

Enterprise Programming 1

Lesson 01: Introduction

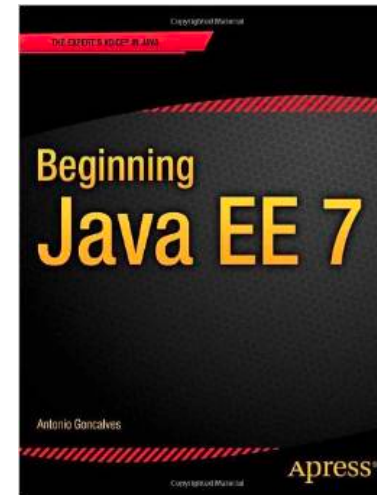
Dr. Andrea Arcuri
Westerdals Oslo ACT
University of Luxembourg

Course Info

- 12 lessons, once a week
- Check TimeEdit for possible changes of time and rooms
- 4-hour lectures
 - Between 2 and 4 hours of teaching
 - Remaining time is for exercises and questions
- Focus on coding and exercises

Course Material

- Git repository:
https://github.com/arcuri82/testing_security_development_enterprise_systems
- *Note: pull often, as material can get updated throughout the course*
- Book: *Beginning Java EE 7*
 - Chapters 1-11
 - Useful to have, but not essential



Necessary Tools

- JDK 8
- Git
- Maven
- An IDE (I strongly recommend IntelliJ IDEA)
- Docker
- A Bash command-line terminal
 - Mac/Linux: use the built-in one
 - Windows: I recommend GitBash

Goals

- Learn about developing enterprise applications, with a focus on how to **test** and **secure** them
- Access to databases
- Business logic with enhanced beans
- Server side rendering with HTML templates
- **Not only studying theory, but being able to build and deploy a full application**

Enterprise Applications



- **Large** business applications
 - Tens/hundreds of people (developers/managers/etc) involved
 - Many (different) processes running on many servers
 - Think about Google, Amazon, Facebook, Netflix, etc.
- A *web application* is usually just the “front-end”, or a not so large application
- *Backend*: databases, (hundreds of) web services, load balancers, gateways, etc.

Enterprise 1 vs Enterprise 2

- In this E1 course, we start from building a web application that can use a SQL database
- In E2 next semester, we will look more into the backend, with RESTful APIs and microservices
- We focus more on backend than frontend, but still we will build full-stack applications

Technologies

- Java Enterprise Edition (**Java EE**) 7
 - Data layer (JPA/JTA)
 - Business logic layer (EJB)
 - Front-end layer (JSF)
- **SpringBoot** Framework
- Testing: **Selenium**
- Deployment: **Docker**

Spring vs Java EE

- **Java EE** was/is the “official” framework for building Java enterprise applications
- **Spring** is a different framework made by Pivotal that builds on top the foundations of Java EE
- As of now, Spring is more *widely* used, and also a *much more pleasant framework to work with*
- But you cannot really appreciate SpringBoot until you have gone through the blood, sweat and tears of debugging an EJB test using Arquillian to deploy to a Wildfly container...

New Versions

- In autumn 2017 many new versions came out, but not stable/widely supported yet
- But in this course, for this year...
- ... using Java 8 instead of 9
- ... using JEE 7 instead of 8
- ... using Spring 4 instead of 5
- ... using Junit 4 instead of 5

If You Skip Class...

- Usually acceptable that a student skips 1-2 classes
- You are supposed to attend, although no strict checks
- If you skip too many classes, it is **YOUR** responsibility to catch up and find out what done in class

Java Enterprise Edition (Java EE)

What is Java EE?

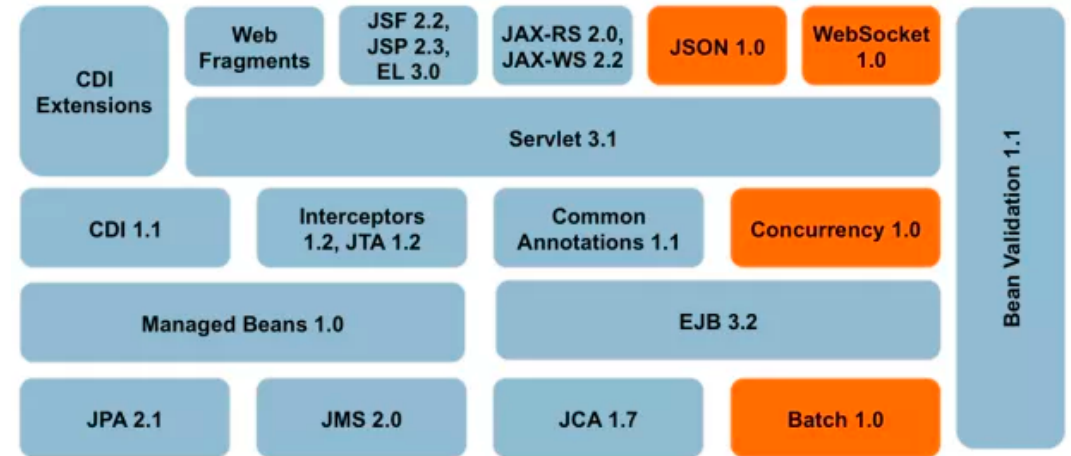
- It is a set of specifications of libraries for developing enterprise applications
- Think about it as a set of “interfaces”, with different possible implementations
 - Eg, Hibernate and EclipseLink are two different implementations for the JPA specs

