

Maven

# Maven

- Maven je nástroj pro automatizaci buildování aplikací určený zejména pro programovací jazyk Java. Plní podobnou roli jako Ant, ale je postaven na jiných konceptech.
  - <http://maven.apache.org/>
- **Maven je možné používat:**
  - Bud' **uvnitř vývojového prostředí (IDE)** jako je Eclipse, NetBeans nebo IntelliJ Idea (v současnosti každé z výše uvedených vývojových prostředí má podporu pro práci s Mavenem out-of-the-box – díky embeddovanému Mavenu).
  - Nebo **pomocí příkazové řádky** – k tomu je nutné si Maven stáhnout, rozbalit například do C:/Program Files/Apache Maven a přidat do Path ve Windows cestu do podadresáře bin, kde se nachází mvn.bat. Maven je čistě Java aplikace a vyžaduje také nastavenou proměnnou JAVA\_HOME, která musí ukazovat do adresáře, kde je nainstalované JDK.

# pom.xml (Project Object Model)

- Maven používá pro popis struktury tvořeného projektu, jeho závislostí (dependencies) na jiných knihovnách a celkově jeho buildování XML soubor pom.xml, který se nachází v domovském adresáři projektu.
- Nejjednodušší pom.xml soubor pro JAR knihovnu:

```
<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>cz.jiripinkas.example</groupId>
```

```
<artifactId>example-jar</artifactId>
```

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

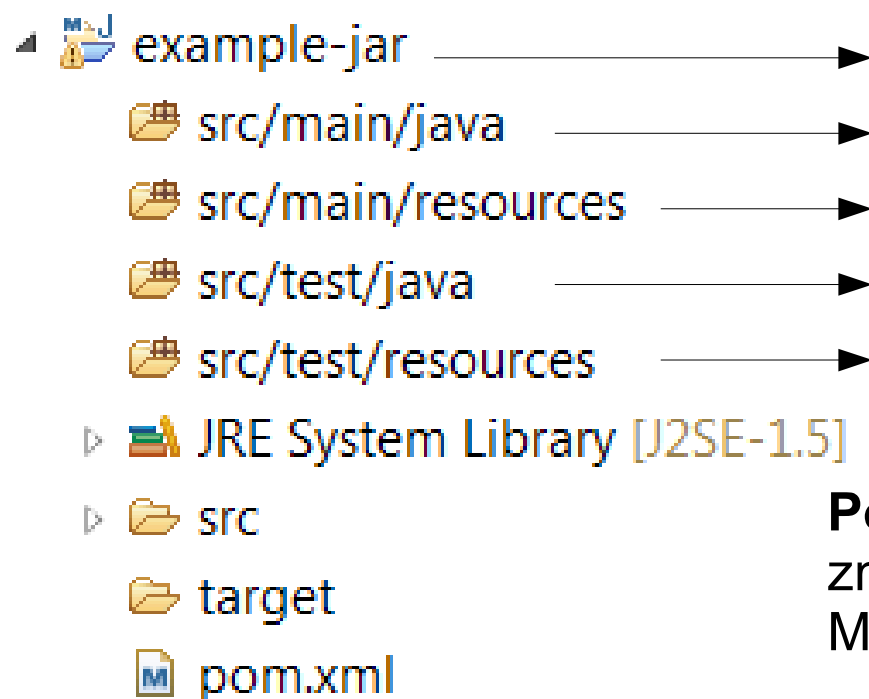
```
</project>
```

Název skupiny (často to samé  
jako název balíčku)

Název artefaktu (název projektu)  
- každý projekt Mavenu  
je artefakt (artifact)

# Struktura projektu I.

- Každý Maven projekt má standardně následující strukturu:



adresář	obsah adresáře
kořenový adresář aplikace	soubor pom.xml a ostatní adresáře
src/main/java	.java soubory
src/main/resources	konfigurační soubory
src/test/java	třídy testů
src/test/resources	konfigurační soubory testů

**Poznámka:** je možné tuto výchozí strukturu změnit, ale nedoporučoval bych to), spousta Maven pluginů tuto strukturu předpokládá.

