Maven 3

Text: EN

Software: EN

REALDOLMEN
to get there, together

Table of contents

1. Managing Project Builds With Maven

1.1. Managing Project Builds With Maven

Learn how to leverage Maven for better, more efficient and standardized Java project builds

1.2. Agenda

- Introduction
- Getting started
- Maven build lifecycle
- Dependency management
- Project testing
- Multi-module projects
- Release management
- Profiles
- Archetypes
- Deployment
- Final thoughts

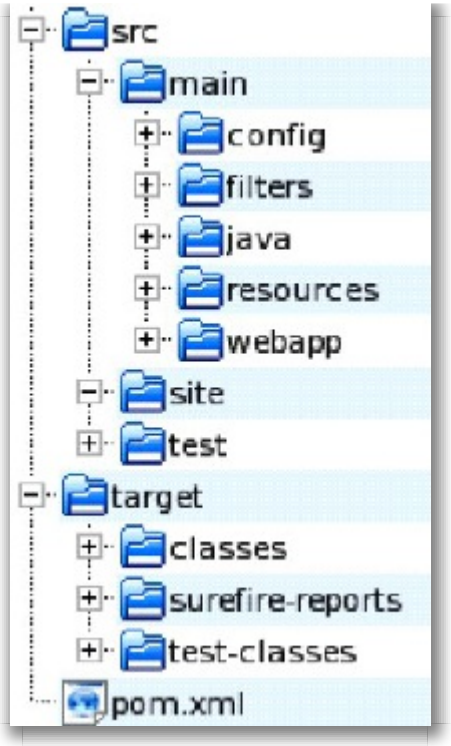
2. Introduction

2.1. What is Maven?

- Maven is
  - A build tool
    - A tool used to build deployable artifacts from source code
  - A project management tool
    - Run reports, generate a web site, facilitate communication among members of a working team,
  - An abstract container for running build tasks
- Indispensable for enterprise projects
  - Consistent management and builds of large collections of interdependent modules and libraries
  - Removes much of the burden of 3rd party dependency management
  - Relieves the effort required to build and maintain software

2.2. Maven principles

- Convention over configuration
  - Maven provides sensible default behavior for projects
  - Maven assumes a specific project structure
    - Source code in `${basedir}/src/main/java`
    - Resources in `${basedir}/src/main/resources`
    - Tests in `${basedir}/src/test`
    - A project is assumed to produce a JAR file
    - Compile byte code to `${basedir}/target/classes`
    - Create a distributable JAR file in `${basedir}/target`

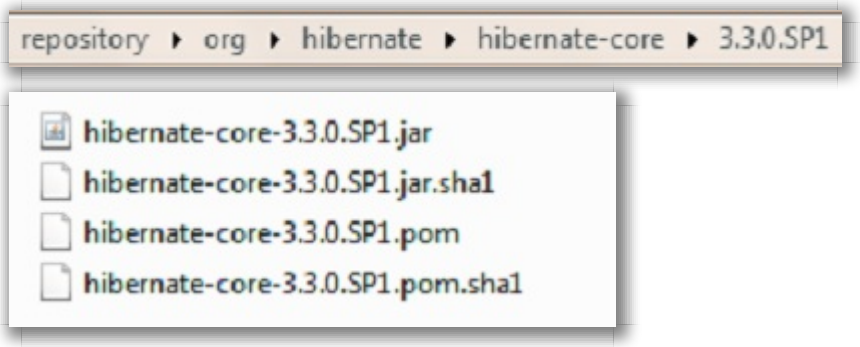


- Convention over configuration
  - Maven has default core plugins
    - Conventions for compiling, packaging, generating web sites, and many other processes
  - If you follow the conventions, Maven will require almost zero effort
    - Just put your source in the correct directory, and Maven will take care of the rest
- A common interface
  - No more need for asking questions when checking out software
    - What does the project need to build?
    - What libraries do I need to download?
    - Where do I put them?
    - What goals can I execute in the build?
  - You check it out from source, and you run `mvn install`
- Universal reuse through plugins
  - Maven delegates most responsibilities to a set of Maven plugins
    - Those affect the Maven Lifecycle and offer access to goals
    - Compiling source, packaging bytecode, publishing sites, and any other task which need to happen in a build
- All Maven magic happens in the Maven plugins
  - Maven retrieves both dependencies and plugins from the remote repository to allow for universal reuse of build logic

- If a plugin has improved, you only need to change your `pom.xml`
- Project conceptual model
  - Dependency management
    - Use a group identifier, an artifact identifier, and a version to declare dependencies
  - Remote repositories
    - Use the coordinates defined in the Project Object Model (POM) to create repositories of Maven artifacts
  - Universal reuse of build logic
    - Plugins contain logic that works with the POM
  - Tool portability / integration
    - IDE specific project files can easily be generated from the model
  - Easy searching and filtering of project artifacts
    - Tools like Nexus allow to index and search the contents of a repository using the information from the POM

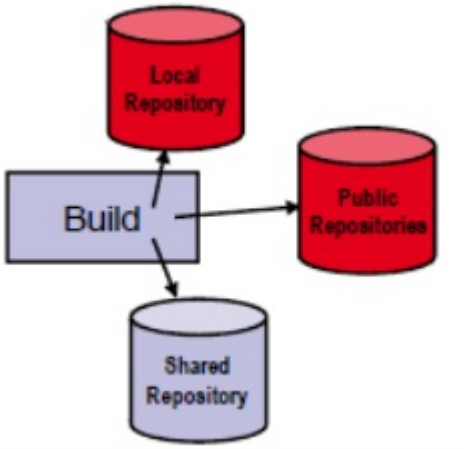
2.3. Repository

- Maven uses repositories to store all kind of artifacts
  - JARs, EARs, WARs, EJBs, ZIPs, plugins, ...
  - This means no need to put jars in the project's versioning system!
- All project interactions go through the repository
  - No need to add relative paths
  - Easy to share between teams
- Every repository has the same hierarchical structure



- Local repository
  - `/Users/<username>/.m2/repository`
- Remote repositories (public)

- Central: http://repo1.maven.org/maven2 (default)
- Codehaus: http://repository.codehaus.org (mojo's)
- Other repositories exist for specific frameworks / libraries
- Repository management using Nexus offers additional benefits



- Shared repository (on the company network)
  - For non-redistributable artifacts
    - Manual deployment
  - Share internal artifacts within team
- Repositories are set up in `settings.xml`
  - This allows you to change the location of the local repository (e.g. to avoid anti-virus scans)

2.4. Basic configuration using the POM

- Single `pom.xml` file
  - Provide information about what is being built and not how
  - Contains information required for the build such as
    - Versioning and configuration management
    - Project structure
    - Dependencies
    - Repositories
    - Application and testing resources
    - Reports
    - Plugins
    - ...

2.5. A simple `pom.xml`

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.realdolmen.maven</groupId>
  <artifactId>introduction</artifactId>
  <version>1.0</version>
</project>
```

- That is all you need!
  - Running the command `mvn install` which will
    - Process resources
    - Compile source
    - Execute unit tests
    - Create a JAR
    - Install the JAR in a local repository for reuse
  - Running `mvn site` will
    - Create an `index.html` file in `target/site` containing links to JavaDoc and reports
- Customizing will let the `pom.xml` grow in size

2.6. Configuring in the `pom.xml`

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <!-- ... -->
  <build>
    <directory>target</directory>
    <outputDirectory>target/classes</outputDirectory>
    <finalName>${artifactId}-${version}</finalName>
    <testOutputDirectory>target/test-classes</testOutputDirectory>
    <sourceDirectory>src/main/java</sourceDirectory>
    <scriptSourceDirectory>src/main/scripts</scriptSourceDirectory>
    <testSourceDirectory>src/test/java</testSourceDirectory>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
      </resource>
    </resources>
    <testResources>
      <testResource>
        <directory>src/test/resources</directory>
      </testResource>
    </testResources>
  </build>
  <!-- ... -->
</project>
```

2.7. Why prefer Maven over ...

- ... an IDE project?
  - Not a reproducible process
  - Hard to version control
  - Might be useful for a quick proof of concept / demo application
  - Only works on your own workstation
- Ant?
  - 100-1000 lines of build scripts per project
  - A lot of rework for new every project
  - No standard build lifecycle
  - Hard to maintain/improve
  - Ant scripts become complex

2.8. m2e Eclipse Plugin

- Improved Maven2 experience inside Eclipse
  - Launches Maven builds from within Eclipse
  - Synchronizes Eclipse build path and POM dependencies
  - Resolves Maven dependencies from projects in workspace
  - Automatically downloads dependencies
  - Searches dependencies in remote repositories
- Can be installed through an update
- Most IDEs have Maven support built in

3. Getting started

3.1. Installing Maven

- Maven requires Java (set the `JAVA_HOME`)
  - Preferably use Java 5 or newer
- Download the latest version
  - http://maven.apache.org/download.html
- Unzip the Maven distribution to a location of your choice
  - `C:\Program Files` or `C:\`
  - Maven is located in a subdirectory such as `apache-maven-X.X.X`
  - Add `MAVEN_HOME` and add the `bin` subdirectory to your `PATH` variable
- Verify the installation `mvn --version`
- Maven is ready to use
  - Maven uses a direct Internet connection by default
  - Setting up a repository manager on your network or locally can allow you to work without Internet connection

---

3.2. Configuring the `settings.xml`

- Located in `/conf` directory
  - Needed when behind a firewall to configure the HTTP proxy
  - Follow the template for instructions
  - The file can also be copied to your home directory in the local repository (`/Users/<username>/.m2`)
    - This allows you to reuse the settings when installing a different version of Maven

You will also add passwords in the `settings.xml` file, specific to your user. To avoid the possibility that someone could read your password, you better encrypt it, using the following command:

```
mvn --encrypt-master-password
```

This will prompt you for your password. The encrypted result can then be copied to your `settings.xml` file.

3.3. Starting a new project

- Maven makes it simple to start a project using archetypes
  - An archetype is a template for a module (JAR, WAR or more complex templates for

popular frameworks)

- To start the project, run from an empty directory
  - Maven will download the archetype functionality and missing Java libraries and store them in your local repository

```
mvn archetype:generate
```

- You will then be prompted with a long list to select an archetype to generate the project from

```
Choose archetype:
[...]
164: internal -> maven-archetype-quickstart
[...]
167: internal -> maven-archetype-webapp
[...]
Choose a number: 164:
```

- The default is `maven-archetype-quickstart`
  - This is archetype that is used in Maven's getting started guide
  - This is a good choice for a minimal Maven project or an archetype from which new modules can be built (JAR projects)
- For a simple Java web application choose `maven-archetype-webapp`
- The archetype plugin prompts you for coordinates to add to the project
  - These are called group ID, artifact ID and version (GAV)
    - Group ID: an identifier for a collection of related modules, usually starting with an organization, then to projects and sub-projects, structured like the Java package system
    - Artifact ID: a unique identifier for a given module within a group
    - Version: used to identify the release or build number of the project
- Fill in the values for these coordinates

```
Define value for property 'groupId': : com.realdolmen.maven
Define value for property 'artifactId': : gettingstarted
Define value for property 'version': 1.0-SNAPSHOT: : 1.0-SNAPSHOT
Define value for property 'package': : com.realdolmen.gettingstarted
```

- The final prompt for a package defines the Java package
  - This is the package that will be used for source code
  - This is not the packaging type (that is provided by the archetype)
- Notice the use of SNAPSHOT
  - In Maven, there are two kinds of versions
    - Release: is assumed never to change, so use them for a single state of the

project when it is released
    - Snapshot: used during development to indicate to Maven that development is still occurring and that the project may change

- Confirm that the values are as you intended
- Maven will then create the project
- You should see the following banner

```
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------
```

- The project will be stored in a subdirectory `gettingstarted`

```
`-- gettingstarted
    |-- pom.xml
    `-- src
        `-- main
        |-- resources
        `-- webapp
            |-- WEB-INF
            | `-- web.xml
            `-- index.jsp
```

- The following `pom.xml` will be created

```
<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.rd.gettingstarted</groupId>
  <artifactId>gettingstarted</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>getting-started Maven Webapp</name>
  <build>
    <finalName>gettingstarted</finalName>
  </build>
</project>
```

- This `pom.xml` includes some default Maven conventions
    - Default build paths

- These settings can be customized
    - Following the conventions make builds simpler and more familiar

3.4. Building the project

- The build is run using the build lifecycle
  - This is a predefined, ordered set of steps
  - The individual steps are called phases
  - The same phases are run for every Maven build using the default lifecycle, no matter what it will produce
  - The build tasks that will be performed during each phase are determined by the configuration in the project file
- To build the project, run the following command from the directory that contains the `pom.xml` file: `mvn package`
  - `package` is just one of the build lifecycle phases
  - Maven will download a few jars from the repository and start building
  - The result:
    - `target/gettingstarted` subdirectory that contains the exploded war
    - `target/gettingstarted.war` file

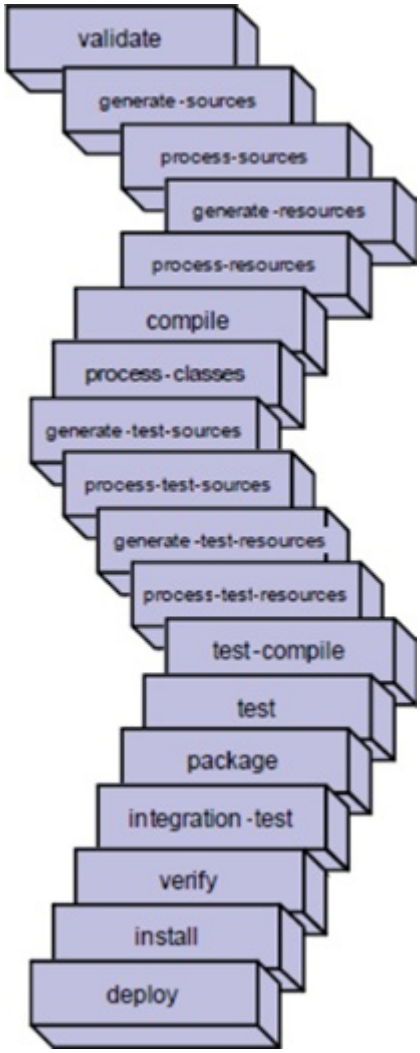
3.5. Exercise: Getting started

- Check your Maven installation
- Create a new project using Maven quick start
- Compile the project and notice the generated artifacts
  - Also notice the default download
- Package and run the project using the command line
- Clean up the project, and repeat the previous steps again
- Create, compile, and package a new web application project
- Run the project on Tomcat

