$ cd presentation`
  `/root_directory/presentation$ mvn compile`
- Use of multi-module Maven handles the ordering automatically

Steps for Creating Multi-module Maven Project

- Step #1: Structure the Maven project as
  - Parent project
  - Child projects
- Step #2: Write parent POM
- Step #3: Create child project directories
- Step #4: Write child POM's

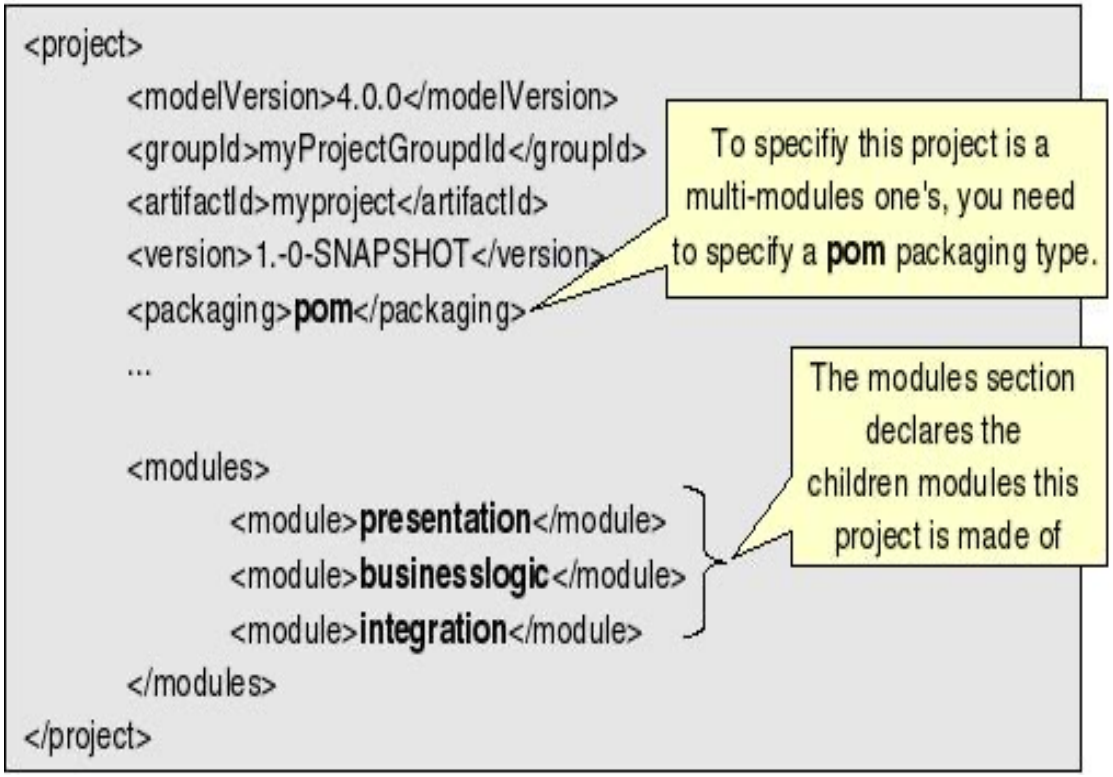
Multi-Module Maven Project to the Rescue



Your multi-module project setup

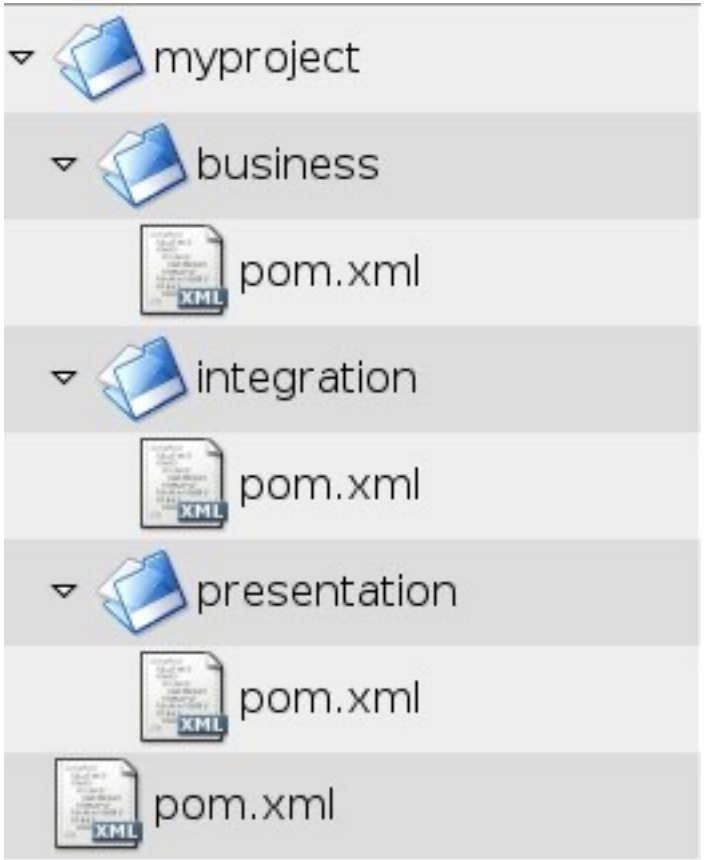
Step #2: Write Parent POM

- First, create a new directory named after your multi-module parent project, something like “myproject”. Now let's write the pom file for your multi-modules parent project