History

- 1998: JPE (Java Platform for the Enterprise)
 - At Sun Microsystems, developer of Java
- 2009: Oracle buys Sun
- 2013: Java EE 7
- 2017: Java EE 8
- 2017: Oracle gives EE to the Eclipse Foundation

Java EE Specs in This Course

- **JPA: Java Persistence API**
 - For database accesses
- **Bean Validation**
 - For handling constraints on data
- **EJB: Enterprise Java Beans**
 - For business logic
- **Servlet**
 - To handle HTTP request
- **JSF: JavaServer Faces**
 - For building web GUIs
- But there is more...



JEE Vendors

- You build Java EE applications against its interfaces, but then you need to choose a *container* to run them
- Different vendors and implementations:
 - RedHat: JBoss and **Wildfly**
 - Oracle: GlassFish and WebLogic
 - IBM: WebSphere
 - Payara Services: Payara
 - Etc.

Why Containers???

- In an ideal world...
 - Files are smaller, as no need to package all the library implementations
 - Can run different EE applications on same container
 - Can deploy on different containers, and not get stuck with a single implementation
- In the real world...
 - Lot, lot of overhead in handling/configuring containers
 - Much worse testing: less automation, and mismatch between development and production environments
 - Changing container is far from simple...

“Partial” Containers: Web Servers

- Not supporting full Java EE specifications
- Mainly supporting **Servlet** and web assets
- **Tomcat** and **Jetty** are most famous/used ones
- Can add needed EE as libraries (eg, Hibernate for JPA)
- Can be **embedded** with the application
 - I.e self-executable JAR files
- Approach used by SpringBoot Framework

Maven

Course Repository

- The Git course repository uses **Maven**:
https://github.com/arcuri82/testing_security_development_enterprise_systems
- More than **100** Maven submodules, with several layers of nesting
- Not uncommon to see something like that in large enterprise systems
- Need to understand how Maven works

Build Tools

- **Maven**

- *Most popular*, XML based, my build tool of choice
- Verbose, but not a big problem with autocomplete in IntelliJ

- **Gradle**

- Popular in Android, script based
- Being scripts is its **best** and **worst** feature
 - Good: highly flexible
 - Bad: harder to maintain and use for new developers in a project
- Note: still handling Maven dependency libraries

- **Ant**

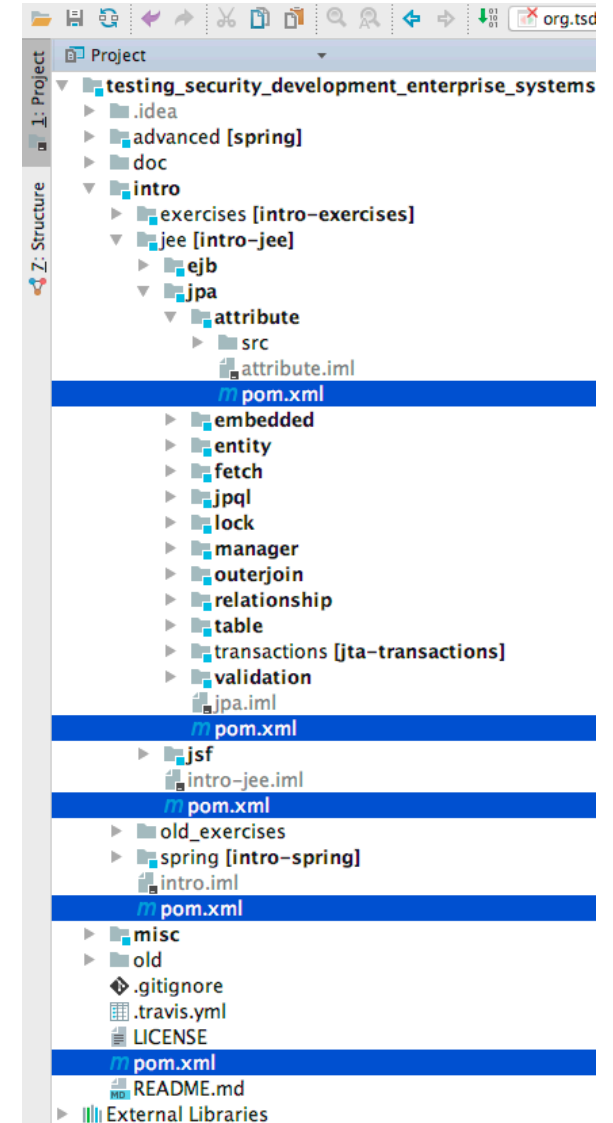
- Old, not so much in use any more

Role of A Build Tool

- Compile your code
- Handle complex modularization
- Automatically download all needed third-party libraries
- Apply custom pre/post processing
- Run test cases
 - Eg, Continuous Integration, fail whole build if any test is failing
- *Easy to checkout and automatically build your project on a new machine*

