Enterprise Programmering 1

Lesson 01: Introduction

Prof. Andrea Arcuri

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
- Module "**intro**"
- *Note: pull often, as material can get updated throughout the course*

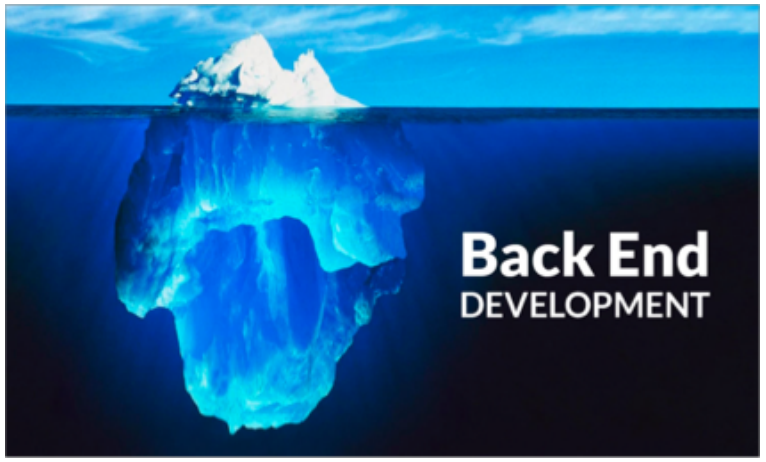
Necessary Tools

- JDK **11**
- 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

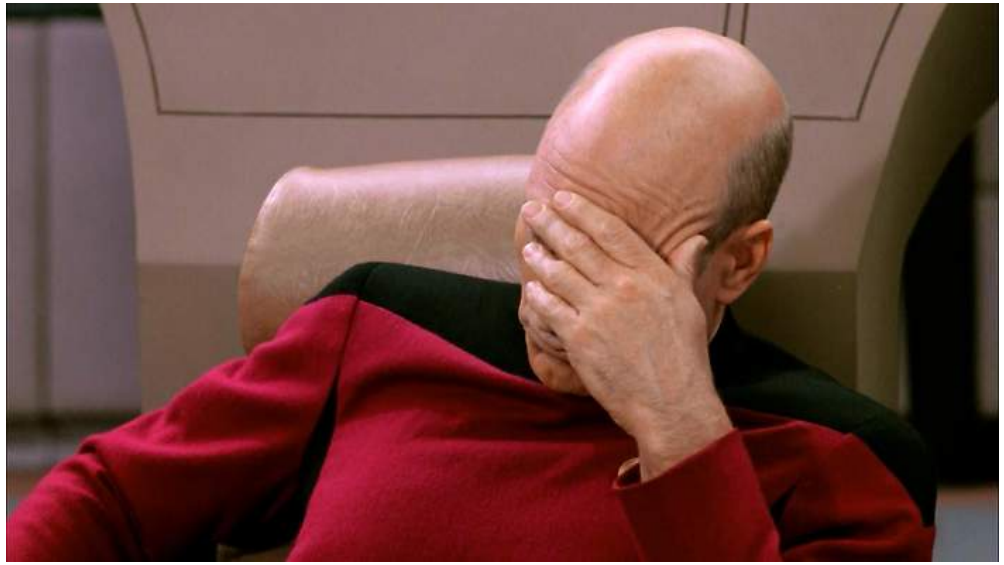
- Learn about developing *enterprise* applications, with a focus on how to **test** and **secure** them
- Access to databases
- Business logic with enhanced beans
- *Server-side* rendering with HTML templates
- **Not only studying theory, but being able to build and deploy a full application**

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 "*frontend*", or a not so large application
- *Backend*: databases, (hundreds of) web services, load balancers, gateways, etc.

Unfortunately, nothing to do with the “real” Enterprise…



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

Enterprise 1 vs Web Development

- In "*Web Development and API Design*" (WDAD) we focus on modern frontend with *client-side* rendering using *Single-Page-Applications* (SPA), written in JavaScript
  - plus intro to *REST* and *GraphQL*
- E1 looks at the more *historical* perspective of *server-side* rendering with no JS (just HTML/CSS)
- WDAD can be seen as a course between E1 and E2
  - although most of you are taking it in parallel with E1

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

Exam

- Home project
- 48 hours

Technologies

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

Why Java?

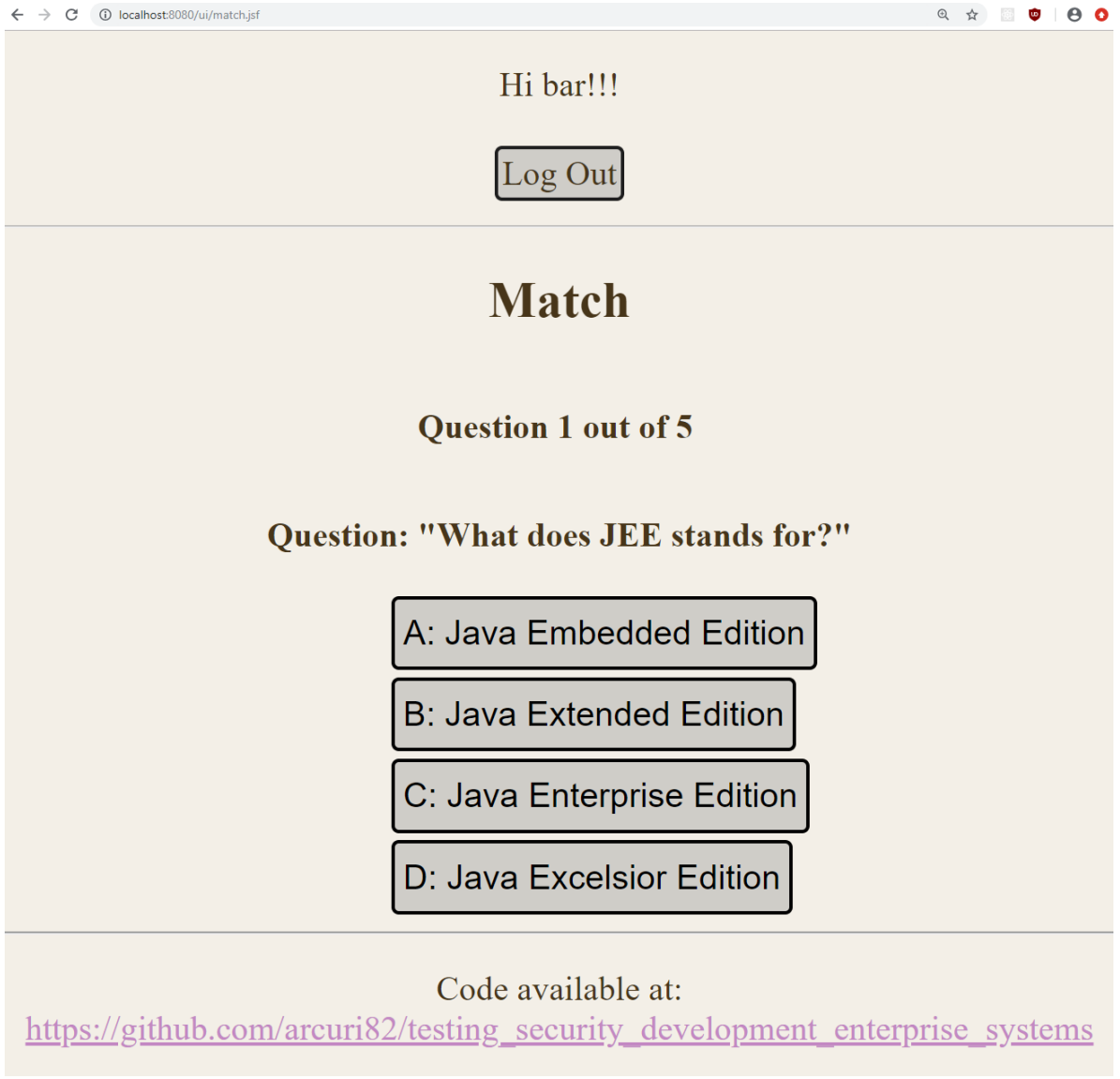
- One of the most popular and used programming language
- The main language for *enterprise backend* development
- *C#* in *.Net* is currently a very good alternative
  - many concepts we will learn do apply as well to C#/.Net, although of course with different syntax/libraries/frameworks
- Need for *good, strongly-type languages*
  - and so **no** JavaScript, Python, Ruby, etc.