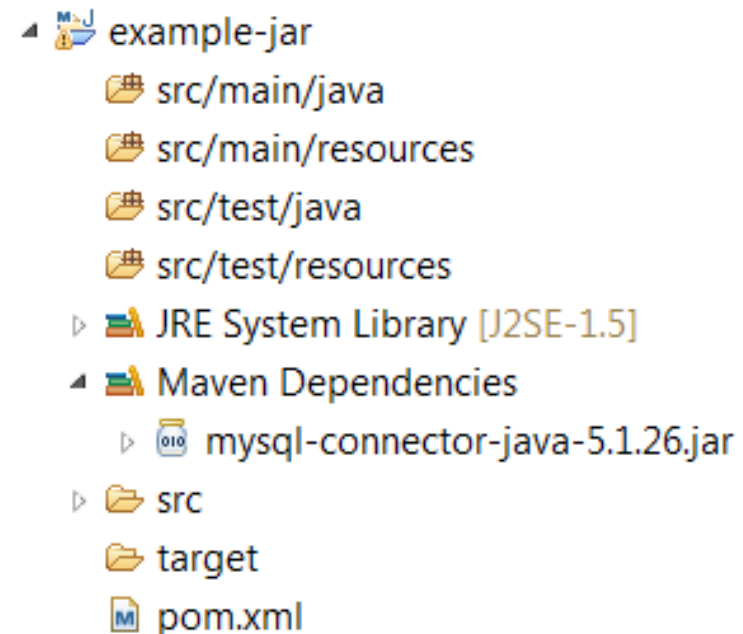
# Struktura projektu II.

- Ve Vašem projektu nemusí být všechny standardní adresáře uvedené na předcházejícím snímku. V praxi když vytvoříte projekt na základě nějakého Maven archetype, pak tam často některý z adresářů chybí.
- Například po vytvoření projektu z archetype `org.codehaus.mojo.archetypes:webapp-javaee6` chybí adresář `src/main/resources`.
- Řešení:
  - Přejděte do Properties projektu a vyberte Java Build Path.
    - Pokud je tu adresář `src/main/resources` definovaný, ale je u něj, že chybí na disku, pak otevřete Total Commander a vytvořte ho. Potom v Eclipse klikněte na projekt a zmáčkněte F5 pro refresh projektu.
    - Pokud zde adresář `src/main/resources` není, pak klikněte na Add Folder, vytvořte a přidejte ho.

# Dependencies I.

- Maven usnadňuje správu závislostí Java knihoven (JAR) souborů. Stačí do pom.xml přidat:

```
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.26</version>
  </dependency>
</dependencies>
```



Kde tento kus XML vzít? Velice často na domovských stránkách příslušného projektu nebo na těchto stránkách: <http://www.mvnrepository.com/>  
Další vyhledávací stroje:  
<http://stackoverflow.com/questions/3430423/recommendable-maven-repository-search-engines>  
Oficiální vyhledávací stroj: <http://search.maven.org/>  
Můj vyhledávací stroj: <http://javalibs.com>

# Version

- Do tagu `<version>` můžete vložit:
  - Konkrétní verzi (doporučené, nejčastěji používané)
  - Hranaté závorky („inclusive“)
  - Kulaté závorky („exclusive“)
    - <http://stackoverflow.com/questions/30571/how-do-i-tell-maven-to-use-the-latest-version-of-a-dependency>

Poznámka: Při tvorbě vlastního projektu je opravdu hodně doporučeno používat standardní verzování:

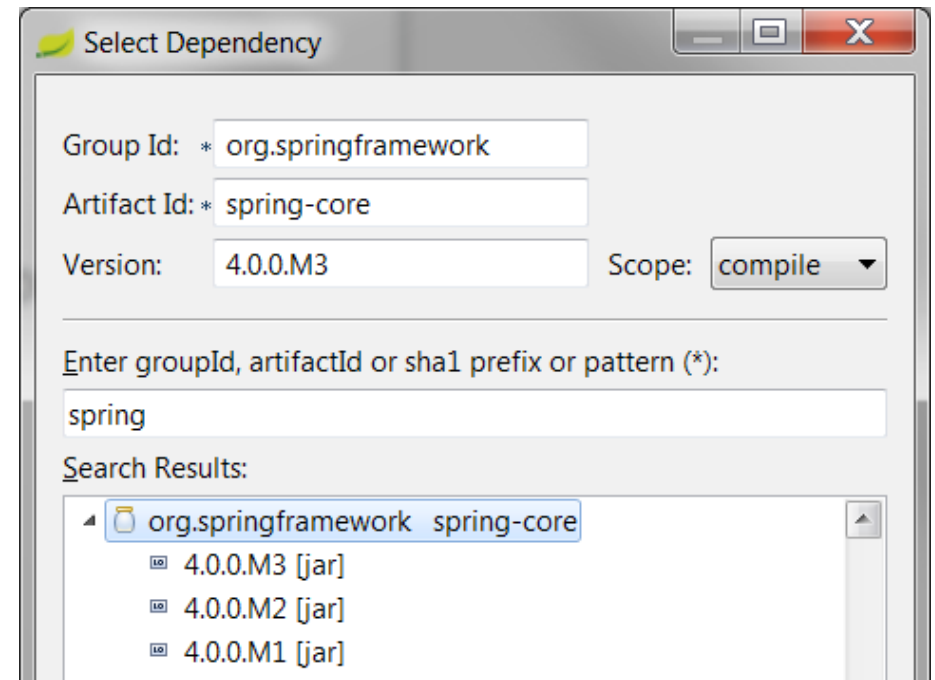
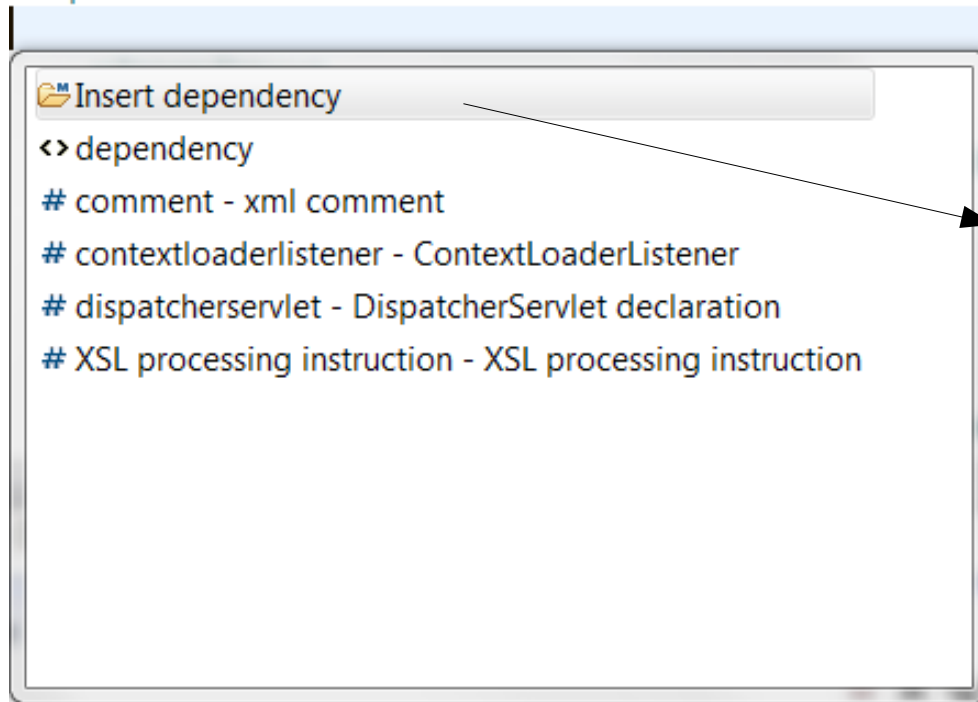
`major.minor.incremental`

A v průběhu projektu ho neměnit.  
Jinak se v Nexusu a jinde špatně řadí.  
(na to také pozor když chcete najít nejnovější artifact).