4. Maven build lifecycle

4.1. Lifecycle phases

- Builds in Maven follow the build lifecycle
- Advantages
  - New developers do not need to learn a new build process
  - It is easy to incorporate new tools into the lifecycle by binding them to phases
- The commonly used phases are
  - `validate`, `compile`, `test`, `package`, `install`, `deploy`
- Phases run in sequence and depend on previous phases
  - If you run the `test` phase, `validate`, `compile` and intermediate phases will run to make sure the code is up to date before running the tests
- The following image contains most of the Maven build lifecycle phases



4.2. Goals

- When Maven builds a project
  - Maven steps through the phases
  - Any goals which may have been registered with each phase will be executed
- Goals are tasks that are attached to the build lifecycle to give the build substance
  - Instead of invoking tasks, plugins or developers attach goals to the default build lifecycle
- Maven knows which goals to run based on the project packaging
  - For a web project: include resources, compile Java source, compile tests, execute tests and then package goals
  - Example goals: [resources:resources], [compiler:compile], [resources:testResources], [compiler:testCompile], [surefire:test], [war:war]
  - You can see these goals when performing the Maven build

4.3. Common lifecycle goals

- Many of the different packaging lifecycles have similar goals
  - Process resources
  - Compile
  - Process test resources
  - Test compile
  - Test
  - Install
  - Deploy

4.4. Process resources

- The process resources phase “processes” resources and copies them to the output directory
  - Has the `resources:resources` goal
  - Default behavior
    - Copy the files from `src/main/resources` to `target/classes`
  - Maven can also apply a filter to the resources
    - This allows you to replace tokens within your resources file
    - Use the `${ }` syntax
    - In combination with build profiles, this allows you to produce build artifacts which target different deployment platforms
    - See example on the next slide
  - You can also change the default resources directory, or even configure additional

resources directories

- The resource: `src/main/resources/META-INF/service.xml`

```
<service>
  <url>${jdbc.url}</url>
  <user>${jdbc.username}</user>
  <password>${jdbc.password}</password>
</service>
```

- The variables: `src/main/filters/default.properties`

```
jdbc.url=jdbc:hsqldb:mem:mydb
jdbc.username=sa
jdbc.password=
```

- Configuring the filter: `pom.xml`

```
<build>
  <filters>
    <filter>src/main/filters/default.properties</filter>
  </filters>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
```

4.5. Compile

- The compile phase compiles all of the source code and copies the bytecode to the build output directory
  - Has the `compiler:compile` goal
  - Default behavior
    - Compile everything from `src/main/java` to `target/classes`
    - The compiler plugin calls javac and uses default source and target settings of 1.5 and 1.5
  - It is possible to change these settings in the `pom.xml`

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.5.1</version>
      <configuration>
```

```
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

4.6. Process test resources

- Is almost the same as process resources
  - There are just some trivial differences in the `pom.xml`
  - Default behavior

```
- Copy the files from `src/test/resources` to `target/test-classes`
```

  - Maven also allows you to add a filter to the test resources, just as with regular resources

4.7. Test compile

- Is almost identical to the compile phase
  - The difference is that test compile is invoking `compiler:testCompile`
  - Default behavior

```
- Compile the source in `src/test/java` to `target/test-classes`
```

  - You can customize the location of these test source code and output in the `pom.xml`

4.8. Test

- In the test phase, the goal of the Surefire plugin is called
  - Surefire is the Maven unit testing plugin
  - Default behavior
    - Look for all classes ending in -Test in the test source directory and run them as JUnit tests
    - Can also be configured to run TestNG unit tests
  - After running the command `mvn test`
    - Surefire produces reports in `target/surefire-reports`
    - Two files for each test: an XML containing execution information and a text file containing the output
    - If there is any problem and a unit test has failed, you can use the contents of

this directory to track down the cause

- Surefire stops running from the moment a unit test failure is encountered
- If you want to continue a build, even in case of failures, you have to override its behavior

- Ignoring test failures

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.19.1</version>
      <configuration>
        <testFailureIgnore>true</testFailureIgnore>
      </configuration>
    </plugin>
  </plugins>
</build>
```

- Skipping tests altogether
  - Note that this affects the Surefire, Failsafe and Compiler plugins

```
mvn install -Dmaven.test.skip=true
```

- Alternative is to use `-DskipTests`
  - This only affects Surefire

```
mvn install -DskipTests
```

4.9. Install

- In the install phase, the Install plugin installs a project's main artifact to the local repository
  - Has the `install:install` goal
  - Default behavior
    - Copy the jar file (or other) from `target/project-1.0.2.jar` to `.m2/repository/com/rd/project/project/1.0.2/project-1.0.2.jar`
    - If the project has POM packaging, the goal will also copy the POM to the local repository

---

4.10. Deploy

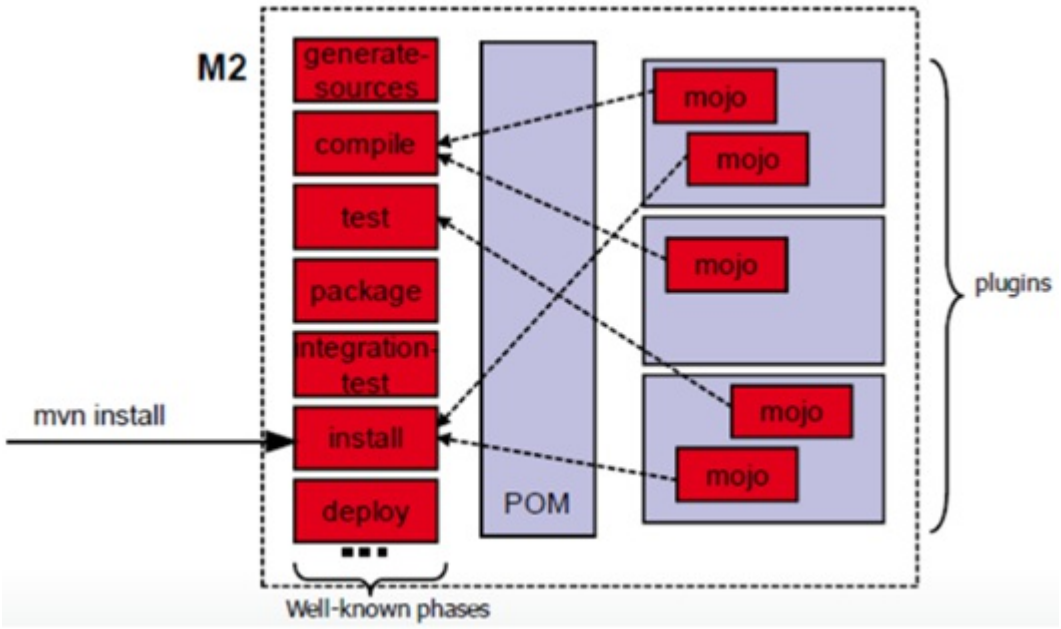
- In the deploy phase, the Deploy plugin will deploy an artifact to the remote Maven repository
  - Usually when you are performing a release
  - Has the `deploy:deploy` goal
  - Default behavior
    - Transporting an artifact to a published repository and updating any repository information which might be affected by such a deployment
  - The deployment procedure
    - Copying a file to another directory
    - Transferring a file over SCP using a public key
  - Deployment settings usually involve credentials to a remote repository
    - Found in the `~/.m2/settings.xml` (individual for each user)

4.11. Other lifecycles

- Maven has other lifecycles next to the build lifecycle
  - The clean lifecycle
    - Use the command `mvn clean`
    - Is used to remove Maven's work directory target
    - Some plugins will bind to the clean lifecycle to clean up after themselves as well
    - Phases: pre-clean, clean, and post-clean
  - The site lifecycle
    - Use the command `mvn site`
    - Much of the site generation is coordinated by Maven's reporting mechanism
    - It is also possible to bind plugins to be executed as part of the site generation lifecycle
    - Phases: `pre-site`, `site`, `post-site`, and `site-deploy`

4.12. Goals vs. plugins

- Plugins consist of one or more mojo's
  - A mojo performs a single task or goal
- Plugins group goals together, and attach those to the lifecycle phases for Maven to execute
  - Are configured in the `pom.xml` or implied through default behavior



4.13. Core plugins

- Archetype
  - To kick-start an application `mvn archetype:generate`
- Clean
  - Deletes files generated at build-time in a project's target directory
  - To delete additional resources

```
<plugin>
  <artifactId>maven-clean-plugin</artifactId>
  <version>2.4.1</version>
  <configuration>
    <filesets>
      <fileset>
        <directory>${basedir}</directory>
        <includes>
          <include>derby.log</include>
        </includes>
      </fileset>
    </filesets>
  </configuration>
</plugin>
```

- Compiler
  - Compiles your project with default javac
  - Other compilers: aspectj, csharp, eclipse, jikes, ...
  - You can also configure the default source and target versions for the compiler (e.g. from 1.5 to 1.8)

```
<plugin>
```

```
  <artifactId>maven-compiler-plugin</artifactId>
  <version>2.3.2</version>
  <configuration>
    <compilerId>csharp</compilerId>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.codehaus.plexus</groupId>
      <artifactId>plexus-compiler-csharp</artifactId>
      <version>1.5.2</version>
    </dependency>
  </dependencies>
</plugin>
```

- Surefire
  - Executes tests using JUnit, TestNG (dependency) or POJO's
    - JDK 1.4 `assert` or `org.apache.maven.surefire.assertion.Assert`
    - public test methods
  - Generates reports in XML and txt format
    - HTML reports by `maven-surefire-report-plugin`
  - Looks for `.java` files containing ending in `-Test` or `-TestCase`
    - `**/Test*.java`, `**/*Test.java`, `**/*TestCase.java`
  - Skip tests with `-Dmaven.test.skip=true` or `-DskipTests`
- Jar
  - Compiles code, tests code, and builds and/or signs jars
  - It can also generate manifests
- Install
  - Copies your project artifact to your local repository
  - You can install third party jar files using the `install-file` goal

```
mvn install:install-file
-Dfile=path-to-your-artifact-jar
-DgroupId=your.groupId
-DartifactId=your-artifactId
-Dversion=version
-Dpackaging=jar
-DgeneratePom=true
```

- Deploy
  - Publishes your artifact to a remote repository
  - Protocols: File, FTP, SCP, DAV, ...
  - You will need to configure the distribution repository
  - You will also need to add username / password to your `.m2/settings.xml`
- Site

    - Generate project website with:
        - Project info
        - Test results
        - Reports
        - Custom content & documentation
    - Preview the site on jetty using `mvn site:run`
    - Can use Velocity based skins
    - You will need to configure the site repository
    - You can also configure the plugin to add quality checks
        - Test coverage
        - Code style
        - JavaDoc
        - Unit test results
        - Dependencies
        - UML
- Release
    - Tags project in Software Configuration Manager (SCM)
    - Increases the project's version
    - Deploys the project artifact to remote repository
    - Requirements
        - SCM executables (e.g. `svn`) need to be on the `PATH`
        - The project needs to be in version control
        - No local changes are allowed (everything will be committed)
        - No SNAPSHOT dependencies allowed (this is a release)
        - Make sure you have a tags folder in Subversion
        - You will have to configure your SCM coordinates
- More plugins
    - http://maven.apache.org/plugins

4.14. Exercise: Maven build lifecycle

- Run the different lifecycle phases on your projects
    - `validate`, `compile`, `test`, `package`, `install`
    - Notice the different output and intermediate phases and goals
- Add a `.properties` file to your project and compile
    - Notice how the lifecycle includes the copy of the resource
- Run the site lifecycle on your projects
    - Notice the generated artifacts

- Add the Jetty plugin to the list of plugins in the web application project (inside the `<build>` element)
  - This makes the Jetty goals available
  - Use the following GAV: `org.mortbay.jetty:maven-jetty-plugin:6.1.26`
  - Run the project on Jetty

```
mvn jetty:run
```

5. Dependency management

5.1. Dependency

- A dependency
  - Is a way of expressing that the current project requires another project in order to build or run in some way
  - Uses Maven coordinates to locate that other project
  - May be another project you are building at the same time
  - May be another project you have already built on the current machine
  - May be a project built by another team member
  - May be a third party project (from a remote repository)
- Declaring a dependency will
  - Download jar files
  - Add it to the classpath
  - Bundle it into the resulting distribution if appropriate

5.2. Adding a dependency

- Create a source file in `src/main/java/com/realdolmen/dependency/`
  - Note the import of the third party library slf4j (http://slf4j.org)

```
package com.realdolmen.dependency;
import org.slf4j.*;
public class Example {
  final Logger logger = LoggerFactory.getLogger(Example.class);
  public boolean execute() {
    logger.info("Example action executed");
    return true;
  }
}
```

- Trying to compile with `mvn compile` fails with an error
  - The slf4j library was not added to the build
  - We will need to add the dependency to the `pom.xml`
- Add the following to the pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.6.4</version>
  </dependency>
  <!-- ... -->
</dependencies>
```

- These Maven coordinates will identify the dependency

    - Group ID: org.slf4j
    - Artifact ID: slf4j-api
    - Version: 1.6.4

- This is enough to retrieve and use the slf4j library
    - When compiling, slf4j will be downloaded and the build will succeed
    - When performing `mvn package`, the library will also be added to `target/simple-webapp/WEB-INF/lib`