Spring vs JEE

- **JEE** was the “official” framework for building Java enterprise applications, but not anymore (since 2017)
  - now it is just a framework like any other
- **Spring** is a different framework made by Pivotal that builds on top the foundations of JEE
- 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…

Exercises

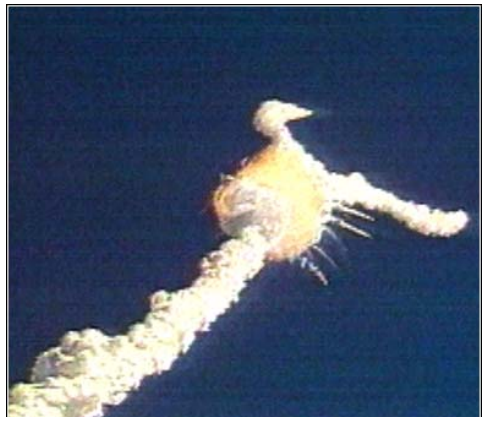
- We will build a full web application, week by week
- Access to database
- Server-side HTML rendering (JSF)
- Cloud deployment
- Can start it from *intro/exercises/quiz-game/part-11/frontend/src/test/java/org/tsdes/intro/exercises/quizgame/LocalApplicationRunner.java*
- *localhost:8080*



Not Just Development…

- In this course, **STRONG** emphasis on **TESTING** and **SECURITY**
- *But why???*

Ariane 5



On June 4, 1996, the flight of the Ariane 5 launcher ended in a failure.

$500 millions in cost

Software bug

Fatal Therac-25 Radiation

1986, Texas, person died

Power Shutdown in 2003

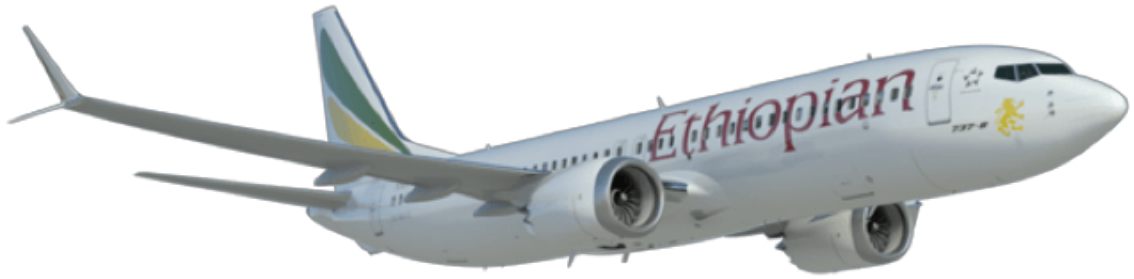
Nearly 50 millions persons affected in Canada/US

2010, Toyota, software bug in braking system, recalled 436,000 vehicles

Knight Capital Group 2012

$460 millions lost in 45 minutes of trading due to bug

2019, a Boeing 737 Max **crashed** due to **software problems**; all **157 people on board died**.



And I could go on the whole day…

- As of 2013, estimated that software testing costing **$312 billions** worldwide
- In 2016, 548 recorded and documented software failures impacted **4.4 billion** people and **$1.1 trillion** in assets worldwide

But what about every-day life in Oslo???

Security

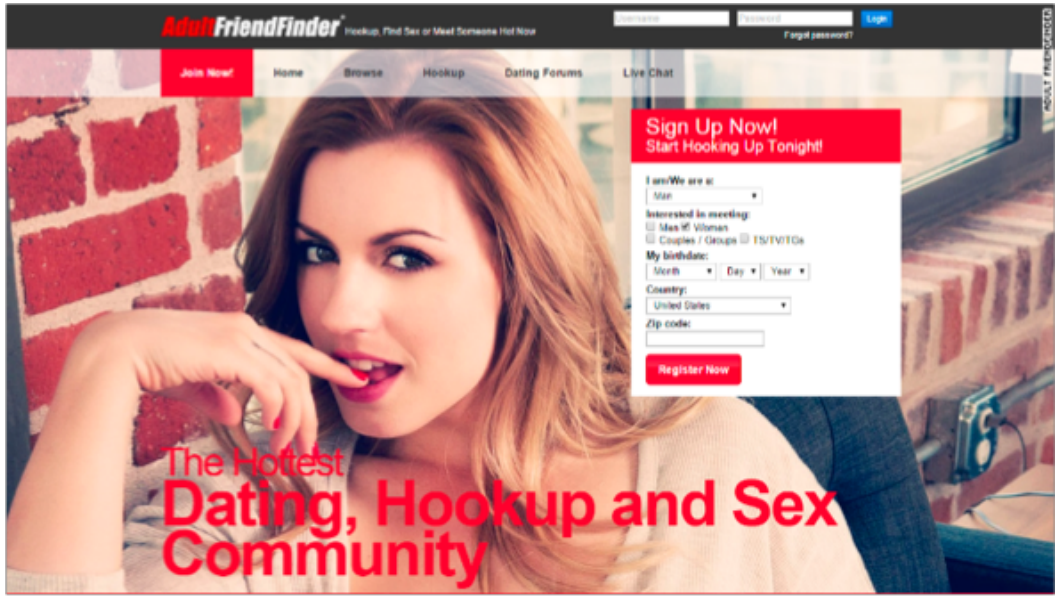
- Those in previous slides were "*functional*" bugs
- But security holes in enterprise applications are bugs as well
- Not only need to know about best practices in security and different kinds of attacks, but also how to *protect* your apps

2013-2014: YAHOO!

- 3 **B**illions accounts compromised
  - yes, it is **B**s, not **M**illions
- Stealing information is bad, especially if *credit card* numbers
- But what is another major issue? People *re-use same password* for different sites, eg Gmail, Facebook, etc.
- Yahoo had easy to crack hashed passwords (eg, MD5)
- *How many of you use a different password for each different service???*

2016: Adult Friend Finder

- Web site for “hook-ups”
- 412 million accounts were compromised
- Users got blackmailed
  - eg, people cheating on spouses
  - some even committed suicide…



2014: eBay

- 145 million accounts compromised
- But fortunately credit card numbers were stored in different databases

2017: Equifax

- 143 million consumer info breached
- Personal information
  - including Social Security Numbers, birth dates, addresses, and in some cases drivers' license numbers
- 209,000 consumers also had their credit cards exposed

2011: Sony's PlayStation Network

- 77 million accounts hacked
- 12 million unencrypted credit card numbers got stolen
- Site down for a month
- Around $200 million in losses

Java Enterprise Edition (JEE)

What is JEE?