Maven “pom.xml” files

- POM: Project Object Model
- XML file describing how to build a *module*
- Project can be composed of several modules, with each module having its own pom.xml file
- Hierarchy of modules and submodules



```
<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/xsd/maven-4.0.0.xsd">
```

```
<modelVersion>4.0.0</modelVersion>
```

```
<groupId>org.tsdes</groupId>
```

```
<artifactId>tsdes</artifactId>
```

```
<version>0.0.1-SNAPSHOT</version>
```

```
<packaging>pom</packaging>
```

```
<name>Root of TSDES</name>
```

```
<modules>
```

```
  <module>misc</module>
```

```
  <module>intro</module>
```

```
  <module>advanced</module>
```

```
</modules>
```

...

- **<project>** defines a series of namespaces and XSD (XML Schema Definition)
- A pom.xml does not contain all kinds of XML tags, but only the ones defined in the XSD schema
- When the Maven command runs, it parses the content of the pom.xml file in current directory

Module/Artifact Coordinates

- Each module/artifact is uniquely identified by 3 tags
- **<groupId>**: a string id identifying a group of related artifacts
- **<artifactId>**: an id that is unique within a group
- **<version>**: the version of the module/artifact, usually in the M.m.p numeric format, ie, Major-Mino-Patch version
 - Usually ending with SNAPSHOT if under development, and not published

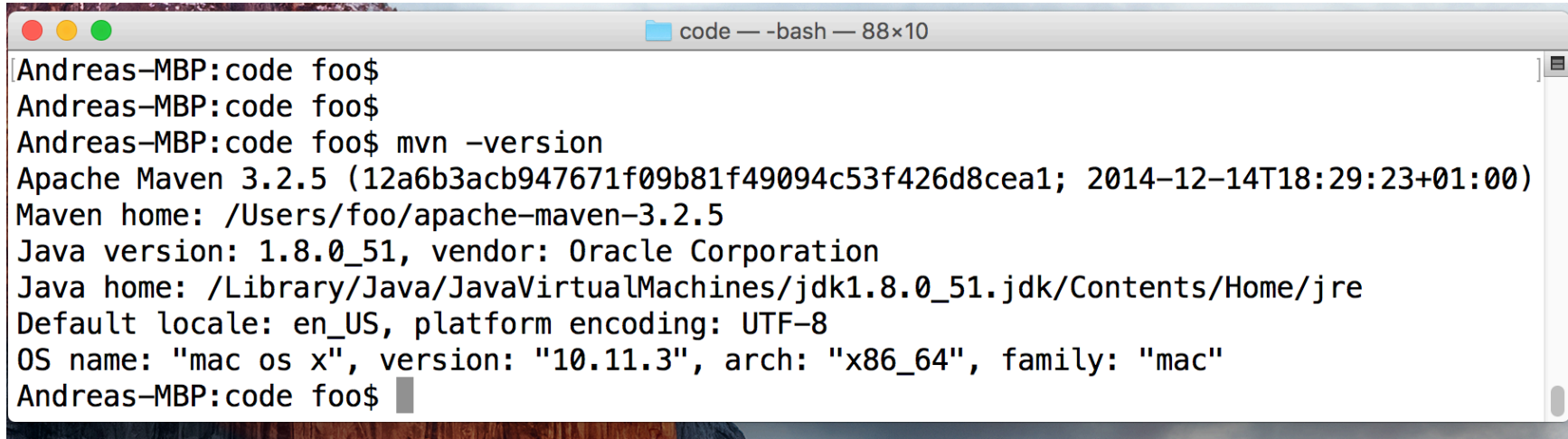
```
<groupId>org.tsdes</groupId>  
<artifactId>tsdes</artifactId>  
<version>0.0.1-SNAPSHOT</version>
```

Type of Packaging

- **<packaging>**: having value either **pom**, **war** or **jar**
- **pom**: this module is responsible to build other modules, specified in **<modules>** tag
 - Useful to share settings that are common among different sub-modules
- **war**: module creates a WAR (Web application ARchive) file
 - These are the files deployed on EE containers like Wildfly
- **jar**: module creates a JAR (Java ARchive) file
 - A single file containing your compiled code

Using Maven

- You can run Maven from an IDE, but **BEST** to learn to use it from command line
- Need to download recent version
- As developers, there are many tasks that are simplified on the command line, or tools with no GUI
- We will go back on this point when dealing for example with self-executable jar files and Docker

A screenshot of a macOS terminal window titled 'code — -bash — 88x10'. The terminal shows the following commands and output:

```
Andreas-MBP:code foo$  
Andreas-MBP:code foo$  
Andreas-MBP:code foo$ mvn -version  
Apache Maven 3.2.5 (12a6b3acb947671f09b81f49094c53f426d8cea1; 2014-12-14T18:29:23+01:00)  
Maven home: /Users/foo/apache-maven-3.2.5  
Java version: 1.8.0_51, vendor: Oracle Corporation  
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_51.jdk/Contents/Home/jre  
Default locale: en_US, platform encoding: UTF-8  
OS name: "mac os x", version: "10.11.3", arch: "x86_64", family: "mac"  
Andreas-MBP:code foo$
```