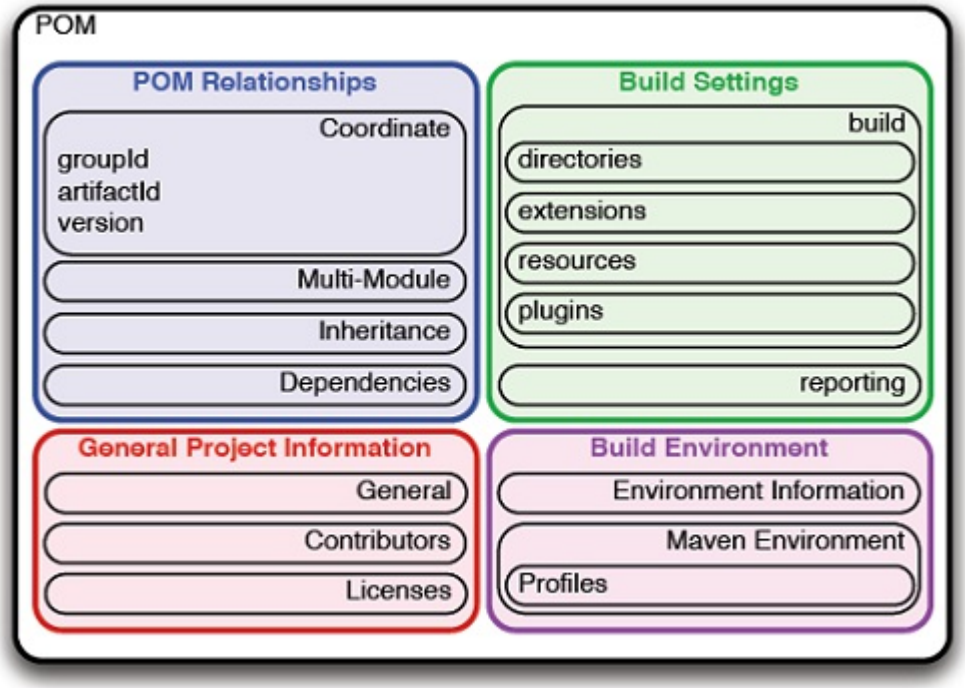
5.3. Finding dependencies

- What if you do not know the Maven coordinates?
    - Often you will need to find out what they are yourself
    - Some libraries will put their coordinates into their documentation
    - Repositories may have a search functionality, that enables you to find the coordinates you need

- Maven has a number of sophisticated ways to manage dependencies
    - Which ones are used
    - How they are used
    - What version should be used
    - Many dependencies will have dependencies of their own
        - Maven resolves these transitive dependencies as well

- Transitive dependencies
    - Are always enabled
    - But you can configure exclusions
        - This filters unnecessary transitive dependencies

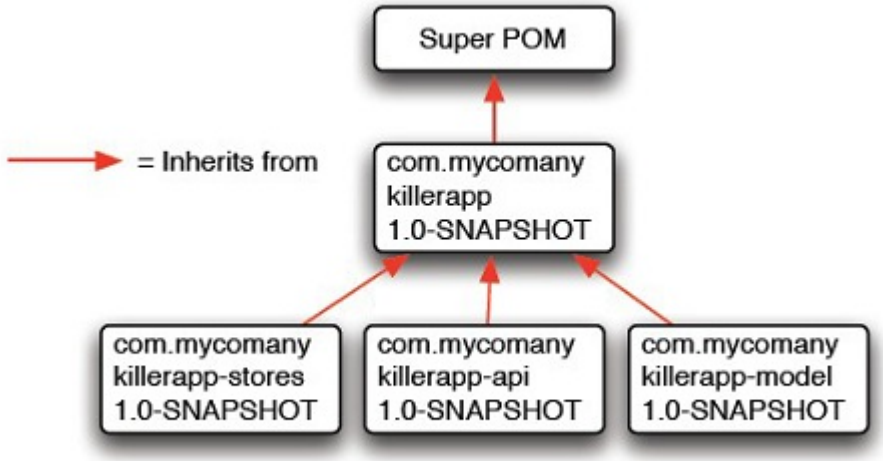
5.4. The POM file



- The `pom.xml` file is used to configure
  - The project
  - The build
  - The build environment
  - Dependencies

5.5. The Super POM

- The Super POM defines some standard configuration variables that are inherited by all projects



- The Super POM is always the base parent
  - Defines a single remote Maven repository (central)

- Contains the default Maven plugin repository (will not update them)
- Contains the default build directories
- Provides default versions of the core plugins

5.6. The Effective POM

- POMs inherit configuration from other POMs
  - A Maven POM will be a combination of the Super POM, plus any parent POMs, and finally the current project's POM
  - Maven starts with the Super POM and then overrides default configuration with one or more parent POMs
  - Then it overrides the resulting configuration with the current project's POM
  - You end up with an effective POM that is a mixture of various POMs
- To see the effective POM, execute the following in a directory with a `pom.xml` file
  - This should print out an XML document capturing the merge between the `pom.xml` and its parents to the Super POM

```
mvn help:effective-pom
```

5.7. POM properties

- POMs can include references to properties
  - Properties are simply variables delimited by `${ }`
  - You can also get variables from the `settings.xml` by using the prefix `settings` followed by a `"."`
  - You also have access to the Java System properties
  - The POM allows you to add your own properties

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.realdolmen.dependency</groupId>
  <artifactId>project-a</artifactId>
  <version>1.0-SNAPSHOT</version>
  <build>
    <finalName>${project.groupId}-${project.artifactId}</finalName>
  </build>
</project>
```

```
<properties>
  <foo>bar</foo>
</properties>
```

5.8. Dependency scopes

- You can define scopes to control classpath and bundling
  - `compile`
    - Default scope, available in all classpaths, packaged
  - `provided`
    - When you expect the JDK or a container to provide them, available on the compilation classpath (not runtime), not transitive, nor packaged
  - `runtime`
    - To execute and test the system, but they are not required for compilation (e.g. JDBC API vs. JDBC Driver implementation)
  - `test`
    - Available only during test compilation and execution phases
  - `system`
    - Similar to provided except that you have to provide an explicit path to the JAR on the local file system, to allow compilation against native objects that may be part of the system libraries
    - Not recommended for use

```
<project>
  <!-- ... -->
  <dependencies>
    <dependency>
      <!-- no scope = compile, test, execution -->
      <groupId>org.codehaus.xfire</groupId>
      <artifactId>xfire-java5</artifactId>
      <version>1.2.5</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <!-- ... -->
</project>
```

5.9. Optional dependencies

- You can mark dependencies as optional
  - You need the dependencies to compile your project
  - You do not want those libraries to show up as transitive runtime dependencies for other projects that uses your project artifact
  - The other project will have to add these explicitly when needed
  - Note that when using sub-modules, the use of optional is not really necessary

```
<!-- ... -->
<dependency>
  <groupId>net.sf.ehcache</groupId>
  <artifactId>ehcache</artifactId>
  <version>1.4.1</version>
  <optional>true</optional>
</dependency>
<dependency>
  <groupId>swarmcache</groupId>
  <artifactId>swarmcache</artifactId>
  <version>1.0RC2</version>
  <optional>true</optional>
</dependency>
<!-- ... -->
```

5.10. Dependency version ranges

- To specify a range of versions that would satisfy a given dependency
  - `(, )` Exclusive quantifiers
  - `[, ]` Inclusive quantifiers
  - `[4.0,)` any version greater than or equal to 4.0
  - `(,2.0)` is any version less than 2.0
  - `[1.2]` only version 1.2, and nothing else

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[3.8,4.0)</version>
  <scope>test</scope>
</dependency>
```

- When declaring the version normally, this is represented as "allow anything, but prefer the chosen version”
  - This allows Maven to use conflict algorithms when a conflict is found

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

```
  <version>[,3.8.1]</version>
  <scope>test</scope>
</dependency>
```

5.11. Transitive dependencies

- Are dependencies of dependencies
  - Are always enabled
  - Part of Maven's appeal is that it can manage transitive dependencies automatically
    - E.g. when you need the Spring framework, you configure Spring, but you will not have to worry about all of Spring's dependencies
  - Maven accomplishes this by building a dependency graph and dealing with conflicts that might occur
    - Maven will always favor the most recent version of a dependency
- Unnecessary transitive dependencies can be filtered
  - In some edge cases, transitive dependencies can cause configuration issues
    - Use dependency exclusion in those cases

5.12. Conflict resolution

- Excluding a transitive dependency
  - You might need to exclude or replace a transitive dependency when
    - A group ID or artifact ID has changed, which may result in 2 copies of the library
    - An artifact is not used in your project and the transitive dependency is not marked as optional
    - An artifact is provided by your runtime container
    - To exclude a library which might be an API with multiple implementations (see next example)

```
<dependency>
  <groupId>com.realdolmen.dependency</groupId>
  <artifactId>project-a</artifactId>
  <version>1.0</version>
  <exclusions>
    <exclusion>
      <groupId>com.realdolmen.dependency</groupId>
      <artifactId>project-b</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

- Excluding and replacing a transitive dependency
  - Both Oracle's JTA and Geronimo's JTA implement the JTA specification, but Geronimo's is freely distributed and available in the Maven repositories...

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.2.5.ga</version>
    <exclusions>
      <exclusion>
        <groupId>javax.transaction</groupId>
        <artifactId>jta</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jta_1.1_spec</artifactId>
    <version>1.1</version>
  </dependency>
</dependencies>
```

5.13. Dependency management in parent POM

- Imagine your super-complex enterprise with two hundred and twenty inter-related Maven projects
  - Every project uses a dependency like the MySQL Java Connector
  - Every project lists the dependency's version number
  - You need to upgrade to a new version...
  - You are going to have to manually edit each of the `pom.xml` files...
- Maven provides a way to consolidate dependency version numbers in the `dependencyManagement` element
  - Usually in a top-level parent POM
  - Allows you to reference a dependency in a child project, without having to list the version
  - Maven will walk up the hierarchy until it finds a project with the `dependencyManagement` element, and will use its version
- The top-level POM
  - Note the `dependencyManagement` element and version
  - This does not necessarily add the dependency in the children
    - It will help to manage the dependency and version in a centralized place

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.realdolmen.dependency</groupId>
```

```
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
  <!-- ... -->
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.18</version>
      </dependency>
      <!-- ... -->
    </dependencies>
  </dependencyManagement>
```

- The child POM
  - The version of the parent POM will propagate
  - If the child would specify a version, it would override the value of the one in the parent's `dependencyManagement` element

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.realdolmen.dependency</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <artifactId>project-a</artifactId>
  <!-- ... -->
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <!-- no version specified -->
    </dependency>
  </dependencies>
</project>
```

5.14. Grouping dependencies

- When you have a set of dependencies that are logically grouped together
- Create a project with `pom` packaging that groups dependencies together
  - E.g. when using Hibernate, you might need a dependency to Spring and the MySQL Connector
  - Create a project `persistence-deps` and have other projects depend on it

```
<project>
  <groupId>com.realdolmen.dependency</groupId>
  <artifactId>persistence-deps</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <properties>
```

```
    <mysqlVersion>(5.1,)</mysqlVersion>
    <springVersion>(2.0.6,)</springVersion>
    <hibernateVersion>3.2.5.ga</hibernateVersion>
    <hibAnnVersion>3.3.0.ga</hibAnnVersion>
  </properties>
  <!-- ... -->
```

```
<!-- ... -->
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>${hibernateVersion}</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-annotations</artifactId>
    <version>${hibAnnVersion}</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>
```

- Run `mvn install` to install the POM in your repository
- Add the dependency to projects
  - If you later decide to switch dependencies, you just need to adapt the persistence-deps
  - All dependent projects will be able to use the new dependency
  - This might be better than parent-child POMs
    - You can only have one parent
    - Your project can reference as many consolidated dependency POMs as it needs

```
<dependency>
  <groupId>com.realdolmen.dependency</groupId>
  <artifactId>persistence-deps</artifactId>
  <version>1.0</version>
  <type>pom</type>
</dependency>
```

5.15. Exercise: Dependency management

- Add an import to log4j in your project

```
import org.apache.log4j.Logger;
```

- Compile your project, and notice the failed build
- Add a dependency to log4j 1.2.4 in your POM and compile again (log4j:log4j:1.2.4)
  - Notice the dependency download
- Change the dependency by range for version 1.2.4 or higher and compile again
  - Notice a new dependency will be downloaded
- Package your project and look at the generated artifacts
- Remove your dependency, and replace the import for slf4j

```
import org.slf4j.*;
```

- Add the missing dependency and notice the download
- Add the following code and compile
  - Notice how everything compiles correctly
  - The code will not run, however...

```
import org.slf4j.*;
public class Example {
  final Logger logger = LoggerFactory.getLogger(Example.class);
  public boolean execute() {
    logger.info("logging execution");
    return true;
  }
}
```

- Take a look at the effective POM
- Add a dependency to testng (org.testng:testng:6.3.1)
  - Tweak the testng dependency scope and put it on test
  - Create a small test, then compile and run the test
  - Add source and target to the `maven-compiler-plugin` to accept annotations (at least Java 1.5!)
  - Add a dependency to org.slf4j:slf4j-simple:1.6.4, with a scope of runtime
  - Compile and run the test again
  - The test will fail, so adapt the test to succeed and try again

```
import org.testng.annotations.*;
public class ExampleTest {
  private Example ex = new Example();
  @Test
  public void valueIsTrue() {
    assert !ex.execute();
  }
```

```
}
```

- Organize your dependency versions using properties

6. Project testing

6.1. The rewards of testing

- Testing can be the most rewarding part of development
  - You can see for certain that the application works the way you intended
  - You can make sure it stays that way
  - You do not need to fire up the application to view your changes by hand and to find issues
- Testing can become tedious
  - There is always the temptation to “just skip them this once”
  - Remember: tests will bring repeated value as they run later
- Tests should be automated
- The concept of testing is built into the Maven lifecycle from the start

6.2. Types of testing in Maven

- Different types of tests can be automated in Maven
  - Unit testing
  - Integration testing
  - Functional testing (by building a separate test project)
  - These types of testing can be categorized differently and often overlap
- There are various other types of tests
  - System tests
  - Acceptance tests
  - Those can fit into other categories
- The names are not important to Maven
  - It does not constrain you in the types of tests that can be run

6.3. Unit testing

- Also known as code testing
  - Run after compilation but before packaging
  - Run on nearly every build
  - All tests should pass before the build can complete
- Phase is `test`
  - Make sure your tests run quickly, or else there is a greater inclination to just skip them