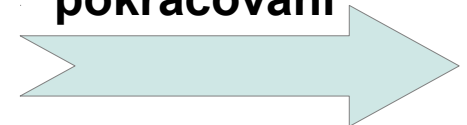
# Maven & Eclipse I.

- V Eclipse v pom.xml funguje když kliknete pravým tlačítkem, dáte Insert dependency a můžete vybírat dependency přímo v IDE bez nutnosti navštívit externí vyhledávací stroj:

<dependencies>



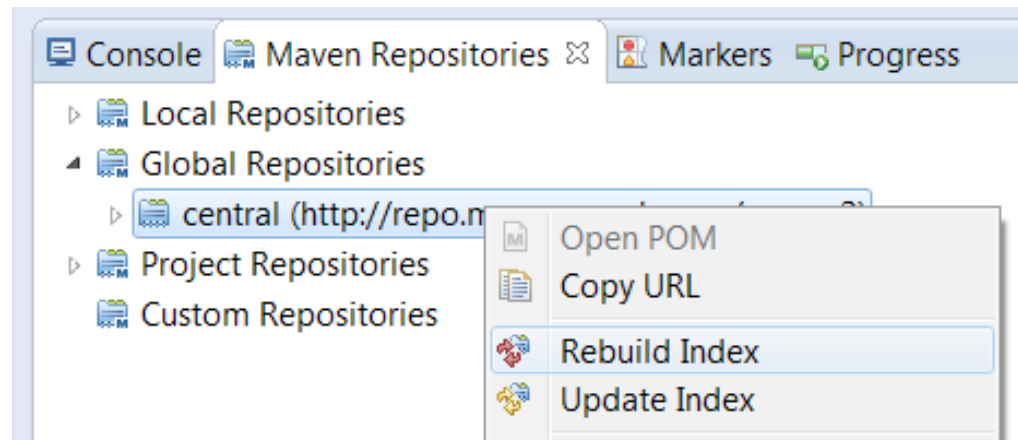
**pokračování**





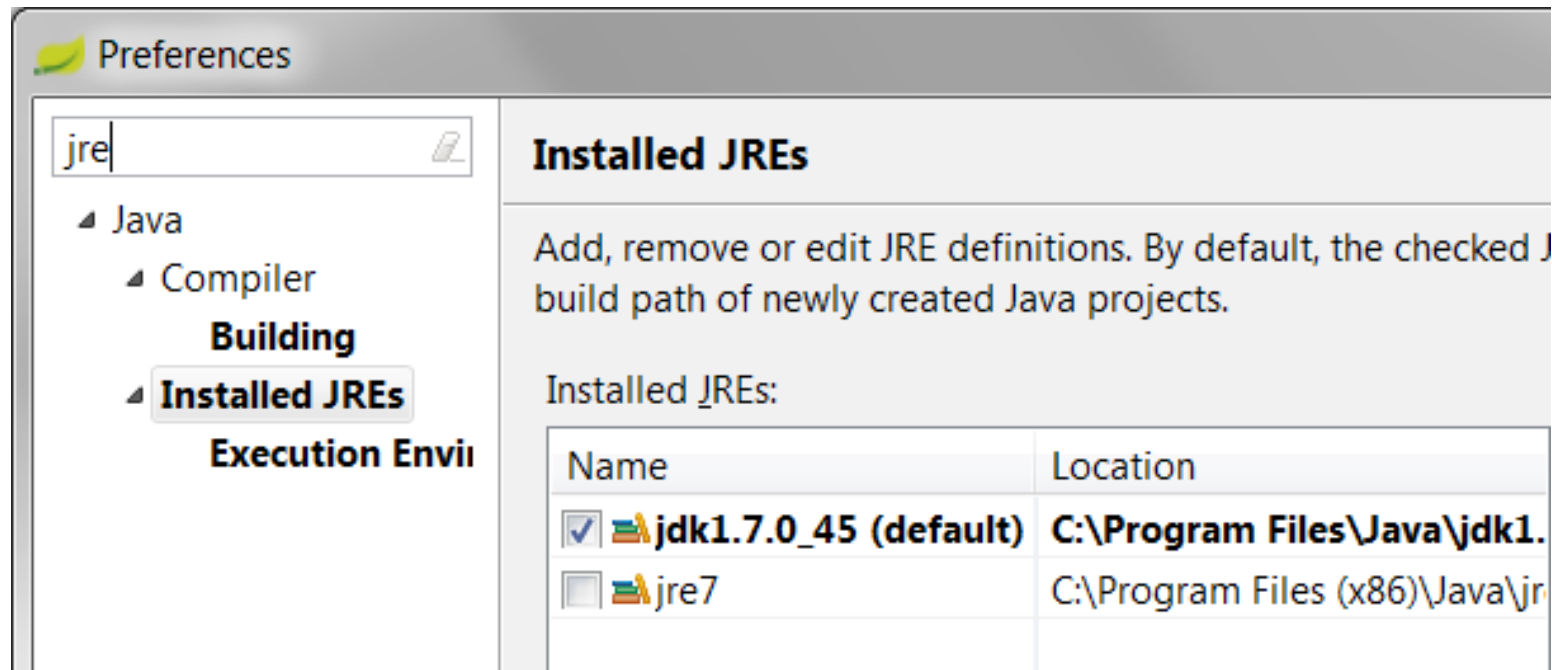
# Maven & Eclipse II.