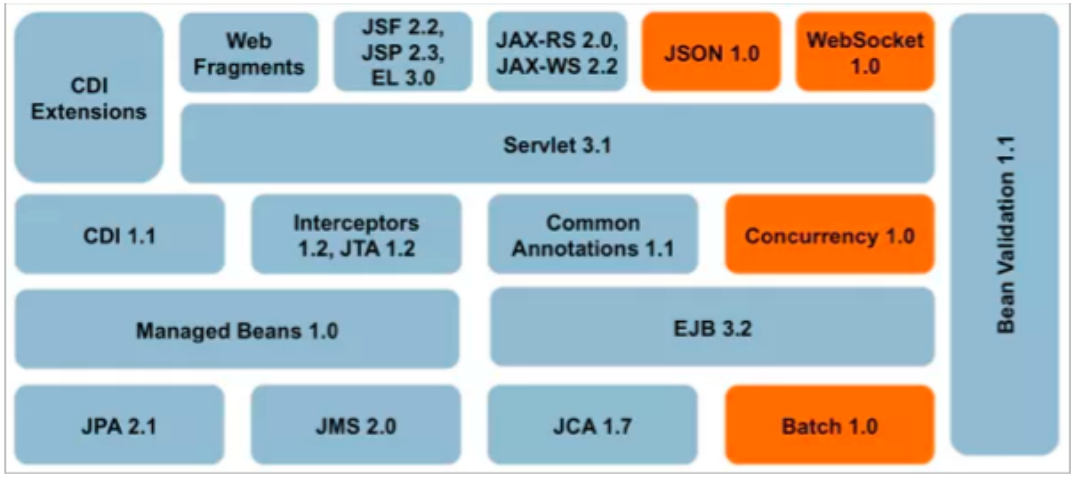
- It is a set of specifications of libraries for developing enterprise applications
- Think about it as a set of “interfaces”, with different possible implementations
  - Eg, *Hibernate* and *EclipseLink* are two different implementations for the JPA specs

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
  - now it is called “*Jakarta EE*”

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
    - Note that in 2018 IBM did buy RedHat…
  - Payara Services: Payara
  - Etc.

Why Containers???

- In an ideal world…
  - JAR/WAR 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 JEE 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
  - Ie 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 **150** 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

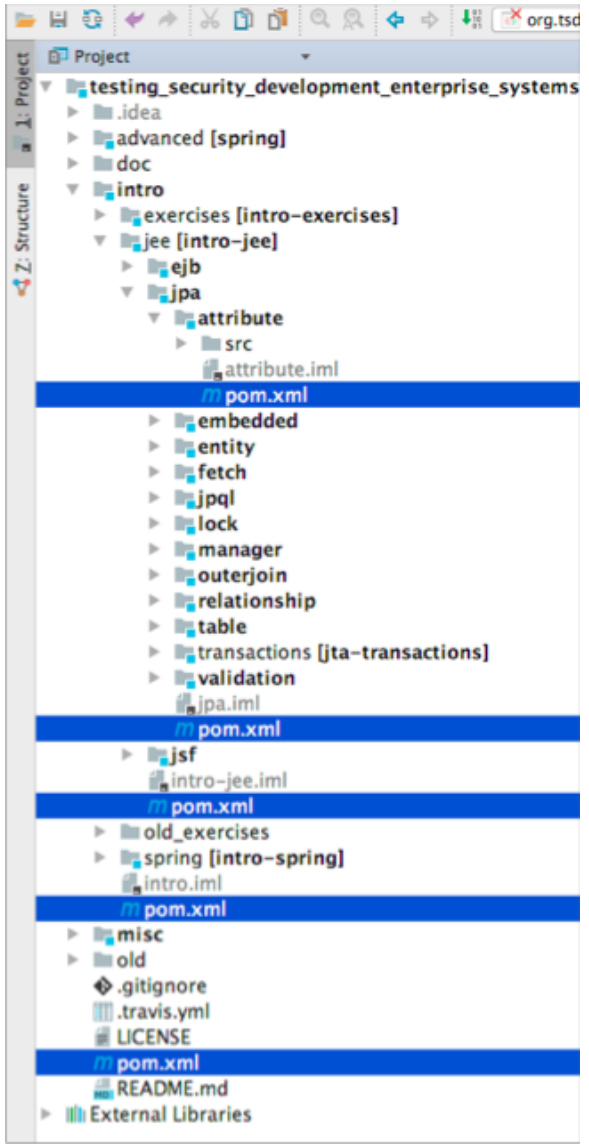
- **Maven**
  - *Most popular*, XML based, my build tool of choice
  - Verbose, but not a big problem with autocomplete in IntelliJ
- **Gradle**
  - Popular in Android, script based
  - Being scripts is its **best** and **worst** feature
    - Good: highly flexible
    - Bad: harder to maintain and use 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