- Make sure you can run Maven from a terminal / console / command line
- On Windows, you might want to try out GitBash
- You need to configure the PATH environment variable to be able to use Maven from command line
- If everything is configured, run “mvn -version”

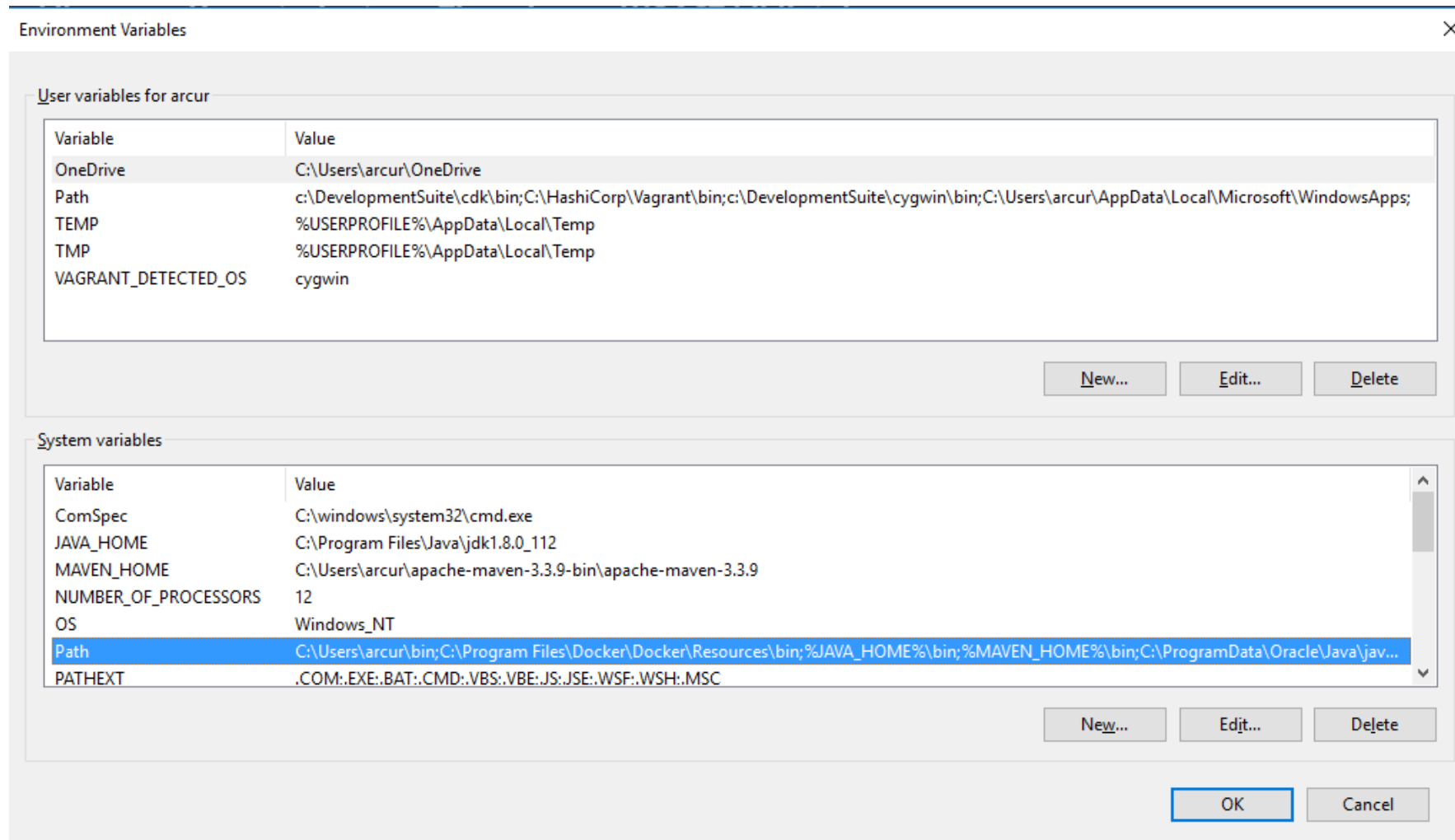


```
###For separated histories for tab
MYTTY=`tty`
HISTFILE=$HOME/.bash_history_`basename $MYTTY`

export M2_HOME=/Users/foo/apache-maven-3.2.5
export M2=$M2_HOME/bin
export PATH=$M2:$PATH
MAVEN_OPTS='-Xmx512m' ; export MAVEN_OPTS

export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_51.jdk/Contents/Home/
```