- Test should not break and be resilient to failure
- Do not rely on external resources
  - E.g. web server, database, ...
  - Such resources might not be present
  - Are best tested in a different stage of testing

6.4. Integration testing

- Also known as module testing
  - Run after packaging, but before installation in a repository
  - Test the code is working when put together with other modules that are already built
  - If the tests fail, the build should fail to prevent other projects to pick up the module
- Phase is `integration-test`
  - Should run reasonably fast as most developers run them on each build
  - Are expected to pass every time

6.5. Functional testing

- This includes broader types of testing
- Are usually run in one of two ways:
  - As a separate project:
    - Containing only test code and relying on pre-built application artifacts to test
  - As an optional part of an existing project:
    - Through combining profiles and the existing test phases
- In general, these tests may be run multiple times to test various different environmental combinations
- Are not enabled by default

6.6. Running tests

- Two steps are needed to add unit tests to projects
  - Add a dependency for the test library you have chosen to use
  - Add test code to your project in the `src/test/java` directory
- Many projects use JUnit
  - Is the most widely used unit testing framework

- Example for TestNG

```
<dependency>
  <groupId>org.testng</groupId>
  <artifactId>testng</artifactId>
  <version>6.3.1</version>
  <scope>test</scope>
</dependency>
```

- Notice the `test` scope
  - Makes sure the dependency is only used for the test phase
  - Tests will not work if you put them in the main application code

---

6.7. Surefire plugin configuration

- Surefire is a test runner
  - Designed to run tests from any supplied framework in a unified way
  - Feeds the results back to Maven consistently
  - Handles the entire infrastructure
    - Sets up class loaders
    - Forks the JVM
    - Tracks the total number of tests, passes, and failures
  - Compatible with JUnit 3, JUnit 4, TestNG, or tests written as plain Java classes
- To alter the way tests are run, we need to provide configuration information
  - In the `pom.xml`, to apply to the tests every time they are run
  - To the command line for once during the development cycle
- Controlling the execution of tests
  - Sets up a clean Java environment to run the tests in
  - By default starts a new JVM at the start
  - Does not impact the speed of the tests and is consistent with the settings used in most IDEs
- You can reconfigure the default behaviour with `forkMode`
  - `once`: The default, starts a new JVM to run all the tests
  - `never`: Runs all the tests within the current JVM
  - `always`: Starts a virtual machine for each test suite

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.19.1</version>
  <configuration>
    <forkMode>never</forkMode>
  </configuration>
```

```
</plugin>
```

- Remarks:
  - The `never` option can give a performance boost
    - It will also limit the amount of memory available for the tests and for the rest of the Maven build
  - The `always` option really isolates each test and removes any options, system properties, and memory for each test suite
    - The extra time required to run the tests is often too costly
- Including and excluding tests

```
<configuration>
  <includes>
    <include>**/*TestSuite.java</include>
  </includes>
</configuration>
```

```
<configuration>
  <excludes>
    <exclude>**/MyBrokenTest.java</exclude>
  </excludes>
</configuration>
```

6.8. Running specific tests from the command line

- Surefire runs through the whole suite of tests every time the build is run
- To be more specific you can use the `test` configuration option
  - Can be used to override the configuration from the command line
  - You can also use wildcards and commas to separate tests to select

```
mvn test -Dtest=ViewProjectPageTest
```

```
mvn test -Dtest=*PageTest
```

- Most developers run their tests from their IDEs which reduces the need to configure the test options
  - Do not use the test command line argument in a multi-module build
    - It is applied to all modules, so it will also skip tests for other modules
  - While you can skip tests, you should better not!

6.9. Producing reports

- The results of the tests can be found in `target/surefire-reports`
  - Both text and XML format
  - Useful for quickly viewing one test suite, particularly to see the reason for a test failure
- It is also possible to generate a HTML report
  - Start from the `webapp` directory

```
mvn surefire-report:report
```

- If you have already run the tests and would just like the report generated, you can use the following:

```
mvn surefire-report:report-only
```

- It is not common for this report to be included in the generated reporting site, as the site is usually generated after a successful build

6.10. Reviewing test coverage

- One of the most popular types of tools during testing are those that measure code coverage
  - While the tests are run the tool analyses which parts of the code are run
  - This gives an idea of how well tested the code is
- Code coverage tools, each with Maven plugins
  - Clover: commercial, from Atlassian
  - Cobertura: open source
  - EMMA: open source

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>emma-maven-plugin</artifactId>
      <version>1.0-alpha-3</version>
    </plugin>
  </plugins>
</build>
```

- Choose the module for which you want coverage

```
mvn emma:emma
```

- This will both run the tests and generate the coverage report
  - The compiled classes will be instrumented to include EMMA information to keep track of the coverage
  - When EMMA runs, it will collect the data
  - Do not use the never fork mode option!
- When analyzing the report, you can drill down to individual classes that have not been 100% tested
  - Color codes identify lines that have not or have partially been executed (e.g. in case of a branch)
- Configuring Cobertura instead of EMMA

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
      <version>2.7</version>
    </plugin>
  </plugins>
</build>
```

- Run with

```
mvn cobertura:cobertura
```

- You can automate test coverage with reporting
  - 100% coverage is not always feasible
    - Do you really need to cover trivial code like generated getters and setters?
  - 100% coverage does not mean 100% tested
    - You could have special situation that require special handling
  - A small percentage below 100% is usually acceptable
- You can configure Cobertura to ignore trivial code

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>cobertura-maven-plugin</artifactId>
  <version>2.7</version>
  <configuration>
    <instrumentation>
      <ignoreTrivial>true</ignoreTrivial>
    </instrumentation>
  </configuration>
</plugin>
```

6.11. Exercise: Project testing

- Add a few unit tests to your previous project
  - Make sure they test some different functionality
  - No complex code required `:-)`
- Reconfigure the fork mode to `never` and `always`
  - Run your tests again and notice the differences
  - Put the fork mode back to the defaults
- Generate a surefire report
- (Optional) Configure code coverage and generate a report

7. Multi-module projects

7.1. Advantages of multi-module builds

- It is rare for a project to contain just a single module
  - Enterprise Java applications quickly have multiple modules
    - WAR, JAR, EAR
- Advantages
  - Reusability: libraries can be reused in different sections of the application
  - Readability: applications will be easier to navigate and understand when broken up in self-contained modules
  - Development efficiency: development can be focused on a specific module, development time can be improved for compilation and testing on just one module
  - Separate release cycles: stable code can be released independently and be re-used by other projects
  - Enforced design constraints: you can separate an API from its implementation, to avoid consumers accidentally depending on implementation details

7.2. Modularization

- Start from a single POM
  - Adding more modules involve simple, incremental changes
  - Modules can be linked together using Maven's dependency mechanism
  - Principles of inheritance and composition used in Maven builds
- Warning!
  - Endlessly adding modules will increase complexity
  - You will need to monitor and refactor your build and plan for application design in advance!

7.3. Directory structure

- Your directory structure will probably be something like this
  - The application's parent sits at the top of the directory tree
  - The modules sit in a subdirectory under that parent project directory

```
blog
  |-- blog-ear
  |-- blog-jar
  |-- blog-web
  |-- pom.xml
```

7.4. Container projects

- Is the same as when creating a new project
  - You will need to set the `<packaging>` element to `pom`
  - Put common information in this parent POM, so that it does not have to be repeated
  - Maven will make sure the information is inherited

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.realdolmen.blog</groupId>
  <artifactId>blog</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Realdolmen Blog</name>
  <modules>
    <module>blog-jar</module>
    <module>blog-web</module>
    <module>blog-ear</module>
  </modules>
  <!-- ... -->
```

7.5. Creating the modules

- Each module can now be added using the `archetype:generate` command
  - This provides skeleton modules
  - The entire project can be build from the parent POM
  - The `archetype` plugin is able to add the modules to the parent POM, and update the child POMs to reflect the parent
- The child POMs: `blog-ear`

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.realdolmen.blog</groupId>
    <artifactId>blog</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <artifactId>blog-ear</artifactId>
  <packaging>ear</packaging>
  <!-- ... -->
```

- The child POMs: `blog-jar`

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
```

```
    <groupId>com.realdolmen.blog</groupId>
    <artifactId>blog</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <artifactId>blog-jar</artifactId>
  <packaging>ejb</packaging>
  <!-- ... -->
```

- The child POMs: `blog-web`

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.realdolmen.blog</groupId>
    <artifactId>blog</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <artifactId>blog-web</artifactId>
  <packaging>war</packaging>
  <!-- ... -->
```

7.6. POM inheritance

- The parent element in the child modules enables project inheritance
  - Makes development in the standard structure easier
  - The project will also be retrieved from the repository for building
- Notice that the group ID and version are no longer present in the child POMs
  - These values are the same as those from the parent POM, so they can be removed
  - You might want to add a name element to provide a descriptive name to the module
- Common dependencies only need to be updated once, in the parent POM

7.7. Module dependencies

- Put all common dependencies in the parent POM
  - Such as JUnit, as it will be used in all the submodules
- Modules will also depend on one another
  - `blog-web` will depend on `blog-jar`, `blog-ear` will depend on both `blog-web` and `blog-jar`
  - Add the dependencies in the child POMs
  - Note the use of expressions to avoid having to retype the same values
  - Modules will first be built following their interdependencies, then with the

ordering specified in the POM

```
<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>blog-jar</artifactId>
    <version>${project.version}</version>
  </dependency>
</dependencies>
```

7.8. POM aggregation

- Instead of declaring the version in each module, use the `dependencyManagement` element in the parent POM

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>blog-jar</artifactId>
      <version>${project.version}</version>
    </dependency>
    <!-- ... -->
```

- The child POM dependency elements can then be reduced to the following

```
<dependency>
  <groupId>${project.groupId}</groupId>
  <artifactId>blog-jar</artifactId>
</dependency>
```

7.9. Multi-module vs. inheritance

- There is a difference between inheritance and multi-module projects
  - A parent project is one that passes its values to its children
    - The parent-child relationship is defined from the child node upwards
    - The parent-child relationship deals more with the definition of a particular project
    - A child project's POM is derived from another (parent) POM
  - A multi-module project manages a group of other subprojects or modules
    - Is defined from the topmost level downwards
    - Groups modules together in a single build

7.10. Exercise: Multi-module projects

- Put your web application project in a multi-module layout
  - Add a small POJO as the domain model (`Book`)
  - Separate your domain model in a module (`core`)
  - Make sure your web application depends on it
  - Add an object to the request scope using a Servlet (`BooksServlet`)
    - You will need to add the Servlet dependency (javax.servlet:servlet-api:2.4) with scope `provided`
  - Show the data from your domain model in the web application using JSTL on a JSP page
    - You will need to add the JSTL dependencies (javax.servlet:jstl:1.1.2; taglibs:standard:1.1.2)
  - Update the `maven-compiler-plugin` to Java 1.7
  - Update your `web.xml` to version 2.4 and add the Servlet
  - You will need to install the dependencies in your local repository
  - Run on Tomcat
- Consider using `dependencyManagement` in the parent

8. Release management

8.1. What is release management?

- Most program Java applications use a multi-module Maven project
  - Each module has version numbers
  - During development you release several versions of your project
  - When you start a new version, you will have to change all versions in the different POM files of the multi-module project
    - This might be OK for just a few POM files, but in projects with more modules, this becomes unmaintainable
- You can configure Maven to adapt these versions automatically
  - So Maven will do release management for you!
- You will need to combine Maven with version control
  - Maven can be instructed to automatically create a new release tag, increase the version number in the trunk and commit the changes
  - Maven does this through the `maven-release-plugin`

8.2. Integration with source control

- To release with Maven, you will have to specify the location of your SCM
  - The example below uses Subversion
  - Maven needs access to the command-line Subversion client `svn`
- Add to your parent POM

```
<!-- ... -->
<modules>
  <module>blog-jar</module>
  <module>blog-web</module>
  <module>blog-ear</module>
</modules>
<scm>
  <developerConnection>
    scm:svn:file:///C:/repository/trunk
  </developerConnection>
</scm>
<!-- ... -->
```

8.3. Configuring project tags

- You can configure the Maven release plugin with the location of the tag base

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
```

```
  <artifactId>maven-release-plugin</artifactId>
  <version>2.2.2</version>
  <configuration>
    <tagBase>file:///C:/repository/tags</tagBase>
    <scmCommentPrefix>[sandbox] release:</scmCommentPrefix>
  </configuration>
</plugin>
```

- Before releasing, make sure to get a clean copy of your project
  - The `maven-release-plugin` will not run if you have locally modified files

8.4. Preparing a release

- First the release is prepared and if all goes well we can perform the actual release
  - You will be asked to
    - Enter a release version
    - Enter an SCM tag
    - Enter the next development version
  - Maven will present you with sensible default values, so just press `Enter`

```
mvn release:prepare
```

- Maven will
  - Update the version number in the pom.xml
  - Verify that the project builds
  - Tag the release in Subversion
  - Commit a `pom.xml` for the next development cycle to Subversion
  - Deploy the current release to your distribution repository

8.5. Preparing a release twice

- If you want to do `release:prepare` again, you need to ignore the created files

```
mvn release:prepare -Dresume=false
```

- You can also clean up the files from a previous `release:prepare`

```
mvn release:clean
del pom.xml
svn update
mvn release:prepare
```

8.6. Performing the release

- Now that the release has been prepared, it can be performed
  - This will build the project, remove the extra files that have been created and put release files in the `target` directory

```
mvn release:perform
```

- Maven will create the new release files and tag the release properly
- The build will fail if you do not have a distribution repository

```
<distributionManagement>
  <repository>
    <id>local</id>
    <name>localRepository</name>
    <url>file:///C:/target/dist</url>
  </repository>
</distributionManagement>
```

8.7. Exercise: Release management

- Configure a local Subversion
  - Use TortoiseSVN
  - After installing, make sure Subversion is available from the command line
- Configure your multi-module project for release
- Prepare and release the project
- Make some changes, test them, then prepare and release again

9. Profiles

9.1. What are profiles?

- Profiles allow to change the build lifecycle and/or settings
  - Useful for supporting different environments such as development, testing, staging, production, ...
- The goal is to make your project as portable as possible
  - Portability is the measure of how easy it is to take a project and build it in different environments