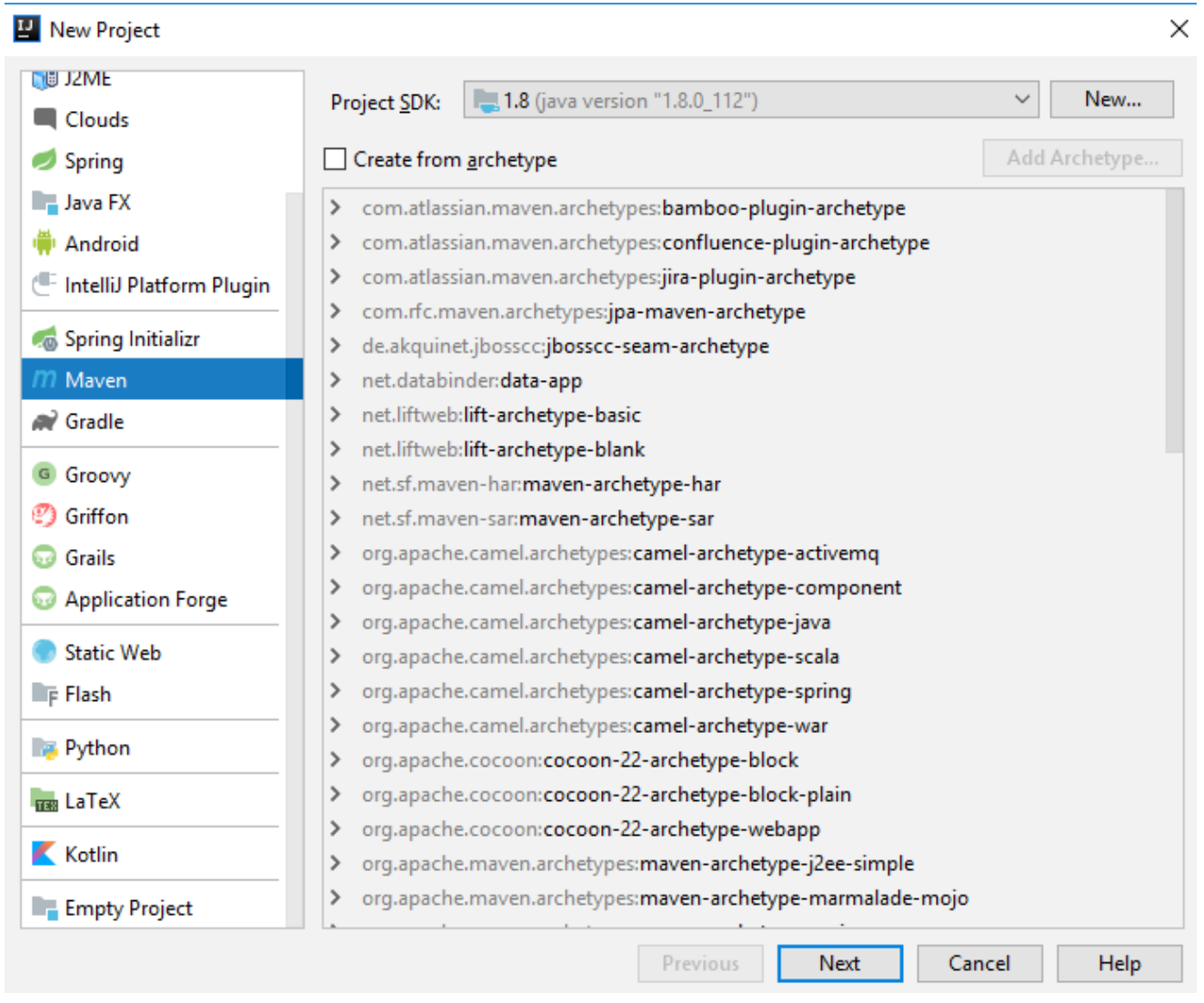


```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0     http://maven.apache.org/xsd/maven-4.0.0.xsd">

   <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

New Projects

- Can ask your IDE to create a *pom.xml* for you…
- … or can just copy&paste&*adapt* an existing one



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-Minor-Patch version
  - Usually ending with SNAPSHOT if under development, and not published

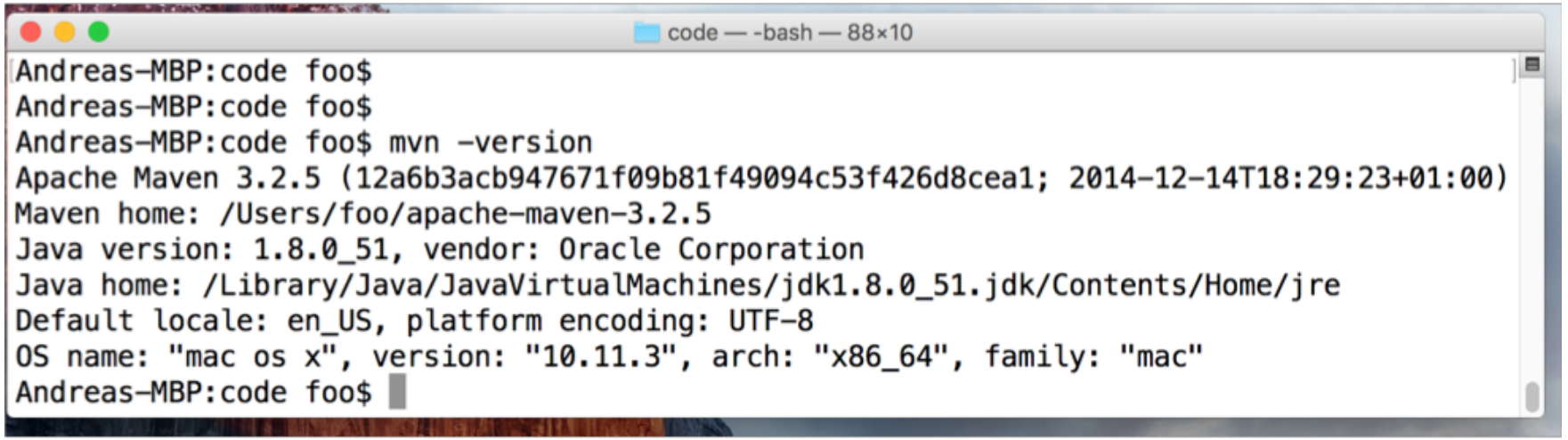
<**groupId**>org.tsdes</**groupId**>
<**artifactId**>tsdes</**artifactId**>
<**version**>0.0.1-SNAPSHOT</**version**>

Type of Packaging

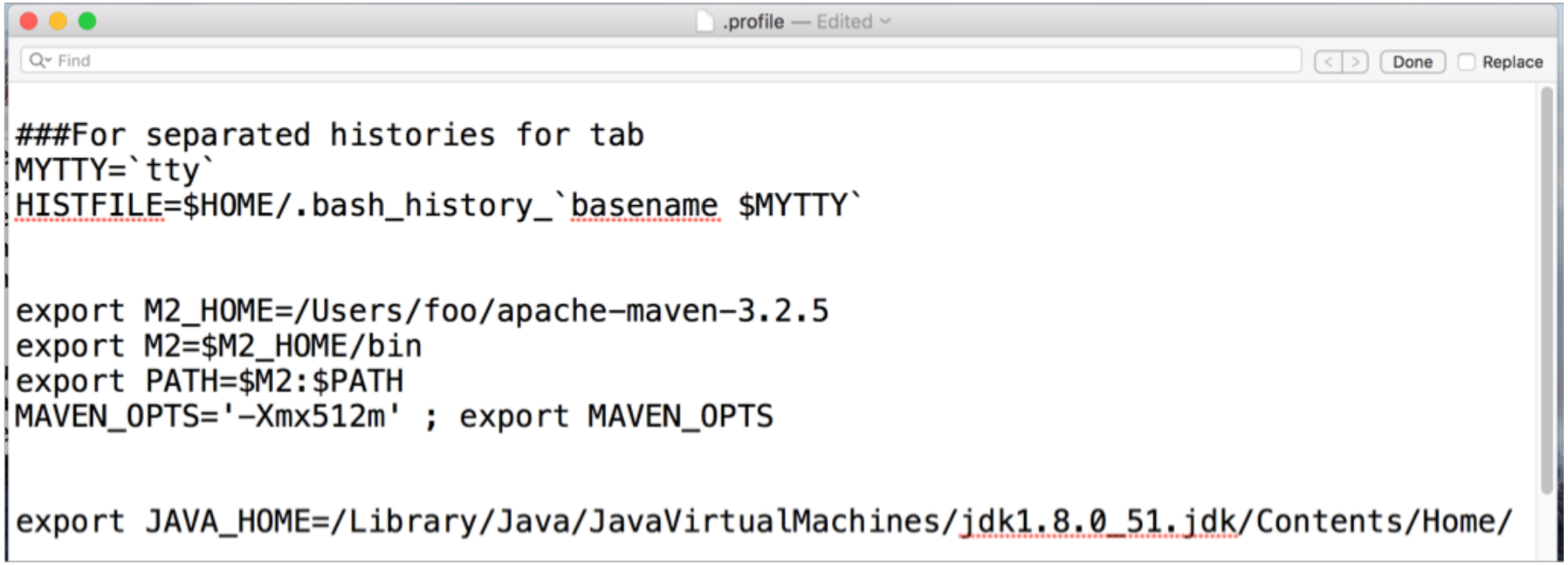
- **<packaging>**: having value either **pom**, **war** or **jar**
- **pom**: this module is responsible to build other modules, specified in **<modules>** tag
  - Useful to share settings that are common among different sub-modules
- **war**: module creates a WAR (**W**eb application **AR**chive) file
  - These are the files deployed on EE containers like Wildfly
- **jar**: module creates a JAR (**J**ava **AR**chive) file
  - A single file containing your compiled code