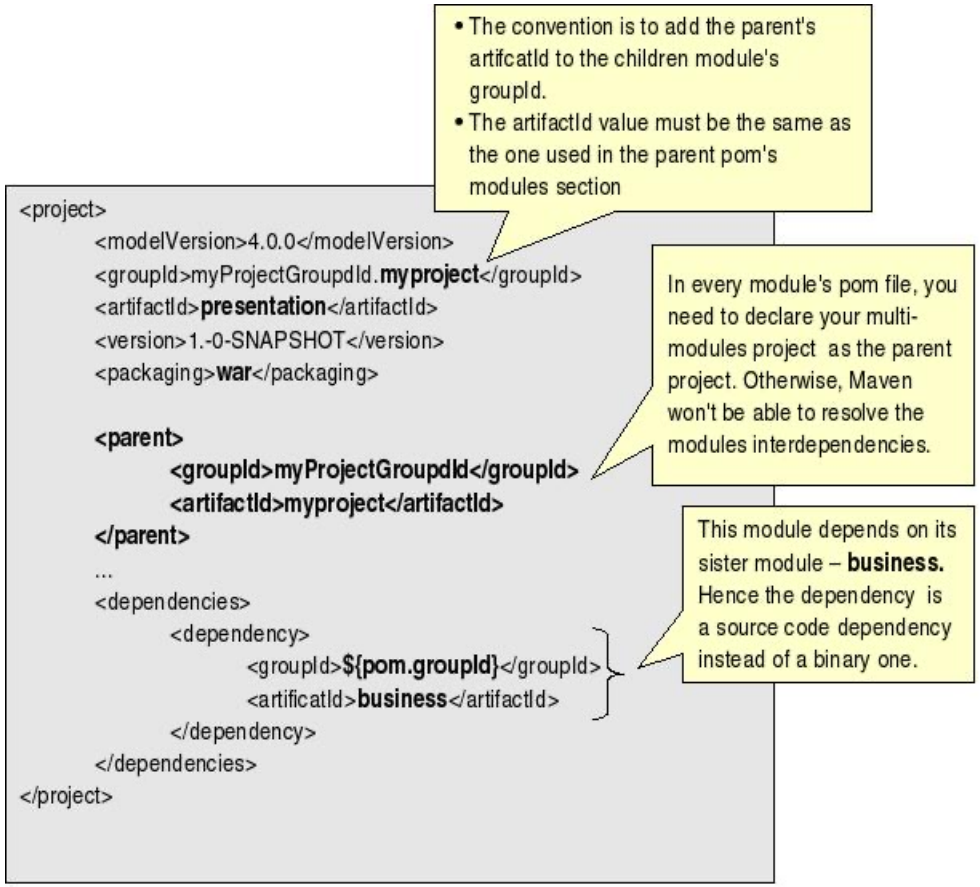
Step #3: Create Child Project Directories

- “myproject” parent directory contains child project directories - ”business”, “integration”, and “presentation”



Step #4: Write Child POM's

- Write “pom.xml” for child projects



Grouping Dependencies

Grouping Dependencies

- If you have a set of dependencies which are logically grouped together, you can create a project with “pom” packaging that groups dependencies together
- Example scenario
  - Let’s assume that your application uses Hibernate. Every project which uses Hibernate might also have a dependency on the Spring Framework and a MySQL JDBC drive
  - Instead of having to include these dependencies in every project that uses Hibernate, Spring, and MySQL you could create a special POM that does nothing more than declare a set of common dependencies
  - You could create a project called “persistence-deps” (short for Persistence Dependencies), and have every project that needs to do persistence depend on this convenience project

pom.xml of “persistence-dep” Project

```
<project>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>persistence-deps</artifactId>
    <version>1.0</version>
    <packaging>pom</packaging>
    <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate</artifactId>
            <version>${hibernateVersion}</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-annotations</artifactId>

<version>${hibernateAnnotationsVersion}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-hibernate3</artifactId>
            <version>${springVersion}</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>${mysqlVersion}</version>
        </dependency>
    </dependencies>
    <properties>
        <mysqlVersion>(5.1,)</mysqlVersion>
        <springVersion>(2.0.6,)</springVersion>
        <hibernateVersion>3.2.5.ga</hibernateVersion>

<hibernateAnnotationsVersion>3.3.0.ga</hibernateAnnotatio
nsVersion>
    </properties>
</project>
```

pom.xml of a project which depends

```
<project>
   <description>This is a project requiring JDBC</description>
   ...
   <dependencies>
      ...
      <dependency>
         <groupId>org.sonatype.mavenbook</groupId>
         <artifactId>persistence-deps</artifactId>
         <version>1.0</version>
         <type>pom</type>
      </dependency>
   </dependencies>
</project>
```

Profiles

What is and Why use Profile?