9.2. Types of portability

- Non-Portability
  - The portability you need to avoid
- Environmental Portability
  - Portable between specific environments (e.g. test database in a test environment, production database in production environment)
- In-House Portability
  - Portable only between environments in a single organization
- Wide (Universal) Portability
  - Important for open source, does not require any customization to build the project

9.3. Configuring projects with profiles

- Configured in the `pom.xml` and given an identifier
  - You can the run Maven with a command-line flag
- Example: override the default settings of the `compiler` in production

```
<profiles>
  <profile>
    <id>production</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.5.1</version>
          <configuration>
            <debug>false</debug>
            <optimize>true</optimize>
          </configuration>
```

```
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

- Notes on the example:
  - The `profiles` element is usually listed as the last element of the `pom.xml`
  - The chosen id can be run by passing `-P<profile_id>` to Maven
  - The profile element can override almost everything you would have in a `pom.xml`
  - We do not want the compiled code to contain debug information and make sure the bytecode has gone through the optimizations
- Running the profile
  - `-X` enables Maven debug output

```
mvn clean install -Pproduction -X
```

9.4. Profile activation

- Maven can activate a profile for different environmental parameters
  - To provide customizations for variables like OS or JDK version
  - No need to add the `-P`
- Example
  - When running JDK 1.6, also build the simple-script module

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

- Different selectors exist for different variables

9.5. Activation configuration

- The following example shows a very specific profile

```
<profile>
```

```
  <id>dev</id>
  <activation>
    <activeByDefault>false</activeByDefault>
    <jdk>1.7</jdk>
    <os>
      <name>Windows XP</name>
      <family>Windows</family>
      <arch>x86</arch>
      <version>5.1.2600</version>
    </os>
    <property>
      <name>mavenVersion</name>
      <value>2.0.5</value>
    </property>
    <file>
      <exists>file2.properties</exists>
      <missing>file1.properties</missing>
    </file>
  </activation>
  <!-- ... -->
</profile>
```

9.6. Activation in absence of a property

- You can activate a profile based on the value of a property
  - E.g. `environment.type` set to `dev` for development, `environment.type` set to `prod` for production
- You can activate a profile if that property is missing

```
<profile>
  <id>development</id>
  <activation>
    <property>
      <name>!environment.type</name>
    </property>
  </activation>
</profile>
```

9.7. Listing active profiles

- Profiles can be defined in
  - `pom.xml`
  - `profiles.xml`
  - `~/.m2/settings.xml`
  - `${M2_HOME}/conf/settings.xml`
- To list all active profiles and to see where they are defined, use the `active-profiles` goal

from the `help` plugin

```
mvn help:active-profiles
```

9.8. Profile example for common environments

- A profile for development on local MySQL database

```
<profiles>
  <profile>
    <id>development</id>
    <activation>
      <activeByDefault>true</activeByDefault>
      <property>
        <name>environment.type</name>
        <value>dev</value>
      </property>
    </activation>
    <properties>
      <database.driverClassName>com.mysql.jdbc.Driver</database.driverClassName>
      <database.url>
        jdbc:mysql://localhost:3306/test
      </database.url>
      <database.user>root</database.user>
      <database.password>root</database.password>
    </properties>
  </profile>
  <!-- ... -->
```

- A profile for the production MySQL database

```
<!-- ... -->
<profile>
  <id>production</id>
  <activation>
    <property>
      <name>environment.type</name>
      <value>prod</value>
    </property>
  </activation>
  <properties>
    <database.driverClassName>com.mysql.jdbc.Driver</database.driverClassName>
    <database.url>
      jdbc:mysql://master01:3306,slave01:3306/prod
    </database.url>
    <database.user>prod_user</database.user>
    <!-- password hidden in settings.xml -->
  </properties>
</profile>
</profiles>
```

9.9. Build parameters

- To target the default development environment

```
mvn install
```

- To make sure it is development

```
mvn install -Denvironment.type=dev
```

- To activate the production profile

```
mvn install -Denvironment.type=prod
```

- To test which profiles are active

```
mvn help:active-profiles
```

- No need to rewrite the build logic to support a new environment
  - Override the configuration that needs to change
  - Share the configuration points which can be shared

9.10. Exercise: Profiles

- Create a development profile
  - Override the `pom.xml` configuration to run on Tomcat
  - Make this the default configuration

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>tomcat-maven-plugin</artifactId>
  <version>1.1</version>
</plugin>
```

- Create a production profile
  - Override the `pom.xml` configuration to run on Jetty
- Test your profiles by running them on your project

10. Archetypes

10.1. What are archetypes?

- Project start-up is a work-intensive part, usually reserved to one individual developer
  - Archetypes will help you make this step easier
  - Archetypes will help to create small projects quickly
  - Archetypes can be tailored to your specific environment
  - Archetypes simplify project and module creation, and increase consistency
- Archetypes are templates for Maven projects
  - Maven projects have specific directory structures
    - Each type of project has different source folders and/or required files
    - Different project types have different packaging and POMs
  - Archetypes let you create the directories and default files in one command

10.2. Benefits of archetypes

- Archetypes conform to Maven's standard directory layout
- They implement best practices
  - Example: Maven Quickstart Archetype creates both the source and test directories suggesting to users that they write unit tests
- Maven projects have deep directory structures, that archetypes create automatically
- Archetypes create an error-free and consistent structure

10.3. Using archetypes

- You can use an archetype by invoking the `generate` goal of the Archetype plugin via the command-line or with m2e
- The following command line can be used to generate a project from the Quickstart archetype:

```
mvn archetype:generate \
-DgroupId=com.realdolmen.archetypes \
-DartifactId=blog \
-Dversion=1.0-SNAPSHOT \
-DpackageName=com.realdolmen.archetypes \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-quickstart \
-DarchetypeVersion=1.0 \
-DinteractiveMode=false
```

- It is more interesting to use the interactive mode
  - `interactiveMode`

        - It will prompt the user for all the previously listed command line parameters
        - When `interactiveMode` is false, the `generate` goal will use the values passed in from the command line
    - The default value of `interactiveMode` is `true`
    - Just use

```
mvn archetype:generate
```

- The Maven Archetype plugin will generate the project in a directory named after the `artifactId` you supplied
- Using m2e
    - Makes creating a new Maven project from a Maven Archetype very easy by providing an intuitive wizard

---

10.4. Using archetypes within multi-module projects

- Construct a multi-module project
    - Always start with a parent POM
    - For each module you want to add, execute `archetype:generate` inside the directory where your parent POM resides
        - Choose the type of module you want to add
    - The generated module automatically references the existing POM
    - The generated module is automatically configured as a new `<module>` element in the existing parent POM
- There is no generic archetype for multi-module projects
    - It is simply easier and far more flexible to just execute the Archetype plugin each time you need to add a new module
    - You cannot always determine at the beginning of the project how many modules (and sub-modules) the project will eventually have

While there may not be a generic multi-module archetype, there are several multi-module archetypes available from third-party organizations, that may fit your needs. You can find a few examples later in this chapter.

10.5. Common archetypes

- Common Maven archetypes

    - Are very basic templates that include few options
    - Provide the most basic features that distinguish a Maven project from a non-Maven project
    - `maven-archetype-quickstart`
        - A simple project with JAR packaging and a single dependency on JUnit
    - `maven-archetype-webapp`
        - A simple project with WAR packaging and a single dependency on JUnit
    - `maven-archetype-mojo`
        - A simple project with `maven-plugin` packaging and a single mojo class named MyMojo
        - Dependency on `maven-plugin-api` and JUnit
- More Maven archetypes
    - `maven-archetype-site`
        - Contains examples of several documentation formats as well as documentation in different languages
    - `maven-archetype-j2ee-simple`
        - Used in developing enterprise Java applications in the Java EE platform
        - A multi-module archetype, strictly specific for J2EE applications
        - Creates modules for web layer, application/business layer, and the persistence layer
- Notable third-party archetypes
    - Archetypes available from third-parties not associated with the Apache Maven project
    - A more comprehensive list of available archetypes, can be found when using m2e
    - AppFuse
        - An application framework for very popular Java technologies like the Spring, Hibernate, and iBatis
        - Using AppFuse you can very quickly create an end-to-end multi-tiered application that can plugin into several front-end web frameworks like Java Server Faces, Struts, and Tapestry
        - 9 choices available including: `appfuse-basic-jsf`, `appfuse-modular-jsf`, `appfuse-basic-spring`, `appfuse-modular-spring`, `appfuse-core`, ...
    - Confluence and JIRA plugins
        - Archetypes for people interested in developing plugins for both Confluence and JIRA
        - Use `jira-plugin-archetype` and `confluence-maven-archetype`
- More third-party archetypes
    - Wicket
        - A component-oriented web framework which focused on managing the

server-side state of a number of components written in Java and simple HTML
        - Focused on capturing interactions and page structure in a series of POJO Java classes
        - Use `wicket-archetype-quickstart`
    - Other examples:
        - JEE7 archetypes
        - Spring-OSGi archetype
        - Groovy archetypes
        - MyFaces archetypes
        - GWT archetypes

10.6. Creating archetypes

- An archetype consists of the following files:
    - The archetype descriptor: `archetype.xml`
        - This is where the specific files for the Maven directories are configured
    - The archetype's `pom.xml`
        - Each archetype is also treated like a regular Maven project
        - They are just a JAR with resources in a special place
    - The template directory structure
        - This will be copied into the generated project
    - The default `pom.xml`
        - Is included with the template directory structure
        - Will be the `pom.xml` to be placed in the generated project
- The Archetype plugin offers an easy way to create the archetype
    - Use the `create-from-project` goal to create an archetype based on files from an existing project

```
mvn archetype:create-from-project
```

- The plugin first resolves the package by guessing the project directory
    - It then generates the directory tree of the archetype in `target/generated-sources/archetype`
    - We then move to that directory and call `mvn install` on the created archetype
    - Now we can use our new archetype, by moving to a fresh directory and calling the following command
    - These steps also work on multi-module projects

```
mvn archetype:generate -DarchetypeCatalog=local
```

10.7. Exercise: Archetypes

- Use your web project as the base for an archetype
  - Create the archetype and put it in your local repository
- Create a new web project based on your archetype
  - Notice the similar structure and default available files
- Time permitting:
  - Configure a multi-module project using Spring, JPA and MySQL or Java EE

11. Deployment

11.1. What is Deployment?

- Deployment is when you publish your artifacts to a central repository
  - Makes your binaries available to others
  - Repository can be `public` or `private`
- Specialised repository management software is available from different vendors
  - Sonatype Nexus
  - JFrog Artifactory
  - Apache Archiva
- Maven has integrated support for deployment
  - The deploy lifecycle phase
    - Comes after install phase
  - The Maven deploy plugin
    - org.apache.maven.plugins:maven-deploy-plugin:2.8.2

It is worth noting that the install phase builds and deploys artifacts to the users' local repository (usually `.m2/repository`), whereas the deploy phase publishes the artifacts to a remote repository. This is typically done using a transfer protocol such as FTP, WebDAV or SSH.

11.2. Overview

- Throughout the next slides we will explore the different aspects of deployment with Maven
  - Configuring distribution management
  - Using the Maven deploy plugin
  - Setting up Maven wagons
  - Working with Sonatype's Nexus Repository Manager OSS

Our choice for Sonatype Nexus is due to its popularity on the market. Other repository management solutions work in very similar ways.

As we will see later, a wagon is Maven's term for abstracting transfer protocols.

11.3. Distribution Management

- A maven project is deployed using the following command:

```
$ mvn deploy
```

- You will need to add distribution management to your `pom.xml` first
  - `id` and `url` are required
    - These two elements will likely be defined in your repository management software (Nexus, ...)
  - Optionally you can add `name` and `layout`

```
<distributionManagement>
    <repository>
        <id>realdolmen</id>
        <url>http://localhost:8081/nexus/content/repositories/realdolmen/</url>
    </repository>
</distributionManagement>
```

- For snapshots, you can use the similar `<snapshotRepository>`
- For site deployments you need to add a `<site>` section

The optional properties are not usually needed.

- The `name` element is used as a description of the repository and has no technical significance.
- The `layout` element is only used when interfacing with an old Maven 1 type of repository.

The URL can of course be any valid URL where the organization's repository is hosted. This is a task for the repository management software (like Nexus).

For snapshots it is sometimes a good idea to use a separate repository, so your production-quality releases are kept separately.

Often the documentation pages are uploaded to a different location. In this case you can specify this separately using a `<site>` element.

11.4. Server Configuration in settings.xml

- Deploying to a repository needs to be a privileged action
  - Reserved for maintainers
- Repository management tools have security features
  - Username / password
- We need to specify the username and password
  - Not in the `pom.xml` (everyone will be able to see it)
- Maven has a special file for this sort of stuff:
  - `~/.m2/settings.xml`

The ~ refers to the users home directory. On Linux this is `/home/your_username`, on modern Windows this is `c:\Users\your_username`. If you do not already have a `settings.xml` file in this location, you can simply copy it from `$MAVEN_HOME\conf\settings.xml`.

- Add the following to your `settings.xml`:
  - Change where appropriate

```
<servers>
    <server>
      <id>realdolmen</id>
      <username>deployment</username>
      <password>deployment123</password>
    </server>
</servers>
```

- The specified user needs to have privileges to write on the given repository

For this example, we use username `deployment` and password `deployment123` since this is one of the default user accounts for Nexus.