  - Technically, it is a zipped file, so can use "*unzip*" to open it

Using Maven

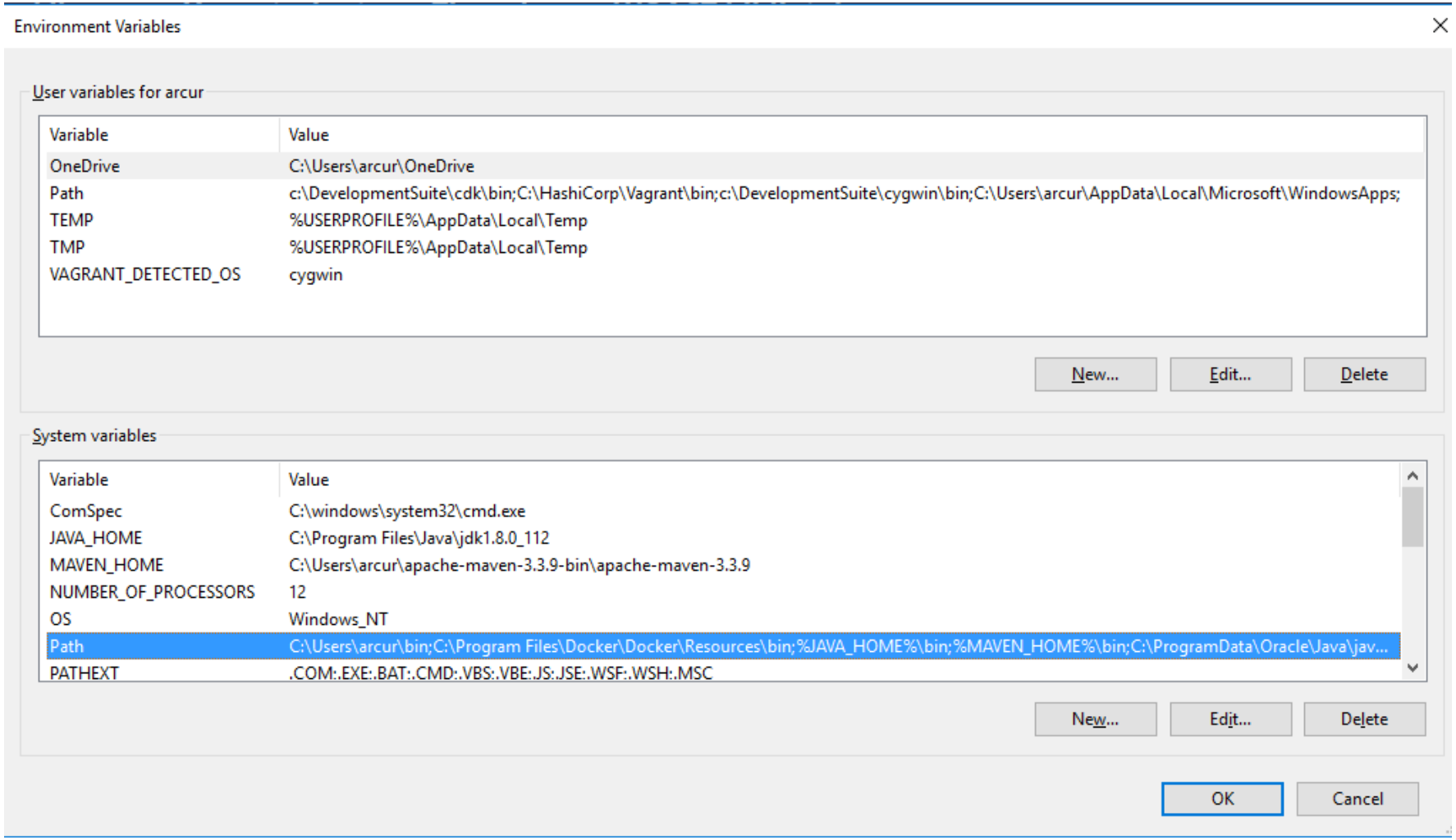
- You can run Maven from an IDE, but **BEST** to learn to use it from command line
- Need to download recent version
- As developers, there are many tasks that are simplified on the command line, or tools with no GUI
- We will go back on this point when dealing for example with self-executable jar files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



- 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*"



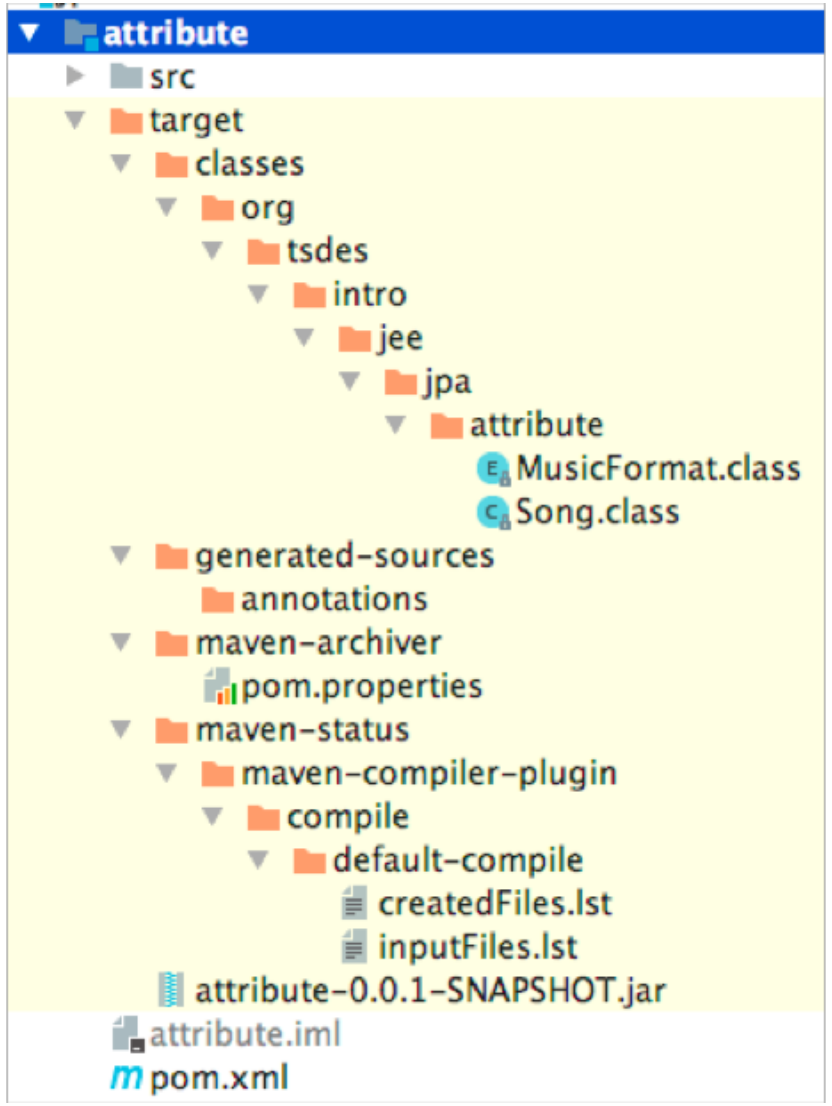
- On Mac, need to edit the "*.profile*" file under your home directory
- Of course, actual paths depend on where you install the *JDK* and *Maven*…



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



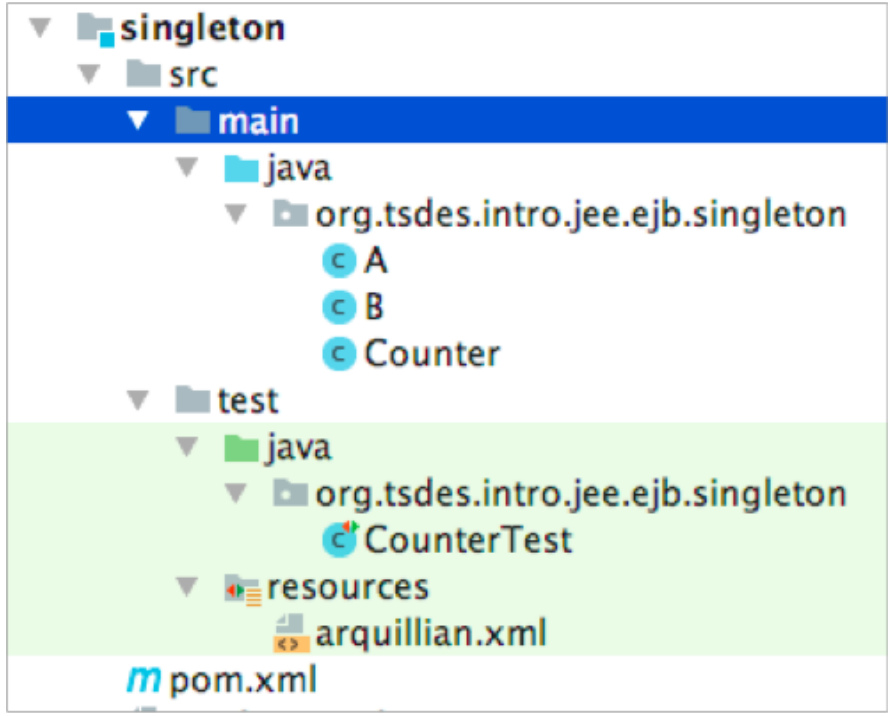
Maven Main Phases

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* expect 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

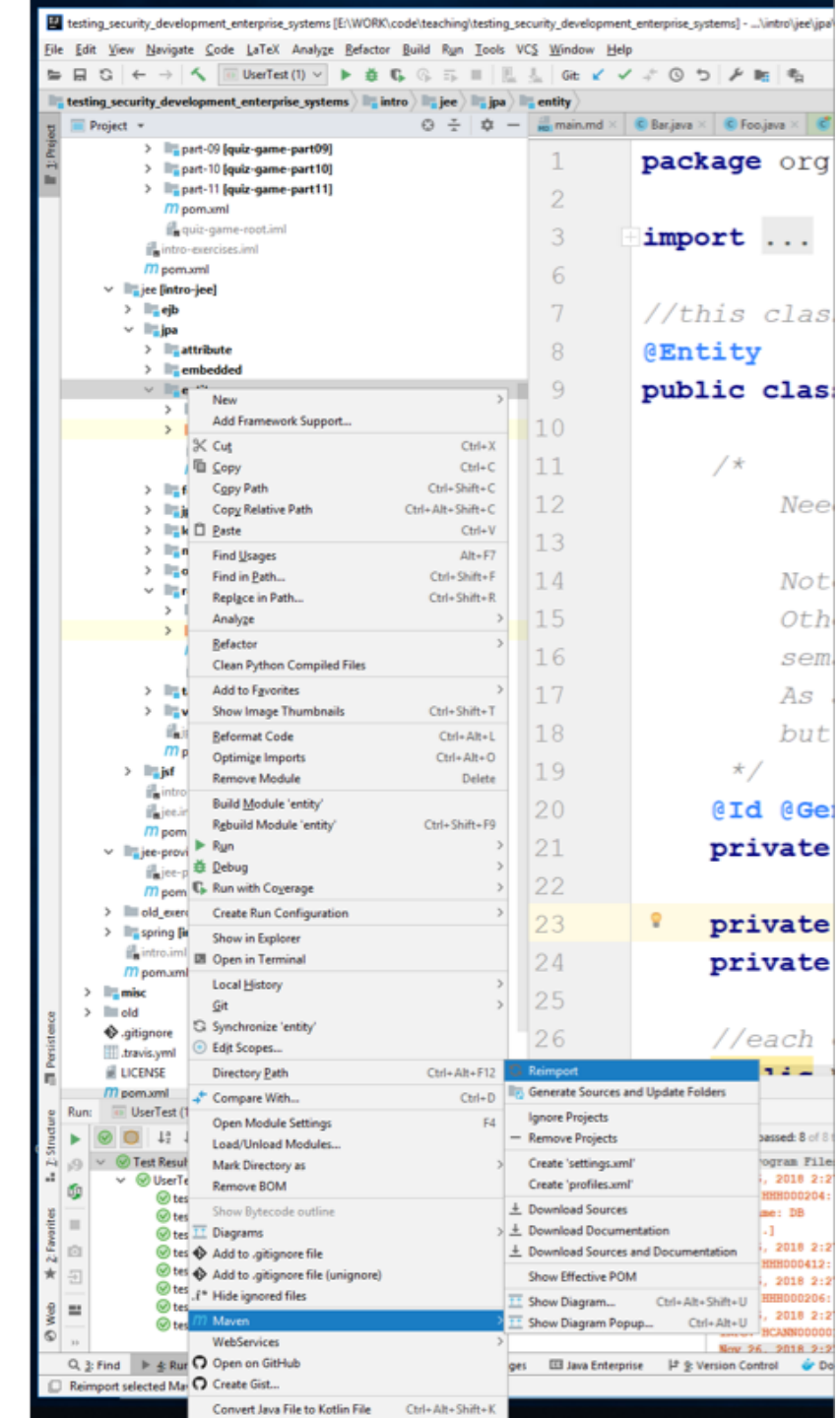


Maven from IDE

- When you open a *Maven* project from an IDE, usually IDEs will automatically configure it for you
- Need to make sure *Maven* support is activated