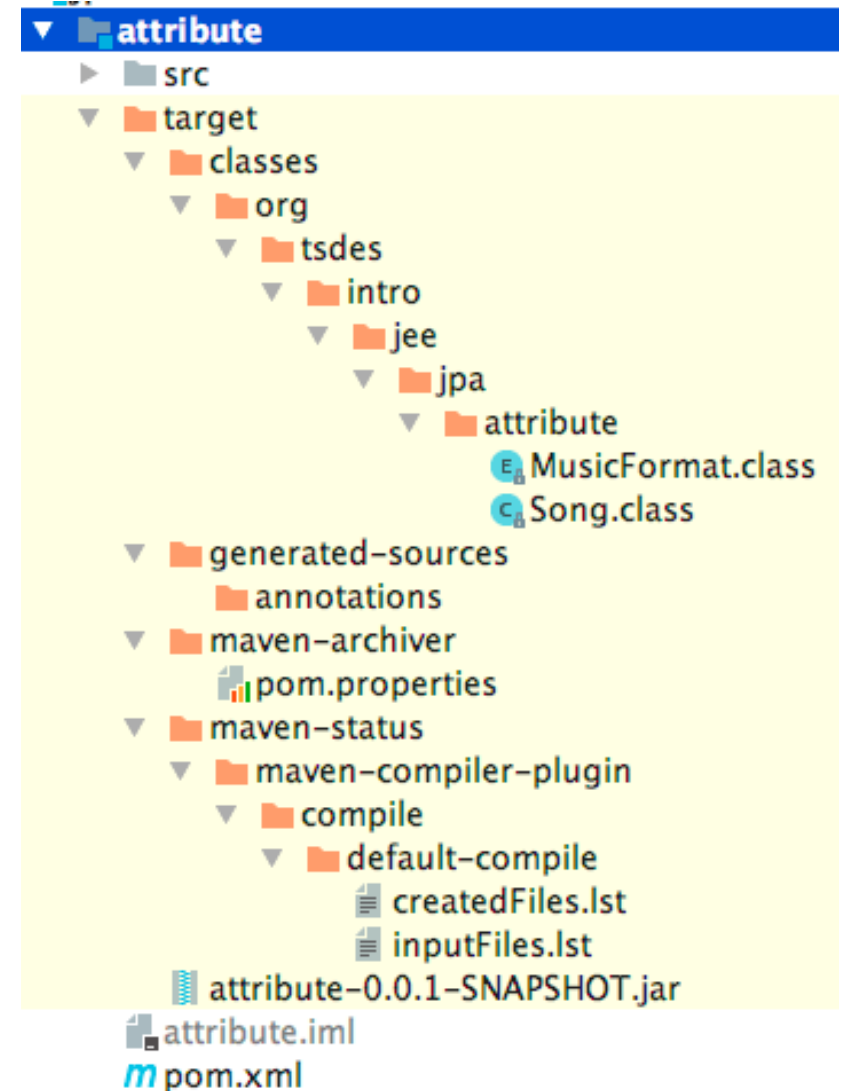
- On Mac, need to edit the “.profile” file under your home directory
- Of course, actual paths depend on where you install the JDK and Maven...



In Windows, setup environment variables for MAVEN_HOME and JAVA_HOME, and then update PATH

Mvn Clean

- “mvn clean”
- Used to “clean” your project, ie delete all generated files
- When you build a project, a “target” folder is created, where all compiled files (.class files) and other built artifacts (eg, JAR and WAR files) are stored



Maven Main Phases

1. **compile**

- compile all your .java files into .class

2. **test**

- run all the *unit* tests

3. **package**

- create a WAR/JAR file

4. **verify**

- run all the *integration* tests

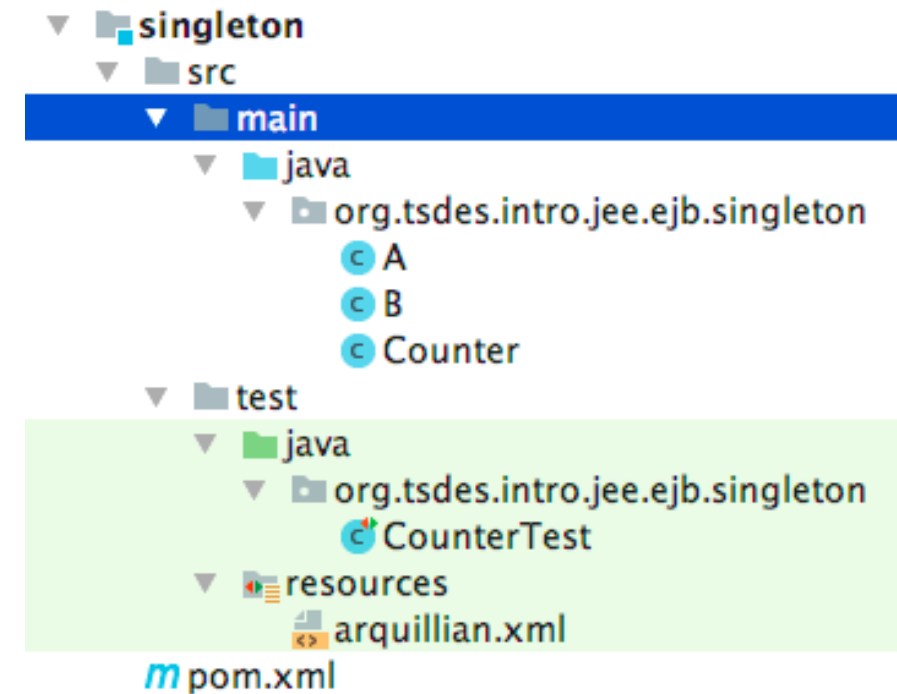
5. **install**

- copy the packaged WAR/JAR into your local Maven repository

- When you run a phase like “**mvn package**”, all the previous phases are executed as well
- Note: there are more phases, but these here are the most important ones