If you are concerned with exposing your password in a text file like this, it is possible to encrypt your password first. Check out this url for more details on how this works: http://maven.apache.org/guides/mini/guide-encryption.html

11.5. Maven Deployment Plugin

- When invoking `mvn deploy`, the `maven-deploy-plugin` is doing the work
  - By default it hooks its `deploy` goal to the `deploy` lifecycle phase
- If you want you can manually configure it as follows:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-deploy-plugin</artifactId>
    <version>2.8.2</version>
    <configuration>
        ...
    </configuration>
</plugin>
```

- Configuration properties include:
  - altDeploymentRepository, altReleaseDeploymentRepository, altSnapshotDeploymentRepository, deployAtEnd, retryFailedDeploymentCount, skip, updateReleaseInfo

The different properties have the following meaning:

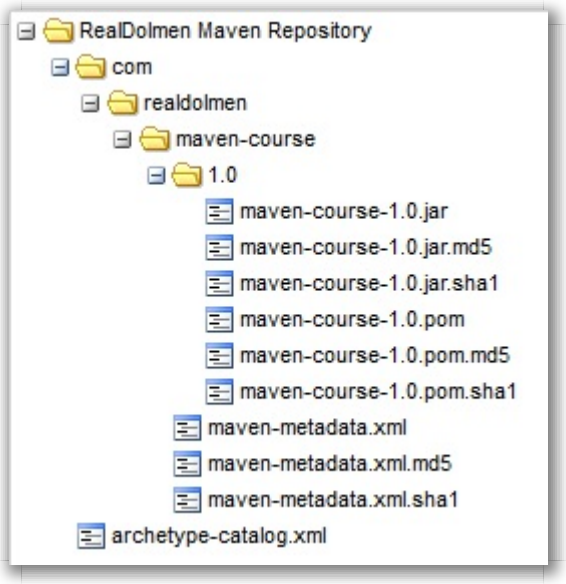
Property	Description

altDeploymentRepository	Specify a custom deployment location
altReleaseDeploymentRepository	Specify an alternative release deployment location (for final versions)
altSnapshotDeploymentRepository	Specify an alternative snapshot deployment location
deployAtEnd	In case of multi-module build: deploy all modules at the end
retryFailedDeploymentCount	How many times to retry (for network transients)
skip	Skips the deployment
updateReleaseInfo	Should artifact be marked as 'release' grade

Note that using the `maven-release-plugin` also allows a deployment to take place. In this case this is paired adding a new tag or branch in version control and upgrading the project version.

- When now executing the deploy command it will upload the artifact to the remote repository server

```
$ mvn deploy
```



The following are typical errors that you may encounter at this stage:

- Receiving HTTP 405: Your repository URL is bad (probably wrong path)
- Receiving HTTP 401: Your user credentials are bad (probably wrong password)

11.6. Maven Wagon

- We have previously entered a `http` protocol URL in distribution management
  - http://localhost:8081/nexus/content/repositories/realdolmen/

- When doing this, maven automatically selects the HTTP protocol to perform an upload using `PUT`
  - This is supported and enabled by default on a Nexus repository management server
- Maven supports many other protocols as well, though
  - Some must be specifically configured with a wagon
    - A type of plugin specifically for enabling protocols
- The following protocols are supported by maven using a wagon

Protocol	URL
File	file://
HTTP	http://
FTP	ftp://
SSH/SCP	ssh://
WebDAV	dav:https://

- Some wagons may need their own custom configuration
  - This is done in the `settings.xml` file in the corresponding `<server>` element
- Adding or creating custom wagons is also possible

For example using the URL `file:///var/www` under distribution management, the File protocol would be selected.

11.7. Example Using SCP / SSH Deployment

- Distribution management (pom.xml)

```
<distributionManagement>
    <repository>
        <id>realdolmen</id>
        <url>scp://realdolmen.com/repositories/public</url>
    </repository>
</distributionManagement>
```

- Plugin configuration (pom.xml)

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-deploy-plugin</artifactId>
    <version>2.8.2</version>
    <dependencies>
        <dependency>
            <groupId>org.apache.maven.wagon</groupId>
```

```
            <artifactId>wagon-ssh</artifactId>
            <version>2.7</version>
        </dependency>
    </dependencies>
</plugin>
```

When deploying using a secure copy, you need to enable the SSH wagon, which is specified as a dependency of the maven-deploy-plugin.

If you fail to enable the wagon you would get an error like this:

```
[ERROR] role: org.apache.maven.wagon.Wagon
[ERROR] roleHint: scp
```

11.8. Sonatype Nexus Repository Management OSS

- One of the most popular repository management solutions is Nexus
  - http://www.sonatype.org/nexus/

- It is available in two flavors
  - Commercial (paid) version
    - Requires payment
  - Open Source (gratis) version
    - This is the version we will focus on
    - Uses the Eclipse Public License (EPL)

The Open Source version of Nexus - the one we use in this course - is released under the EPL license. This means it is free and open source, but allows proprietary add-ons to be developed which are not. Some of these proprietary plugins are included with the release and may be used gratis.

11.9. Nexus Features

- Sonatype Nexus is a full featured solution for repository management
- It has the following features

- Centralizes software components
- Caches remote repositories
- Supports multiple package types
  - Maven artifacts, NPM packages, ...
- Security management
- Scalability

Centralization of software components allows businesses to group all project dependencies together, allowing easy access and control of the dependencies used by project teams.

Public repositories are transparently cached; every dependency that has been fetched once will remain cached on the server. This improves download speeds and offers more resilience to network transients.

Modern versions of Nexus do not only support Maven, but also NPM.

The security mechanism allows fine grained control over who can do what on the system, varying from read-only access to full repository control for deployment of artifacts.

For the most demanding enterprises, Nexus also features high-scalability. As an example, JBoss uses Nexus for its private and public artifact repositories.

11.10. Installation and Setup of Nexus

- Nexus can be obtained from the website of Sonatype
  - http://www.sonatype.com/download-oss-sonatype

- After downloading extract it to a suitable location

The distribution of Nexus comes in a zip and tar.gz package. When on Windows, use the zip version, otherwise you can use the tar.gz version which is better supported on Linux systems. Both really contain the same contents, so it does not actually matter much.

- Once extracted, you can proceed to install it
  - You will need administrator rights to do so
- Open up a terminal window, and navigate to the extracted location

```
$ cd bin
$ nexus install
```

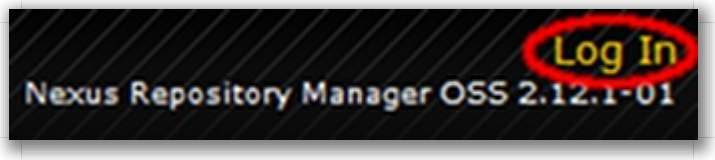
- After installing you can launch the service
  - This will start a daemon, and make sure it is launched automatically upon startup

```
$ nexus start
```

- Congratulations! Your Nexus instance is now ready for work
  - Navigate to its landing page: http://localhost:8081/nexus

11.11. Logging on

- Nexus comes with 3 default user accounts (can be changed later)
  - Administrator (admin/admin123)
    - Has full control
  - Deployment (deployment/deployment123)
    - Has control over repositories and has write access
  - Anonymous
    - Has read access only
- You can log on using the "Log In" link on the top right



11.12. Nexus Administration Panel

- The Nexus Administration section allows privileged users to configure server parameters
  - Logging configuration
  - Active plugins
  - Gathering of metrics
  - Configuring task schedules
  - Editing settings

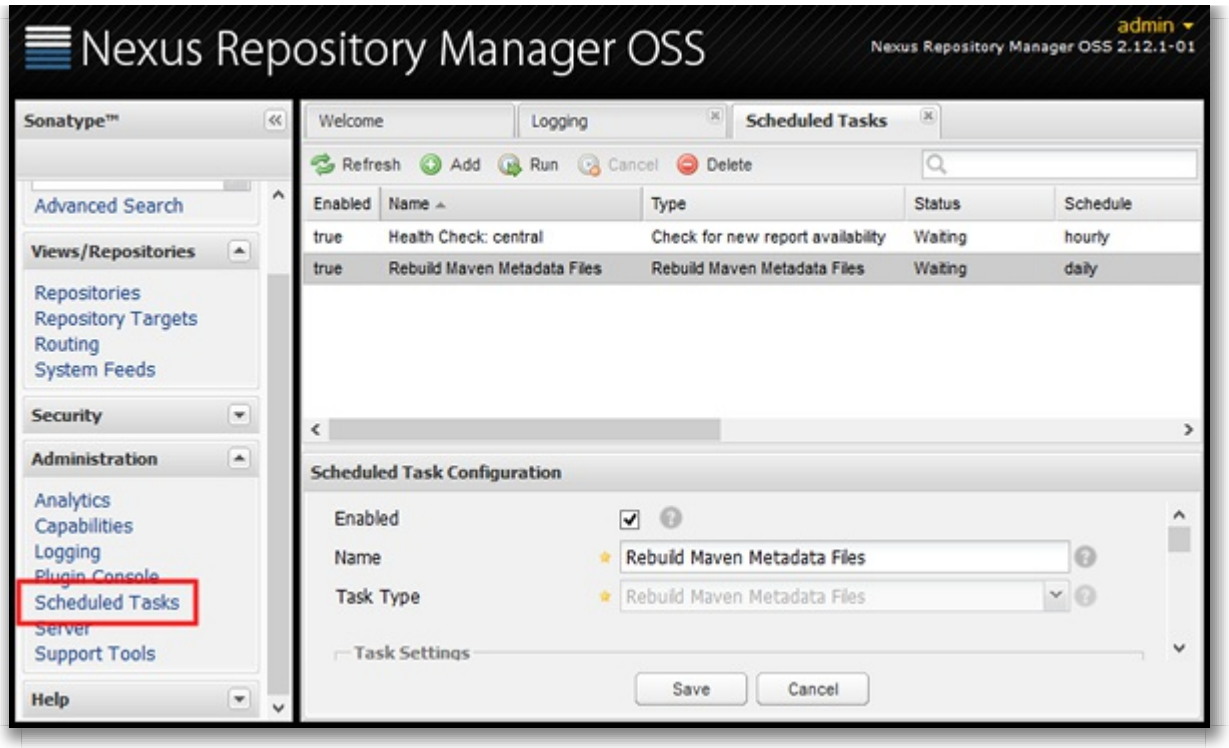
    - Mail server, HTTP server, Proxy settings, etc...
- This tends to be a task for the system administrators rather than developers
- Server administration allows configuration of
    - SMTP server, HTTP settings, anonymous user, application context root, HTTP proxy and email notifications

This panel allows configuration of usual server settings. The most useful of these are:

- Setting the base context-root of the application
- Configuring the SMTP server to use for sending emails
- Setting the HTTP proxy for accessing the internet

Other features are more specialized, but sometimes important for deploying in an enterprise environment.

- Scheduled tasks administration allows configuration of recurring tasks such as
    - Rebuilding the Maven indexes, garbage collection, ...



The configuration of scheduled tasks allows tasks be created with a name, a task to perform (which can be selected from a drop-down menu) and a recurring pattern in CRON-style (or Quartz for Java developers). Here are some examples:

Pattern	Meaning
0 0 12 ?	Fire at 12pm (noon) every day
0 15 10 ?	Fire at 10:15am every day
0 15 10 ?	Fire at 10:15am every day
0 15 10 ? *	Fire at 10:15am every day
0 15 10 ? 2016	Fire at 10:15am every day during the year 2016

- Logging administration allows configuration of logger categories

The logger configuration is very similar to what is typically done in a Java logging framework such as `log4j.xml` or `logback.xml`. This should be familiar to most Java developers.

11.13. Nexus Security Panel

- The Nexus Security section allows fine-grained control over user accounts
  - Connecting Nexus with an external LDAP system (for reusing the company usernames/passwords)
  - User accounts
  - Roles and Privileges
- It is strongly advised that you change the passwords for the default accounts here ASAP!

When using the LDAP feature, you can connect Nexus' authentication mechanism to your company's credential store. This allows the same user accounts to be reused, offering end-users a seamless experience.

As an alternative, you can also manually manage User accounts (or add additional ones that are not in the company database). An interesting thing to point out is that Nexus differentiates between Roles and Privileges.

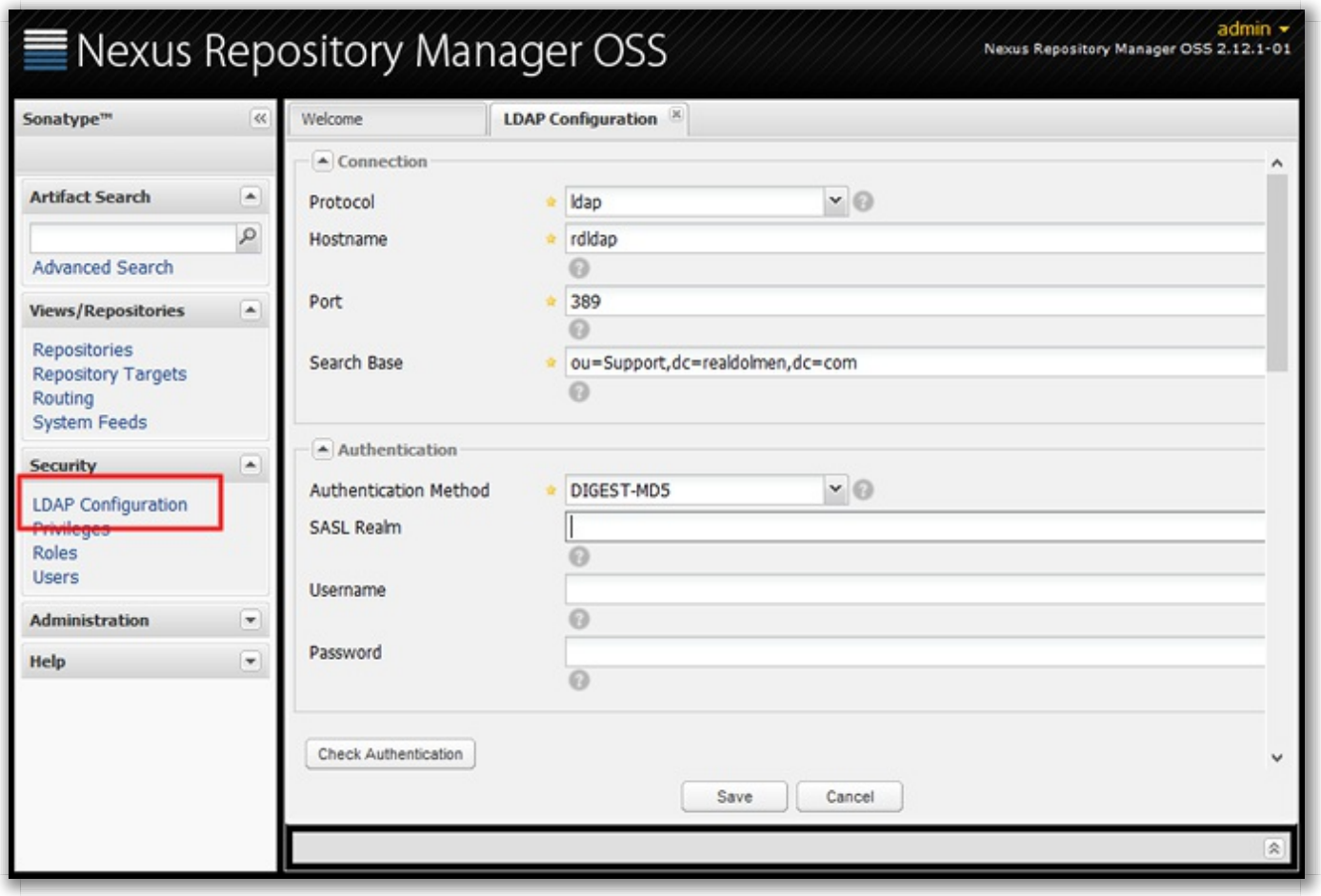
Privileges are the actions that are allowed to be performed by any user who holds that privilege. These are predefined in Nexus and are described on a per-repository level. Thus it is possible for a certain privilege to grant access to a specific repository, while not granting access to another one.