- K tomu aby funkcionality na předcházejícím snímku fungovala, je nutné otevřít Window → Show View → Other a najít zde view s názvem Maven Repositories.
- Poté musíte v tomto view kliknout na repozitáře ze kterých chcete získávat artefakty a vybrat „Rebuild Index“. Tato operace bude chvíli trvat, Eclipse si přitom stáhne informace o tom, jaké artefakty jsou v příslušných repozitářích:



# Maven & Eclipse III.

- Maven závisí na JDK, tudíž musíte mít v Eclipse ve Window → Preferences → Installed JREs nastavené jako výchozí JDK:



# Dependencies II.

- Pokud není soubor v lokálním repozitáři, pak Maven JAR soubory stáhne ze vzdáleného repozitáře do lokálního repozitáře a poté je propojí s Maven projektem.
- Lokální repozitář je standardně v `[home]/.m2`
- V řadě společností se setkáte s vnitrofiremními repozitáři (viz. dále).
- Out-of-the-box je podporován Central repozitář Mavenu, kde je drtivá většina open source projektů.



Maven Central repozitář: <http://repo1.maven.org/maven2>

# Nepoužívané dependency

- Postupně se v každém projektu nejspíš dostanete do situace, kdy budete mít v `pom.xml` dependency, které ve skutečnosti nebudete používat. Jak je zjistit?
- Vodítko k nepoužívaným dependencies je spustit:  

```
mvn dependency:analyze
```
- Proveďte kompilaci aplikace a zjistí, které třídy se v kódu aplikace nepoužívají. Bohužel tato operace není stoprocentní, například pokud definujete JDBC DataSource pomocí Spring XML konfiguračního souboru, pak se nezjistí, že se tato knihovna bude v aplikaci používat ... takže to berte s rezervou.

# Novější dependency

- Jak zjistit že existují novější dependency?

```
mvn versions:display-dependency-updates
```

- Obdobným způsobem zjistíte že existují novější pluginy:

```
mvn versions:display-plugin-updates
```

- Co ale nezjistí je, když tvůrce přestane dependency / plugin vyvíjet a vytvoří nový artifact, ve kterém pokračuje. Příklad: `hsqldb:hsqldb` vs. `org.hsqldb:hsqldb`:

- <http://javalibs.com/artifact/hsqldb/hsqldb>
- <http://javalibs.com/artifact/org.hsqldb/hsqldb>

# Maven Force Update

- Někdy se můžete dostat do situace, kdy Vám Maven začne stahovat dependency, internetové připojení přestane fungovat a pak Vám Maven vrací chybu:
  - ... was cached in the local repository, resolution will not be reattempted until the update interval of central has elapsed or updates are forced.
- Jak tento problém vyřešit? V příkazové řádce:

```
mvn clean install -U
```
- V Eclipse: Na projektu zmáčknete ALT + F5 a vyberte:
  - Force Update of Snapshots/Releases