- A profile is an alternative set of configuration values which set or override default values
- Profiles allow for the ability to customize a particular build for a particular environment
  - Profiles enable portability between different build environments
- Usage examples of profiles
  - Production and Development environments use different configuration – use production database for Production while use development database for development
  - Customize the build depending the Operating System or the installed JDK version

How to Create and Use a Profile?

- Profiles are configured in the pom.xml and are given an identifier
- You can choose a particular profile (use a profile for Maven build) in two ways
  - Through command-line selection
  - Through activation (this is more common)

Through Command-line Selection

- The following pom.xml uses a *production* profile to override the default settings of the Compiler plugin

```
<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>
        </build>
    </profile>
</profiles>
```

- Then run "*mvn clean install -Pproduction*"

Profile Activation Schemes

- Activation By Default
  - <activeByDefault>true</activeByDefault> for a particular profile
- <activeProfiles>
  - List the profiles that should be active for all builds
- Through a system property
  - Either matching a particular value for the property, or merely testing its existence.
- Through JDK version prefix or Operating system
  - For example, a value of '1.4' might activate a profile when the build is executed on a JDK version of '1.4.2_07'

Activation By Default

- The *activeByDefault* element controls whether this profile is considered active by default.

```
<profile>
    <id>dev</id>
    <activation>
        <activeByDefault>true</activeByDefault>
            <jdk>1.5</jdk>
                <os>
                    <name>Windows XP</name>
                        <family>Windows</family>
                        <arch>x86</arch>
                        <version>5.1.2600</version>
                </os>
                <property>
                    <name>customProperty</name>
                        <value>BLUE</value>
                </property>
                <file>
                    <exists>file2.properties</exists>
                        <missing>file1.properties</missing>
                </file>
    </activation>
    ...
</profile>
```

activeProfiles

- List of active profiles

```
<activeProfiles>
    <activeProfile>alwaysActiveProfile</activeProfile>
    <activeProfile>anotherAlwaysActiveProfile</activeProfile>
</activeProfiles>
```

Through a System Property

- Activated when “target-env” property is set to “dev”
- When <value>..</value> is not specified, any value is qualified

```
<profile>
  <id>env-dev</id>

  <activation>
    <property>
      <name>target-env</name>
      <value>dev</value>
    </property>
  </activation>

  <properties>
    <tomcatPath>/path/to/tomcat/instance</tomcatPath>
  </properties>
</profile>
```

Through JDK Version

- Provide customizations based on variables like operating systems or JDK version
- Example: dynamic Inclusion of submodules using Profile Activation: the “simple-script” module will be built only when JDK 1.6 is used for build

```
<profiles>
    <profile>
      <id>jdk16</id>
      <activation>
          <jdk>1.6</jdk>
      </activation>
      <modules>
          <module>simple-script</module>
      </modules>
    </profile>
 </profiles>
```

Types of Profiles

- Per project
  - Define in the POM itself (pom.xml)
- Per user
  - Defined in the Maven-settings (%USER_HOME%/.m2/settings.xml).
- Global
  - Defined in the global Maven-settings (%M2_HOME%/conf/settings.xml).

Listing Active Profiles

- To make it easier to keep track of which profiles are available, and where they have been defined, the Maven Help plugin defines a goal, active-profiles

```
$ mvn help:active-profiles
Active Profiles for Project 'My Project':

The following profiles are active:

- my-settings-profile (source: settings.xml)
- my-external-profile (source: profiles.xml)
- my-internal-profile (source: pom.xml)
```

Dependency Management

Dependency Management

- Allows to consolidate and centralize the management of dependency versions without adding dependencies which are inherited by all children. This is especially useful when you have a set of projects (i.e. more than one) that inherits a common parent.
  - dependencyManagement is used to pull all the dependency information into a common POM file, simplifying the references in the child POM file
- Control of versions of artifacts used in transitive dependencies.
- Useful when you have multiple attributes that you don't want to retype in under multiple children projects.
- Used to define a standard version of an artifact to use across multiple projects.

Dependency Management

- Parent project defines a dependency with a common version

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

- Then child projects specify the dependency without a version

```
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
    </dependency>
  </dependencies>
```

Site Generation

What Maven Site Plug-in for?

- The Site Plugin is used to generate a site for the project. The generated site also includes the project's reports that were configured in the POM

Site Plugin Goals

- site:site
- site:deploy
- site:run
- site:stage
- site:stage-deploy

Configuring Reports

- Maven has several reports that you can add to your web site to display the current state of the project. These reports take the form of plugins, just like those used to build the project
- There are many standard reports that are available by extracting information from the POM. Currently these are provided by default
  - Continous Integration
  - Dependencies
  - Issue Tracking
  - License
  - Mailing Lists
  - Project Team
  - Source Repository

Configuring Reports

- To add these reports to your site, you must add the plugin to the <reporting> element in the POM.

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <version>2.1.1</version>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

Code with Passion!