  - right-click on Project and choose *Maven* from "*Add Framework Support…*"
- Example: if you add **src/test/java** after a project was opened, the IDE might not know it is a test folder
  - should activate automated updates of *Maven*, usually asked first time you open a project with a *pom.xml* file

If IDE Misconfigured

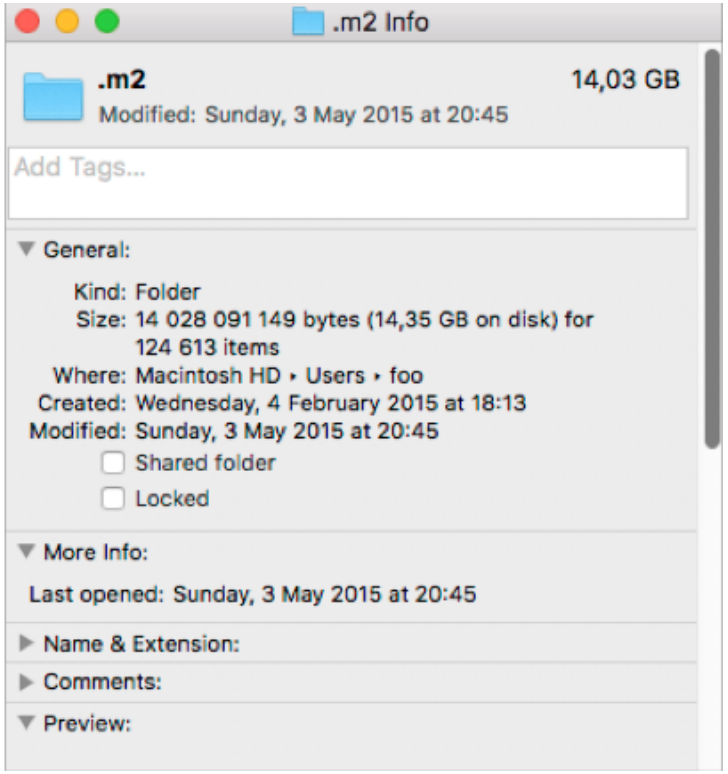
- If IDE does not automatically pick up changes, you can force a Maven *reimport*
- **DO NOT** mark folders manually, eg using "*Mark Directory as*" in IntelliJ

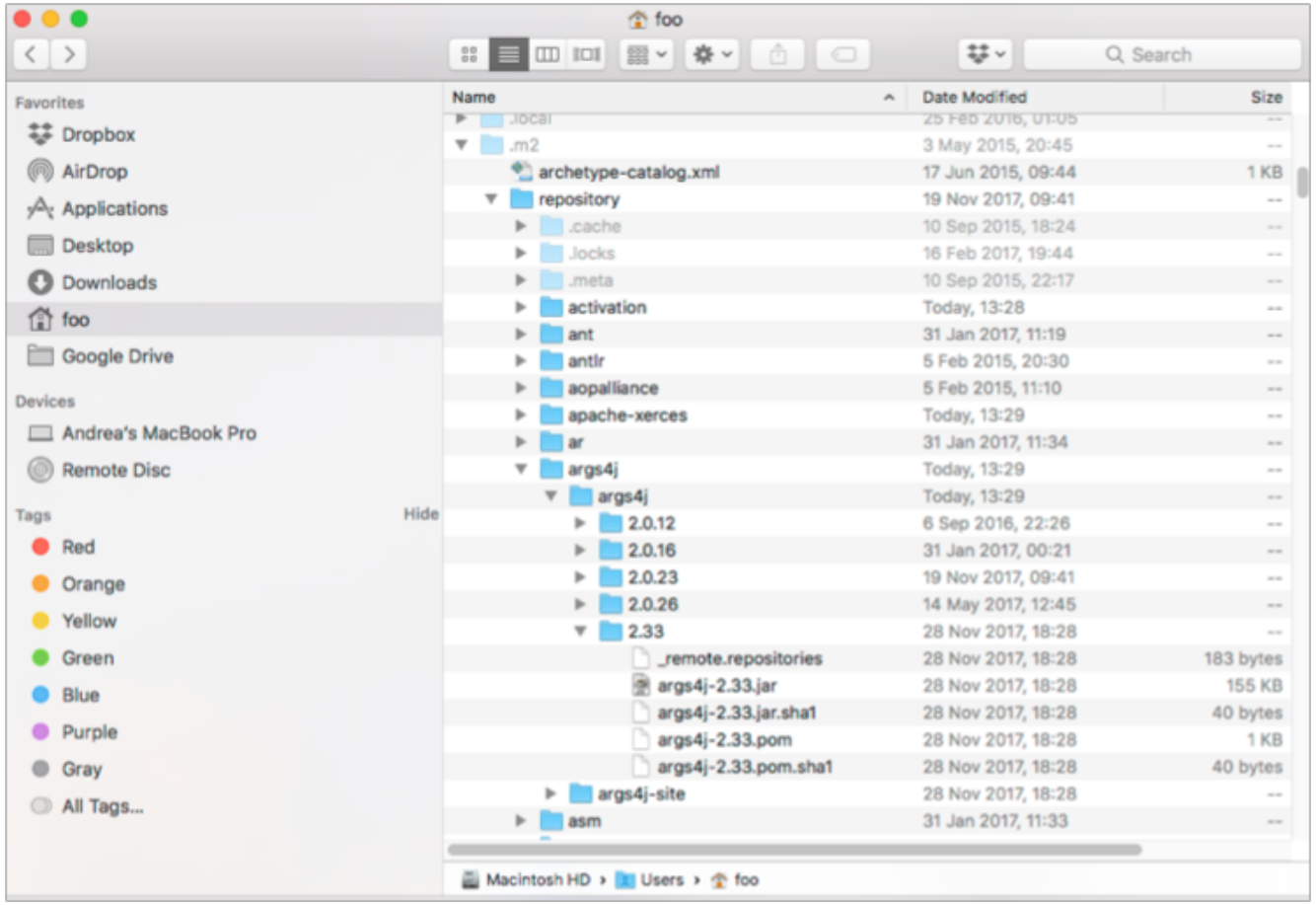


Third-Party Libraries

- 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 JEE 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

Bill of Materials (BOM)

- Example of Spring
- Many dependencies, want to keep them in version sync
  - ie, if 100 related dependencies, with same version number, then want to update version only in 1 place to get all those related dependencies updated and in sync

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>${version.springboot}</version>
    <scope>import</scope>
    <type>pom</type>
</dependency>
```

```
// 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>
```