Convention Over Configuration

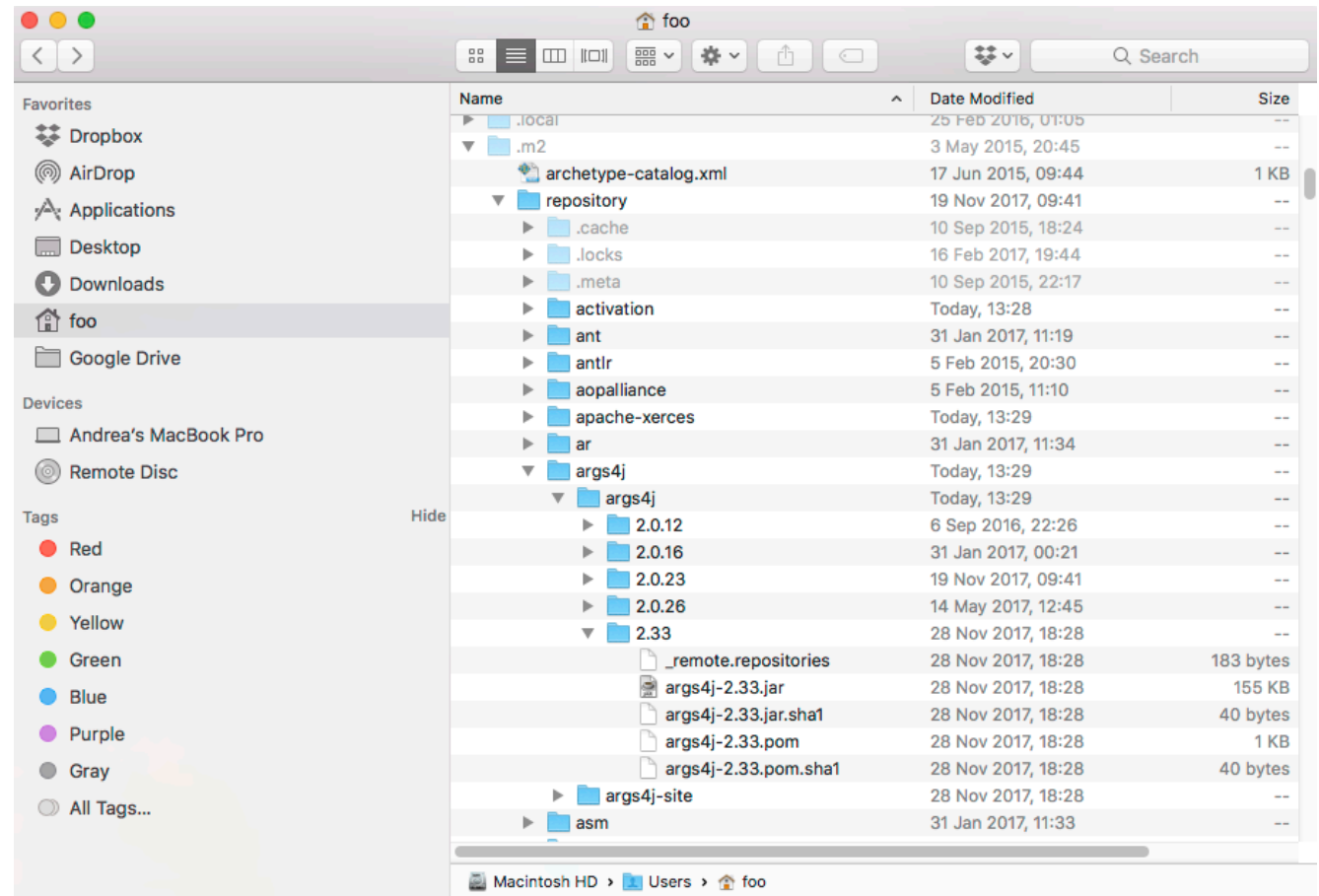
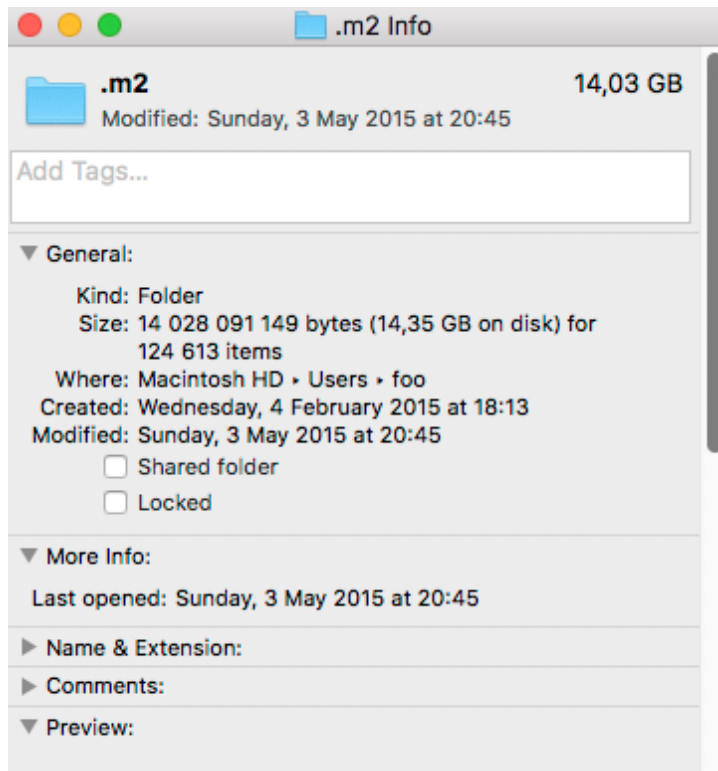
- Maven expects a clear structure of where to put your files
- **src/main/java**: your Java source files
- **src/main/resources**: files that will be added into the JAR/WAR file
- **src/test/java**: sources of test classes
- **src/test/resources**: resources for tests
- You can change these defaults, but not recommended