Roles then, are described as a combination of Privileges, which the company regards as an atomic set of grants that a user with said Role should be able to perform.

Finally, a User us defined as a combination of Roles. A User with a specific Role will thus receive grants for all the Privileges within that Role. A user obviously also has a username and password which allows authentication. This authentication can be performed both on the Nexus web-application as well as through Maven using a server configuration.

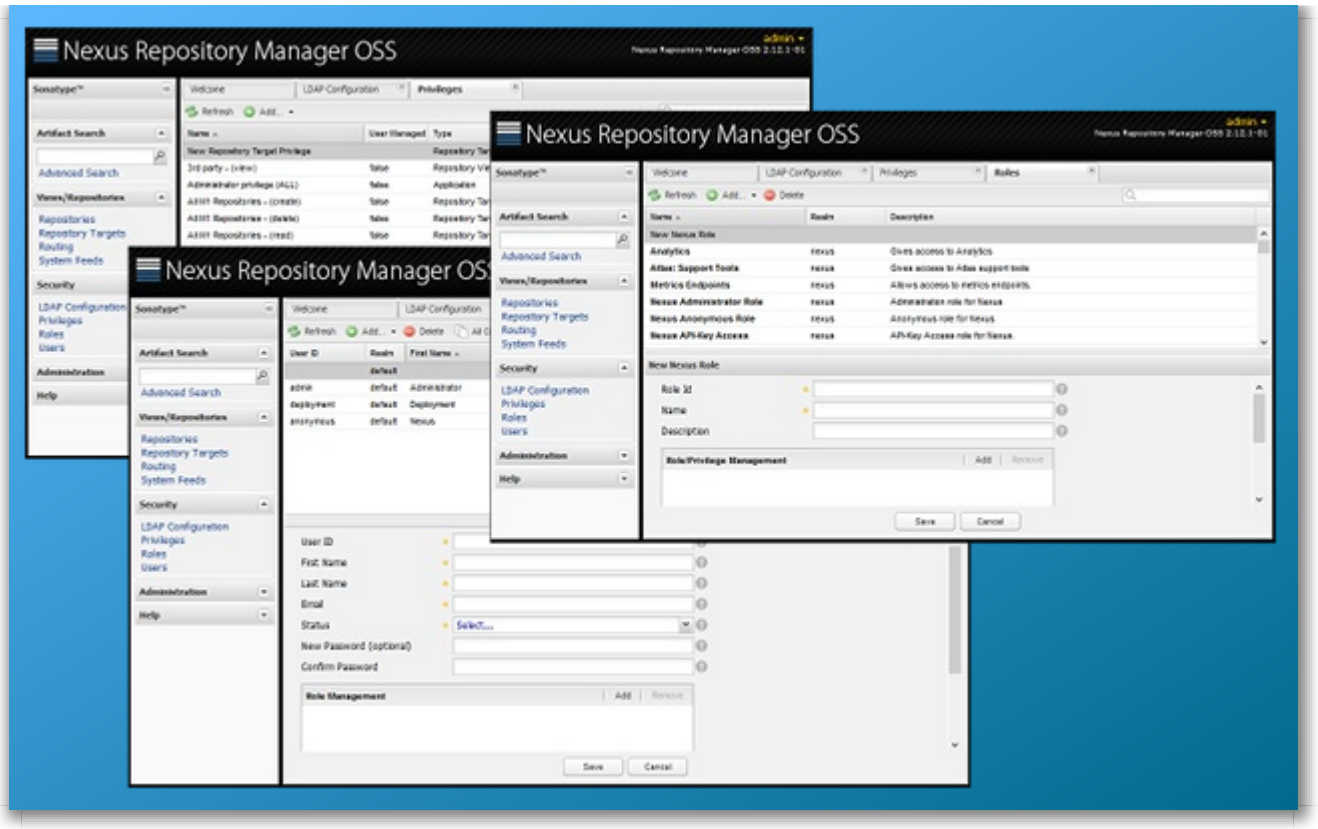
For each of these, the Security Panel offers a pretty self explanatory GUI to create, read, update and delete these entities.

- LDAP security configuration allows coupling Nexus to an external LDAP server



In order to do this, you will need a good understanding of LDAP principles, and the company LDAP layout. This is somewhat outside of the scope of this course, so we will not go into details about this. One example of an LDAP enabled service is Microsoft's Active Directory.

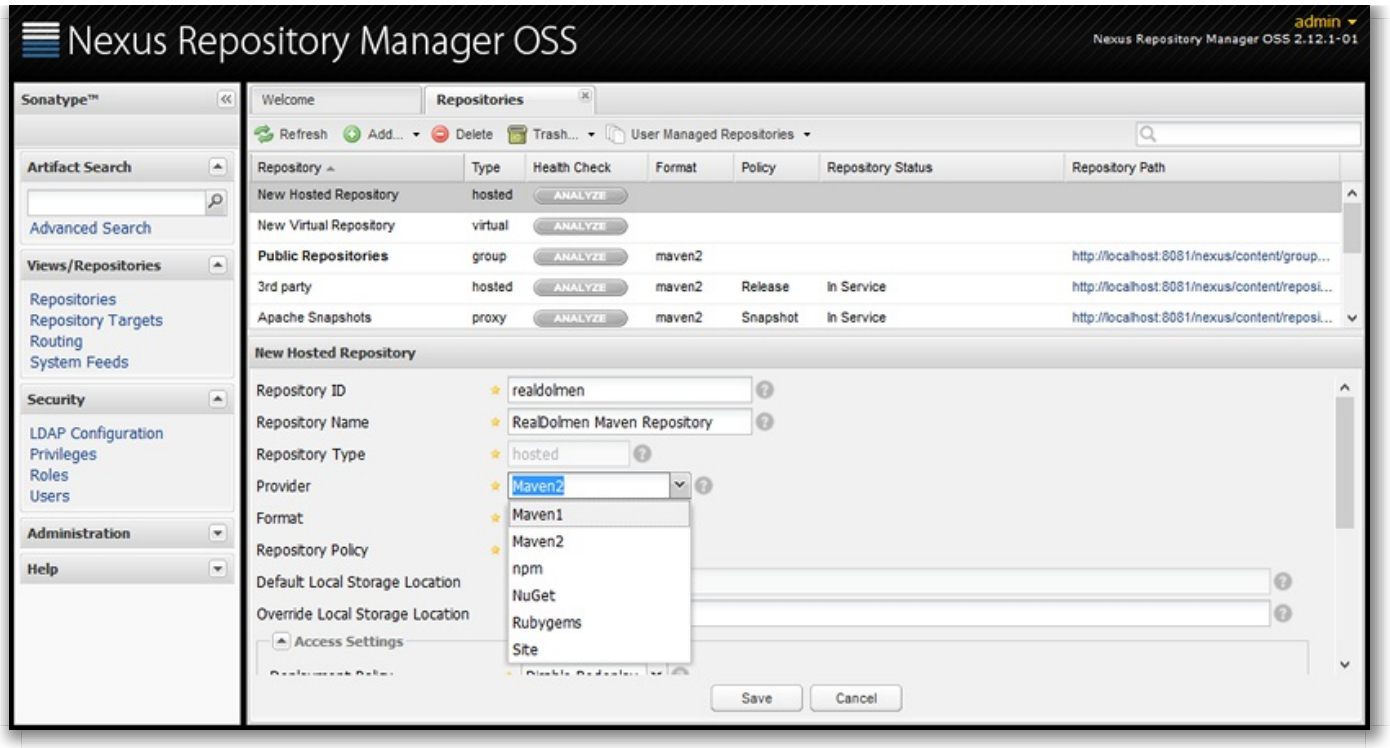
- Using the Users, Roles and Privileges sections you can add and remove user configurations



The user administration in these three panels are obvious, and require little explanation. The users has options to create, edit and remove Users, Roles and Privileges using the displayed GUI controls.

11.14. Nexus Repositories Panel

- The Nexus Repositories section allows administration of different repositories
- There are three types of Repositories
  - Hosted repositories
    - These exist only locally, and contain all (private) company artifacts
  - Proxy repositories
    - There are caches from the public online repositories (such as Maven central)
  - Virtual repositories
    - These are logical repositories intended for exposing different layouts of the same repository
- Using the Repositories section you can add various types of repositories
  - Supports other formats than Maven such as NPM, NuGet, Rubygems, ...



Repositories can be added with the GUI, where you must fill in all relevant features of the new repository such as:

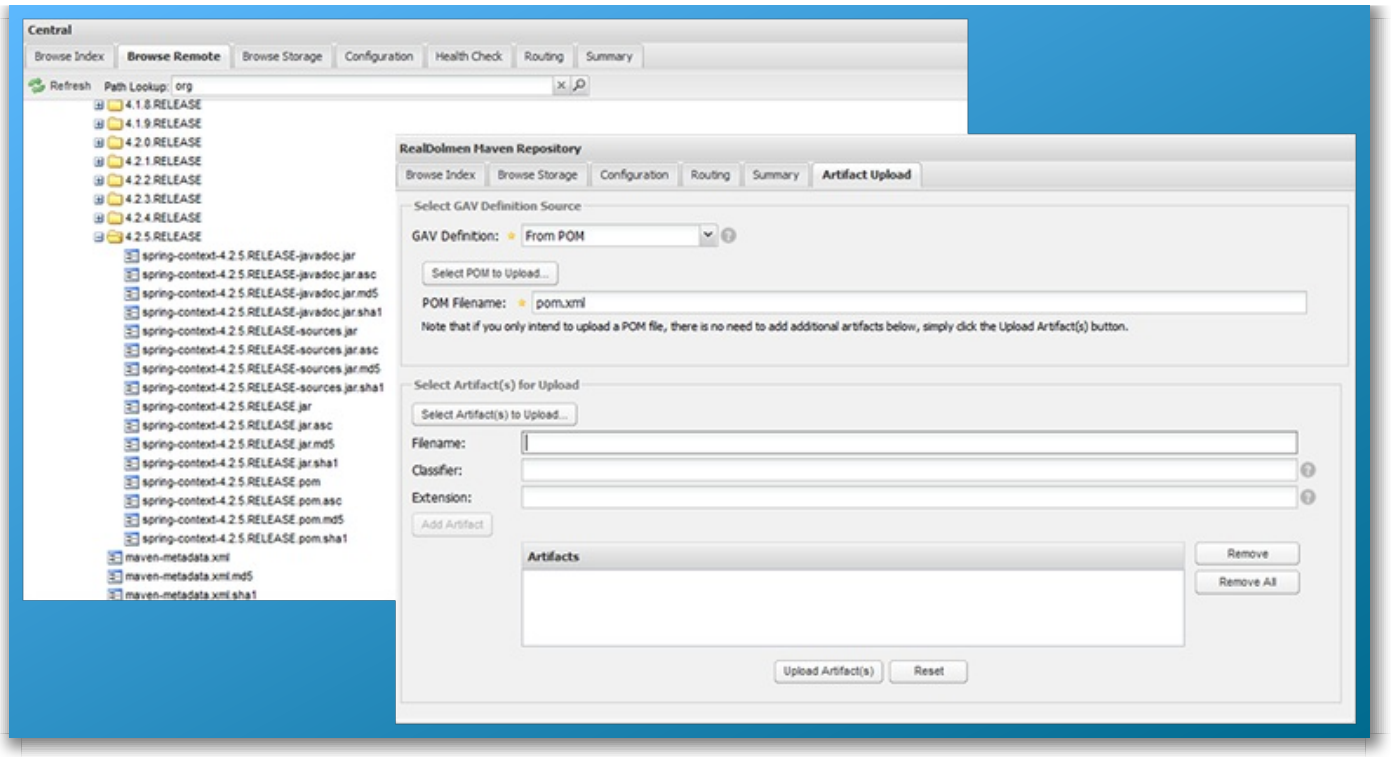
- Id
- Name
- Type (Maven1, Maven2, NuGet, NPM, ...)
- File browsing
- Policy (Snapshots or release artifacts)
- ...

In case the repository is Hosted, it can be referenced from a Maven `pom.xml` to target this repository for deployment. The user will require write privileges to do so.

In case the repository is Proxied, it will start serving as an intermediary and projects can use this repository to download dependencies from without requiring an open internet connection.

In either case, your newly created repository will be accessible on an URL relative to the base context-root of the nexus instance (`http://localhost:8081/nexus` by default). For example, when your repository id is `realdolmen`, using default settings the URL will be `http://localhost:8081/nexus/content/repositories/realdolmen/`.

- Some other Repository capabilities
  - Manually upload an artifact (from a `pom.xml` or binary)
  - Browse proxied and hosted repostories



Uploading an artifact manually is to be avoided. However, it can be very useful if the binary does not exist anywhere on any existing repository. This is most likely the case when you have a proprietary (non open-source) dependency. This way you can still offer the said dependency to project teams using the standard Maven mechanism.