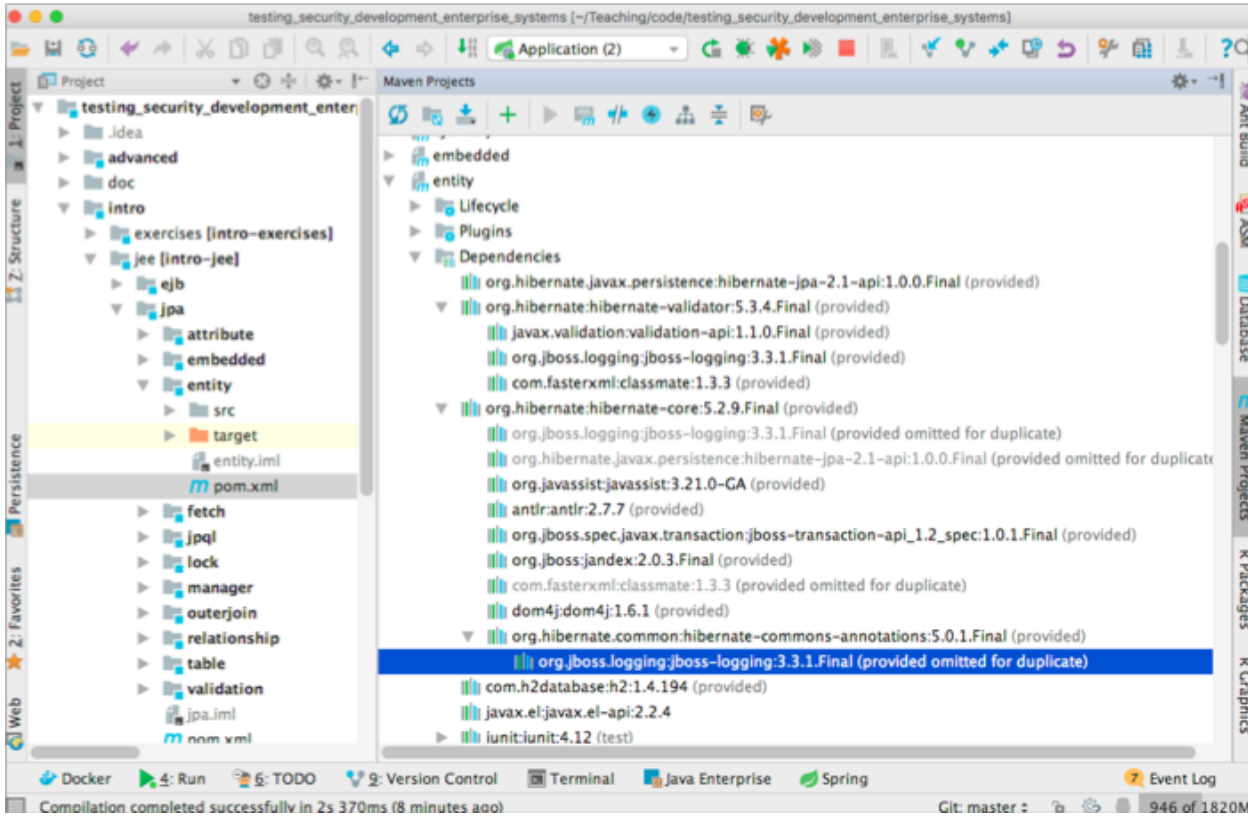
- You might use a library in many, many modules
- To avoid copy&paste and maintain **<version>/<scope>** everywhere, use **<dependencyManagement>** in a shared ancestor *pom.xml*, eg the root one
- All submodules will inherit the **<version>/<scope>** values