Third-Party Libraries

- One of the main benefits of Maven is to automatically download dependencies
- Added on pom.xml files
- Maven will check if such dependency jar is in your “~/.m2” folder
 - “~” is home folder of your user account
- If not, it will be downloaded

```
<dependency>  
  <groupId>junit</groupId>  
  <artifactId>junit</artifactId>  
  <version>4.12</version>  
  <scope>test</scope>  
</dependency>
```



- With passing of years, it can grow large (eg 14 GB in my case)
- “.” in front of a folder/file makes it “hidden” in Mac/Linux
- The “2” just refers to old Maven 2.x version (3.x is backward compatible, but 1.x was not)

Main Dependency Scopes <scope>

- **compile**: default one
- **provided**: needed at compilation, but will not be included in generated JAR/WAR files. Expected to be provided by the runtime (eg, a Java EE container)
- **test**: needed only for testing, not in the generated JAR/WAR files
 - eg, JUnit library to run test cases
- **import**: used for POM dependencies, imported and embedded from the pom.xml of the dependency
 - Used by libraries with many related dependencies, so you do not need to add each single of them manually

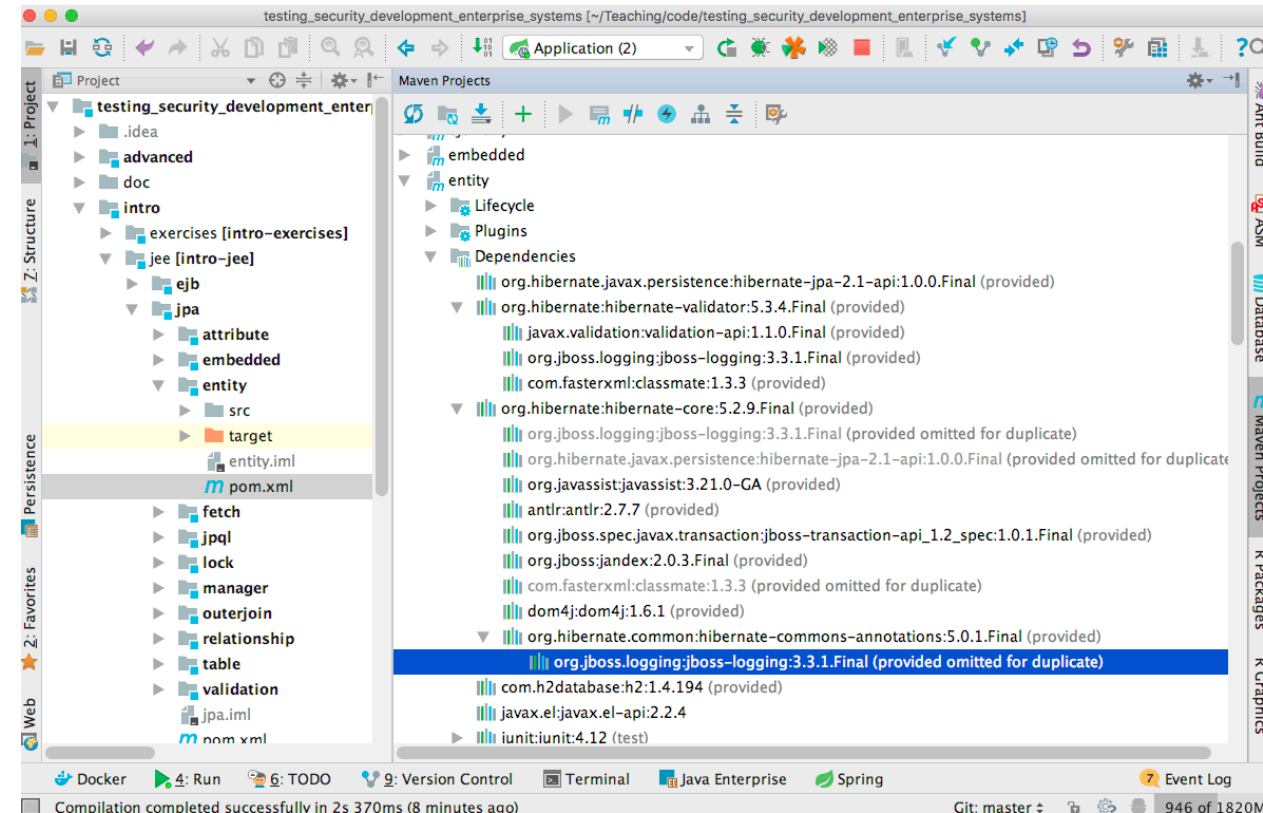
```
// in intermediate pom.xml
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
```

```
//in modules building JAR/WAR
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
  </dependency>
```