The panels Browse Index, Browse Remote and Browse Storage allow repositories to be browsed and searched using a tree-structure.

11.15. Adding a Repository Mirror

- A Proxy repository can be configured as a mirror of public repositories
  - This has the advantage that development is decoupled from the public repository
- Configuring a mirror is done in the `settings.xml` file

```
<mirror>
  <id>your_mirror_id</id>
  <mirrorOf>*</mirrorOf>
  <name>Human Readable Name for this Mirror.</name>
  <url>http://localhost:8081/nexus/content/repositories/proxy</url>
</mirror>
```

Since a Proxy repository can be configured to automatically cache any downloaded dependency within the company network, it is possible to configure maven installations to use this Proxy as a mirror for all dependencies. This can be interesting in some cases, where public internet access is not always guaranteed, or when trust in the availability of the remote repository is low.

You must always be careful though, since configuring mirrors like this may sometimes reduce transparency and become confusing.

12. Final thoughts

12.1. Summary

- Maven is an indispensable build tool for enterprise projects
  - Projects are consistently managed throughout your company
  - No longer do you need to manage 3rd party dependency management and transitive dependencies
  - Building and maintaining software has never been easier
- The fixed Maven build lifecycle enables to customize builds and add plugins
- Maven manages direct and transitive project dependencies and allows to consolidate them
- Maven enables the use of multi-module projects
- Maven easily integrates with version control
- Archetypes in Maven let you configure and start up new projects in no time

Maven 3 Exercises

Text: EN

Software: EN

REALDOLMEN
to get there, together

Table of contents

1. Deployment

These exercises will teach you how to use a repository management system and how to perform a Maven deploy to it.

Note: for some of these exercises you will need an open internet connection.

1.1. Launching a Nexus instance

Start up Nexus

In the real world, installing Nexus (after downloading and extracting) requires executing the following simple steps:

```
cd path/to/nexus/bin
nexus install
nexus start
```

This installs Nexus as a daemon, and automatically starts at boot.

During the course however, we may not have administrator rights, which are required to install daemon processes. For this reason we have prepared a simple script that launches a Nexus instance in the console:

```
start-nexus.cmd
```

You can find it in the portable environment's home directory. Ask the trainer if you can't find it.

Go ahead and execute the file. When the server started successfully, you should something like this:

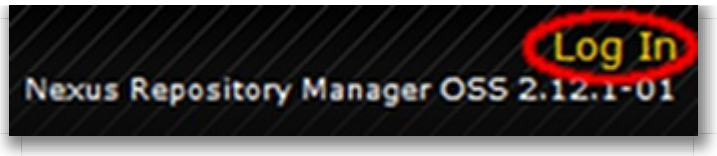
```
SYSTEM org.sonatype.nexus.bootstrap.jetty.JettyServer - Running
SYSTEM org.sonatype.nexus.bootstrap.jetty.JettyServer - Started
```

Configuring Security

Now that your Nexus is up and running, navigate to it's administration console using the following URL:

```
http://localhost:8081/nexus
```

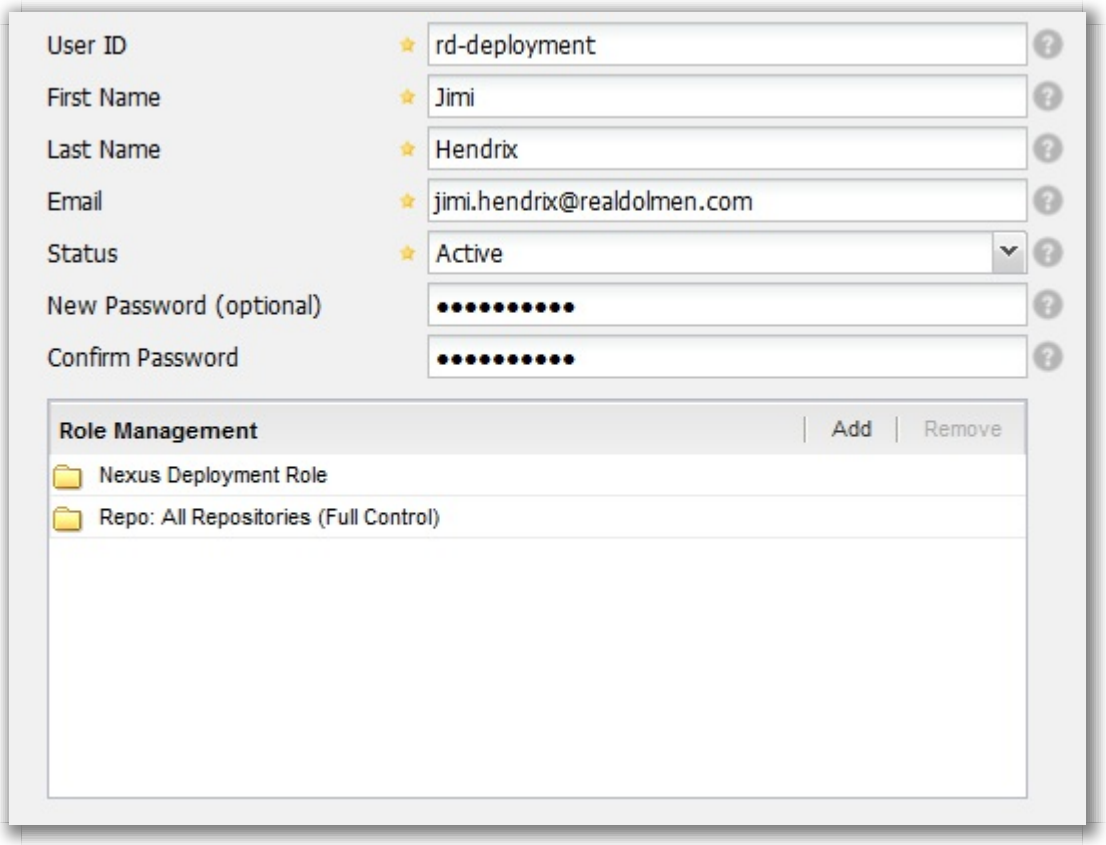
The default login generated by nexus for the administrator account is username: `admin`, password: `admin123`. Use this account to log on.



We're not happy with this default! It makes our Nexus instance vulnerable to attack! Let's fix that now!

- Delete (or disable) the default users `admin` and `deployment`.
- Add two new users:
  - User Id: `r-administrator`, password: `Education`, roles: `Nexus Administrator Role`
  - User Id: `r-deployment`, password: `Realdolmen`, roles: `Nexus Deployment Role`, `Repo: All Repositories (Full Control)`

As for the names and email addresses: you can make something up. What about this:



Note: In the real world, you would of course choose a strong password, and possibly use more fine grained role assignments.

1.2. Creating Repositories

Add a Proxy Repository to Nexus

Note: This exercises requires an open (unproxied) internet connection. If this is not the case, you can skip this exercise.

Before starting this exercise, (temporarily) rename your local maven cache.

We want to create a mirror for the maven central repository. This will improve reliability for our development teams.

- Log on to Nexus as user r-administrator
- Create a new proxy repository
  - The repository should proxy the maven central repository, which has URL `https://repo1.maven.org/maven2/`.
  - The name of your proxy should be `r-central-proxy`

This should be the URL of your new proxy repository:

```
http://localhost:8081/nexus/content/repositories/rd-central-proxy
```

Initially the mirror will be empty. This is due to the fact that the proxy will use a lazy caching strategy.

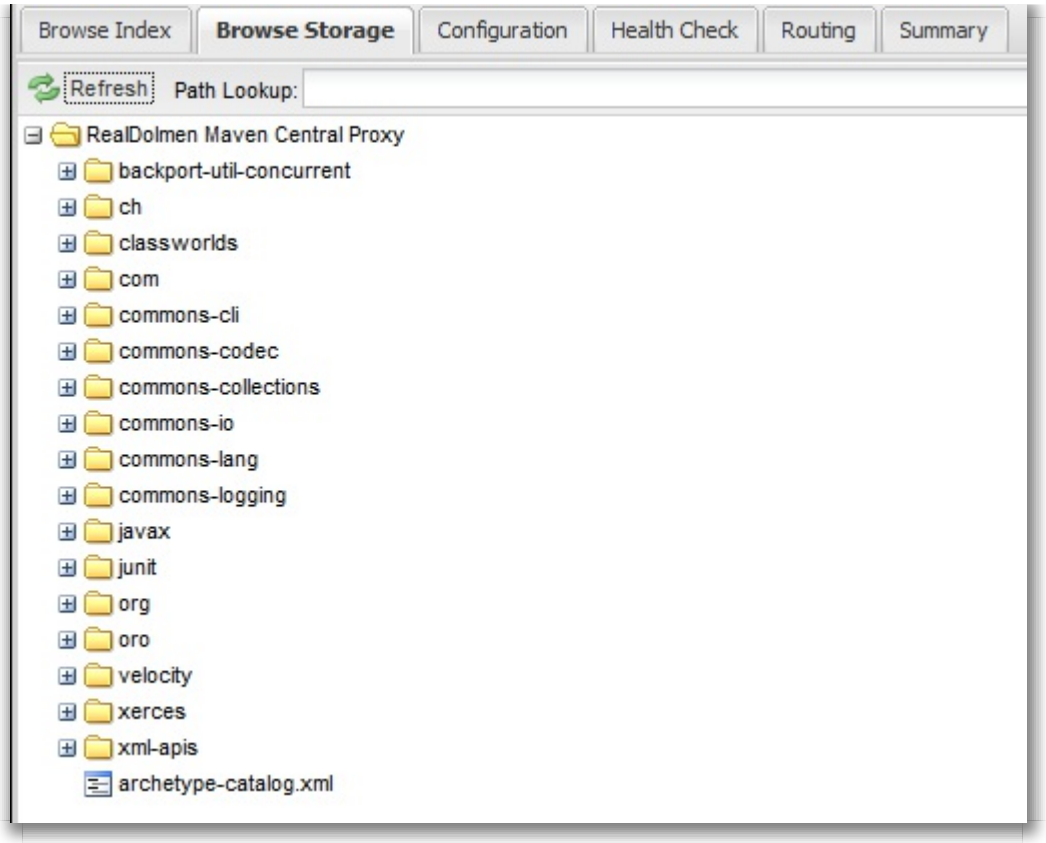
When the repository is up and running, you will have to configure it as the mirror for your Maven project.

- Open your `settings.xml` file and add a mirror for:
  - id: `r-central-proxy`
  - mirrorOf: `central`
  - url: `http://localhost:8081/nexus/content/repositories/r-central-proxy`
  - name: `Realdolmen Maven Central Proxy`
- Execute a maven build `mvn clean package`.

You should see that your build is starting to download dependencies.

Note: If this is not the case, and your build is still successful, you may have forgotten to rename the local Maven dependency cache.

You should also see that the Proxy repository on Nexus is starting to fill its cache.



Depending on the speed of the internet connection this process may take some time to complete. At the end you should have a successful build.

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Add a Hosted Repository to Nexus

In this step, we will add another repository to Nexus. This time it will be a Hosted repository.

- Log on to Nexus as user r-administrator
- Create a new proxy repository
  - The name of your repository will be 'rd-releases'
  - The Repository Policy should be Release
  - Make sure you allow Redeploy

When configured correctly, the new URL will be:

```
http://localhost:8081/nexus/content/repositories/r-releases/
```

We will use this repository for our `mvn deploy` commands!

1.3. Performing Maven Deployments

Adding Distribution Management

Now that we have configured our Nexus instance, we are ready to configure Maven deployment.

The first step to perform is to add distribution management to our `pom.xml`. You need to set at lease these properties:

- id: `r-releases`
- url: `http://localhost:8081/nexus/content/repositories/r-releases/`

Configuring Server Credentials

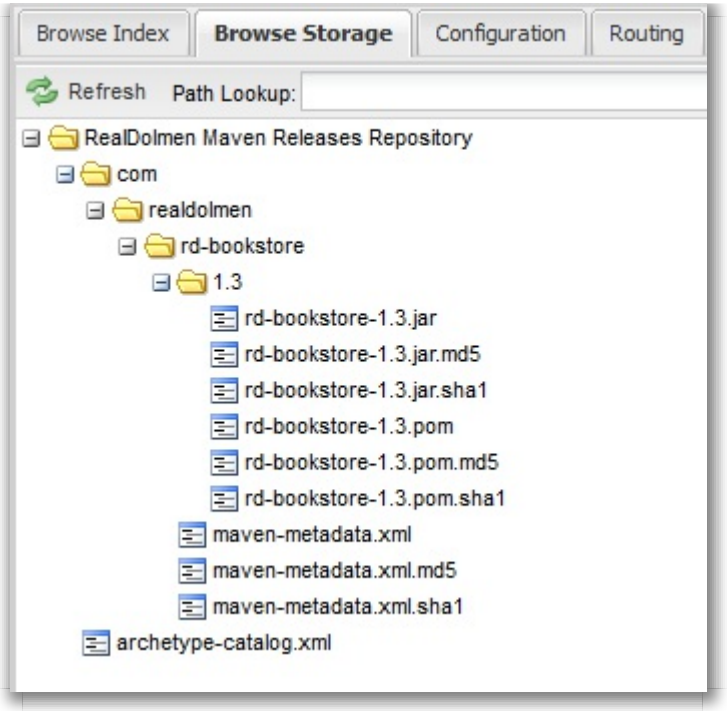
After adding distribution management, we're back to our `settings.xml`. Add a server configuration to add authentication credentials.

- id: `r-releases`
- username: `r-deployment`
- password: `Realdolmen`

When this is complete, you can perform a deployment using `mvn deploy`. You should see results similar to this:

```
Uploading: http://localhost:8081/nexus/content/repositories/r-
releases/com/realdolmen/r-bookstore/maven-metadata.xml
Uploaded: http://localhost:8081/nexus/content/repositories/r-
releases/com/realdolmen/r-bookstore/maven-metadata.xml (302 B at 6.3 KB/sec)
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Also verify your repository on Nexus. It should now show the dependency in Browse Storage.



Congratulations! You have succeeded in deploying your artifact to a custom Nexus repository!