<dependencyManagement> Confusion

- You add dependencies with **<dependency>** tag inside a **<dependencies>** list
  - those will be used in the application
- Configuration declarations are inside a **<dependencyManagement>**, which itself will have a **<dependencies>** with a list of **<dependency>**
  - those are NOT used in the application… it is just configurations (eg **<version>** and **<scope>** for sub-modules)

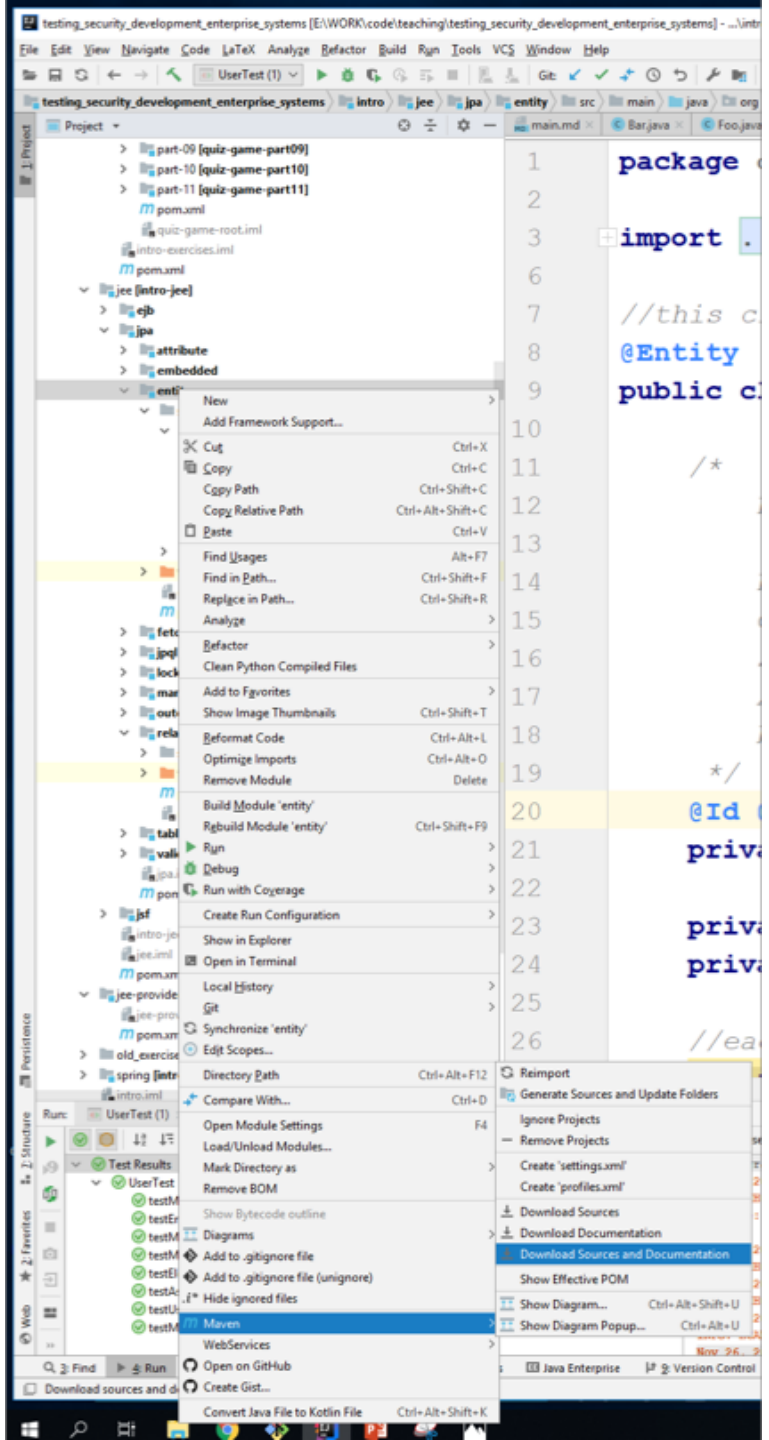
Transitive Dependencies

- 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*



Debugging

- To run your apps, you just need the bytecode in the JAR of the dependencies
- For debugging, you might also want their *JavaDocs* and *source-code*
- Note: you can look at source code with right-click "**Go To -> Implementation(s)**"
- You need to tell IDE to download sources for the *Maven* dependencies



Java Packages

- Classes can be grouped in different folders (with meaningful names), to better organize them
- Java uses *packages* to group related classes
- A class is identified by its *name* and its *package*
  - eg, *java.lang.String* is the class named *String* in package *java.lang*
- In Java, a package must match the folder structure
- Example: *org.tsdes.intro.jee.jpa.entity.User01* must be in the file *src/main/java/org/tsdes/intro/jee/jpa/entity/User01.java*
  - each "." represents a new nested folder

Modules vs. Packages

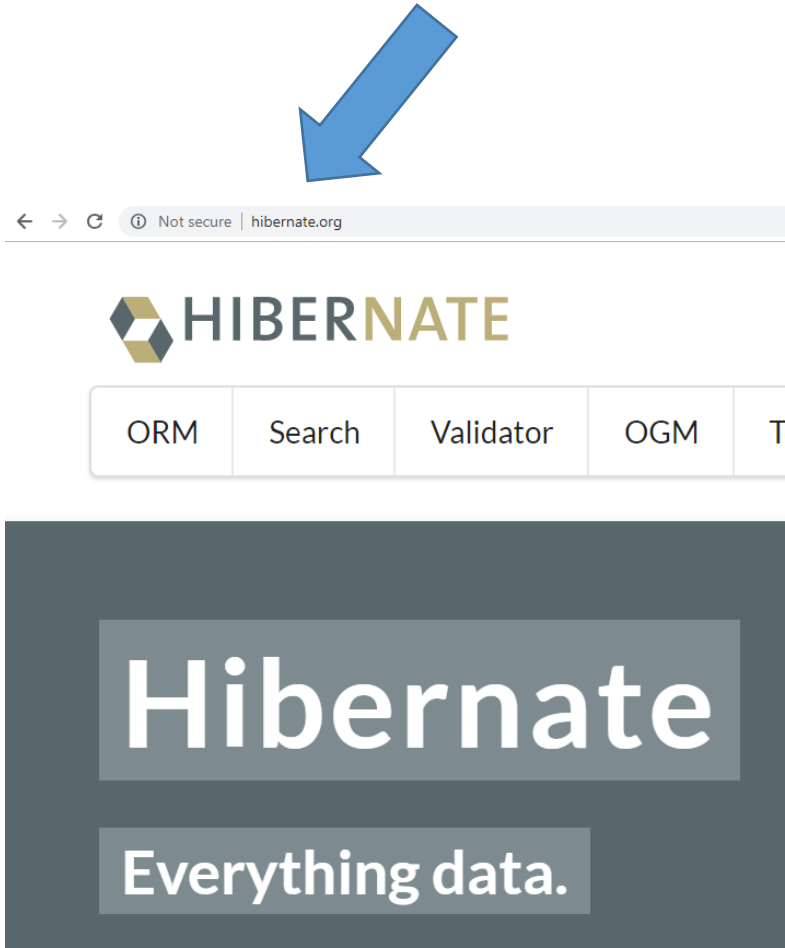
- *Packages* are critical to organize large projects
- *Maven Modules* play the same role, in which a too large project can be split in different parts
  - it is just a step up compared to packages
- Modules can be built independently, and provide their own different outputs
  - ie. packaged JAR/WAR files

Naming Packages

- Technically, can name a package as you want, but there are *conventions*
- The package prefix is usually a *reversed hostname*, like for example *org.hibernate*
  - eg, start with *com.*, *no.*, *org.*, etc followed by a name describing your project
- After the prefix, just use names meaningful for grouping your classes
  - eg *org.tsdes.intro.jee.jpa.entity.User01,* where *org.tsdes* is the prefix

Publishing a Library: Naming

- Common to use as *<groupId>* the prefix of your packages
- For security (eg, phishing attacks) Maven Central might require you to own the domain (eg hostname on DNS servers) if you want to publish with a given *<groupId>*
  - eg, *http://hibernate.org/*




```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
</dependency>
```

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.jpa>1.0.0.Final</version.hibernate.jpa>
   <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>
```

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 JEE 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

Wrapper Objects

- In the *@Entity* classes we will not use *primitive* types like **int** and **long**
- We use wrapped objects from *java.lang.** package
  - eg, **java.lang.Integer** instead of **int**
- Reason: need to handle the case of database columns being **NULL**
  - which would not be possible with primitive types
- Note: Java can do automated (un)boxing of primitive types
  - eg, **Integer x = 0;**

Hibernate as a Library

- We will start using *Hibernate* as a library in a Java SE context
- We will need to start/commit/close *transactions* manually
- Later, we will use Hibernate/JPA in a JEE container (*WildFly*), where transactions will be handled *automatically*
- Using an *embedded* database like H2
  - typically used for testing
  - we will see *Postgres* later in the course

Git Repository Modules

- *NOTE: most of the explanations will be directly in the code as comments, and not here in the slides*
- **intro/jee-provided-dependencies**
- **intro/jee/jpa/entity**
- **intro/jee/jpa/table**
- **intro/jee/jpa/embedded**
- **intro/jee/jpa/attribute**
- Exercises for Lesson 01 (see documentation)