- You might use a library in many, many modules
- To avoid copy&paste and maintain **<version>/<scope>** everywhere, use **<dependencyManagement>** in a shared ancestor pom.xml, eg the root one
- All submodules will inherit the **<version>/<scope>** values

Transitive Dependencies

- A dependency JAR can have its own dependencies.
- And these transitive dependencies can have their own dependencies, and so on...
- Can use IntelliJ “Maven Projects” view to see exactly what is used in a module
- For example: *hibernate-core* pulls in *hibernate-commons-annotations*, which pulls in *jboss-logging*



Properties \${}

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <fs>${file.separator}</fs>
  <version.java>1.8</version.java>
  <version.jacoco>0.7.9</version.jacoco>
  <version.javax.el>2.2.4</version.javax.el>
  <version.hibernate.jpaa>1.0.0.Final</version.hibernate.jpaa>
  <version.hibernate.core>5.2.9.Final</version.hibernate.core>
  <version.hibernate.validator>5.3.4.Final</version.hibernate.validator>
  <version.h2>1.4.194</version.h2>
  <version.postgres>42.1.4</version.postgres>
  <version.resteasy>3.1.3.Final</version.resteasy>
  <version.testcontainers>1.4.3</version.testcontainers>
</properties>
```

```
<dependency>
  <groupId>javax.el</groupId>
  <artifactId>javax.el-api</artifactId>
  <version>${version.javax.el}</version>
</dependency>
<dependency>
  <groupId>org.glassfish.web</groupId>
  <artifactId>javax.el</artifactId>
  <version>${version.javax.el}</version>
</dependency>
```

Plugins

- Used to extend the functionalities of Maven
- Plugins are downloaded and configured like any third-party library
- Many of the base functionalities of Maven are themselves represented as plugins
 - Eg, compile Java code

```
<pluginManagement>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <inherited>true</inherited>
      <configuration>
        <source> ${version.java} </source>
        <target> ${version.java} </target>
      </configuration>
    </plugin>
  </plugins>
</pluginManagement>
```


Surefire and Failsafe

- Surefire: plugin to run *unit* tests
 - By default, all files in src/main/test with name pattern *Test.java
 - Unit tests are run before the **package** phase
- Failsafe: plugin to run *integration* tests
 - By default, all files in src/main/test with name pattern *IT.java
 - Integration tests are run after the **package** phase, so can use the packaged JAR/WAR files
- Note: in both cases, still writing them with JUnit
- **WARNING:** If you misspell *Test.java/*IT.java, tests will not run from Maven

Build Course Git for First Time

- From root folder: “**mvn clean install -DskipTests**”
- It will recursively build all the modules
- **clean**: just make sure you start from a clean state
- **install**: it executes all previous phases, including **compile** and **package**
- **-DskipTests**: avoid running tests
- WARNING: first time, it will take a long while, as many libraries will need to be downloaded

JPA: Java Persistence API

Object-Relational Mapping (ORM)

- Mapping data from SQL Databases (DB)
- In your programs, using Java classes to represent data from DB
- Idea of JPA: define *@Entity* classes for each table in the DB, and let the JPA framework do all the work to read/write to/from DB when modify the *@Entity* classes
- In theory, no need of SQL. But still might want to use in some cases (eg, for complex queries), or when the JPA implementation gives *weird* results...

Hibernate

- A JPA implementation
- Most popular in Java
 - Default in Java EE containers like Wildfly
 - Default in SpringBoot
- As a library, can be used in any Java program
 - ie, not necessarily in EE or Spring
- **src/main/resources/META-INF/persistence.xml**
 - Configuration file for JPA

Database Schema

- Given an existing DB, need to write *@Entity* classes for each table
- Other option: write the *@Entity* classes first, and generate the schemas automatically afterwards
 - Easier when you are more familiar with Java than SQL
 - Good for prototyping

Git Repository Modules

- *NOTE: most of the explanations will be directly in the code as comments, and not here in the slides*
- **intro/jee/jpa/entity**
 - @Entity, @Id, @GeneratedValue, etc
- **intro/jee/jpa/table**
 - @Table, @Column, etc.
- **intro/jee/jpa/embedded**
 - @Embeddable, @Embedded, @EmbeddedId, etc.
- **intro/jee/jpa/attribute**
 - @Lob, @Enumerated, @Temporal, @Transient, etc.
- Exercises for Lesson 01 (see documentation)