# Cannot be read or is not a valid ZIP file

- Někdy můžete narazit při buildu na tuto chybu:
  - Archive for required library: xxx cannot be read or is not a valid ZIP file.
  - Příčinou problémů bývá, že Maven špatně stáhnul JAR soubor. S tímto problémem se občas setkávám ve spojení s Eclipse.
- Řešení:
  - Většinou musíte zavřít Eclipse, smazat adresář ve kterém se chybný JAR soubor nachází a znovu Eclipse nastartovat. Maven si automaticky znovu stáhne požadovaný JAR soubor a vše funguje.

# Transitive dependencies I.

- Závislosti také mohou být tranzitivní. Při logování se často používá kombinace tří knihoven: log4j, slf4j a slf4j-log4j. Můžete do pom.xml přidat všechny tři dependency, nebo můžete využít tranzitivních závislostí a přidat pouze slf4j-log4j:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.5</version>
</dependency>
```

V Eclipse můžete vidět tranzitivní závislosti  
v pom.xml souboru na záložce  
Dependency Hierarchy:

## 📦 Maven Dependencies

- ▶ 📦 slf4j-log4j12-1.7.5.jar
- ▶ 📦 slf4j-api-1.7.5.jar
- ▶ 📦 log4j-1.2.17.jar

## Dependency Hierarchy

- ▶ 📦 slf4j-log4j12 : 1.7.5 [compile]
  - 📦 slf4j-api : 1.7.5 [compile]
  - 📦 log4j : 1.2.17 [compile]

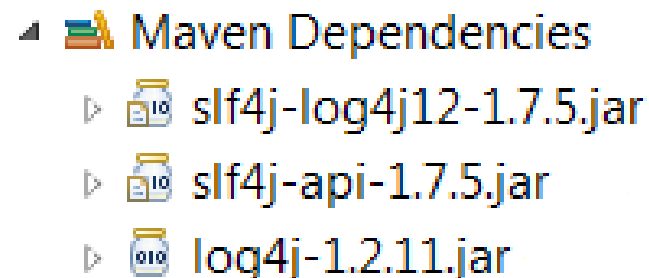


# Transitive dependencies II.

- Když chcete „přebít“ tranzitivní závislosti Vaší vybranou verzí knihovny (ať novější nebo starší), stačí ji přidat do dependencies:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.5</version>
</dependency>

<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.11</version>
</dependency>
```



## Dependency Hierarchy

- slf4j-log4j12 : 1.7.5 [compile]
  - slf4j-api : 1.7.5 [compile]
  - log4j : 1.2.17 (omitted for conflict with 1.2.11) [compile]
  - log4j : 1.2.11 [compile]

Získání seznamu dependencies  
z příkazové řádky:

mvn dependency:resolve

Získání stromu dependencies:

mvn dependency:tree

# Transitive dependencies III.

- Někdy můžete chtít tranzitivní závislost kompletně vyloučit:

```
<dependency>

  <groupId>org.springframework</groupId>

  <artifactId>spring-context</artifactId>

  <version>3.2.4.RELEASE</version>

  <exclusions>

    <!-- Exclude Commons Logging in favor of slf4j -->

    <exclusion>

      <groupId>commons-logging</groupId>

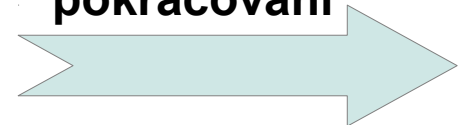
      <artifactId>commons-logging</artifactId>

    </exclusion>

  </exclusions>

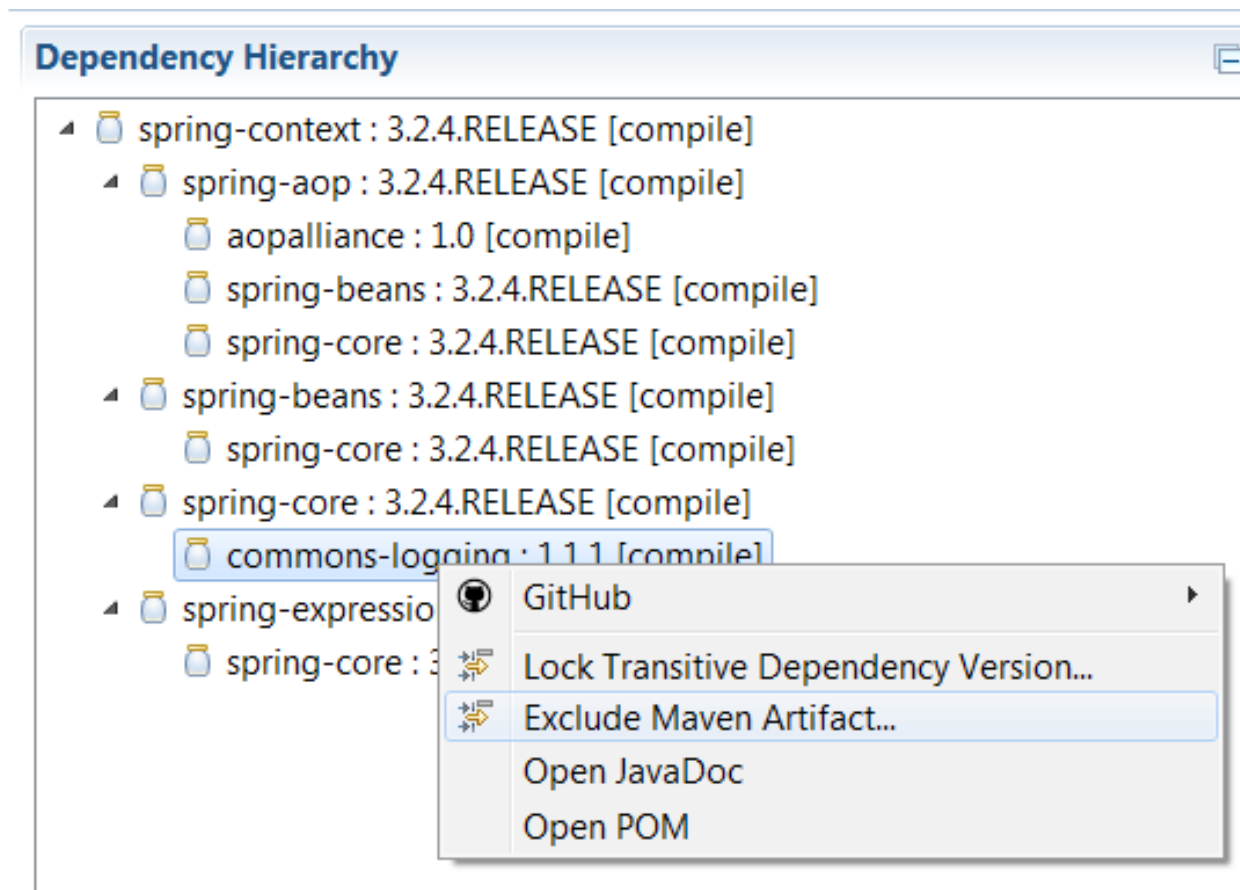
</dependency>
```

**pokračování**



# Transitive dependencies IV.

- Jak na to jednoduše v Eclipse? V pom.xml v záložce Dependency Hierarchy klikněte na závislost pravým tlačítkem:



# BOM (Bill Of Materials)

- Některé frameworky obsahují bom dependency, která slouží k zajištění stejné verze jednotlivých knihoven:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-framework-bom</artifactId>
      <version>4.1.0.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

# Properties I.

- Protože často chcete od stejné verze nějakého projektu více knihoven, lehce se můžete dostat do situace, kdy se Vám bude v `pom.xml` duplikovat text:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>3.2.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>3.2.4.RELEASE</version>
</dependency>
```

**pokračování**



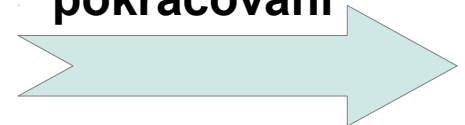
# Properties II.

- Řešením je použití properties:

```
<properties>
  <spring.version>3.2.4.RELEASE</spring.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${spring.version}</version>
  </dependency>
</dependencies>
```



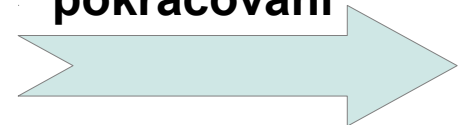
**pokračování**



# Properties III.

- Existují také předdefinované properties:
  - `${basedir}` = adresář kde je `pom.xml`
  - `${version}` = verze projektu
- Všechny elementy v `pom.xml` souboru můžete zpřístupnit pomocí prefixu `project`.
  - Příklady:
    - `${project.artifactId}` = artifact ID (název projektu)
    - `${project.build.directory}` = cesta do adresáře `target`
- Obdobně všechny elementy v `settings.xml` souboru (bude probrán později) můžete zpřístupnit pomocí prefixu `settings`.
  - `${settings.localRepository}` = kde je lokální Maven repozitář na disku

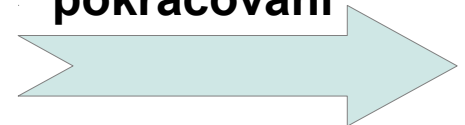
pokračování



# Properties IV.

- Stejným způsobem se dají používat proměnné prostředí, Java systémové proměnné a další.
- Seznam všech proměnných je zde:
  - <http://docs.codehaus.org/display/MAVENUSER/MavenPropertiesGuide>

**pokračování**





Pozor! Předbívám :-)  
Je to tu jen pro doplnění!

# Properties V.

- **TIP:** Pokud chcete zobrazit seznam všech proměnných, přidejte toto do pom.xml:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-antrun-plugin</artifactId>
      <version>1.6</version>
      <executions>
        <execution>
          <phase>install</phase>
          <configuration>
            <target> <echoproperties /> </target>
          </configuration>
          <goals> <goal>run</goal> </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

A potom zavolejte `mvn install`

# Webová aplikace

- Takto vypadá pom.xml soubor webové aplikace:

```
<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>cz.jiripinkas.example</groupId>
```

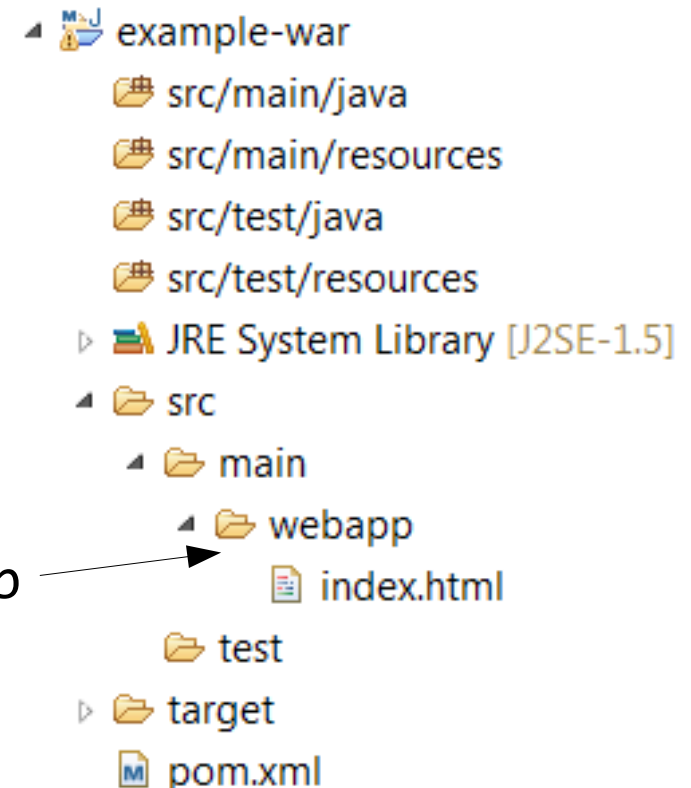
```
<artifactId>example-jar</artifactId>
```

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

```
<packaging>war</packaging>
```

```
</project>
```

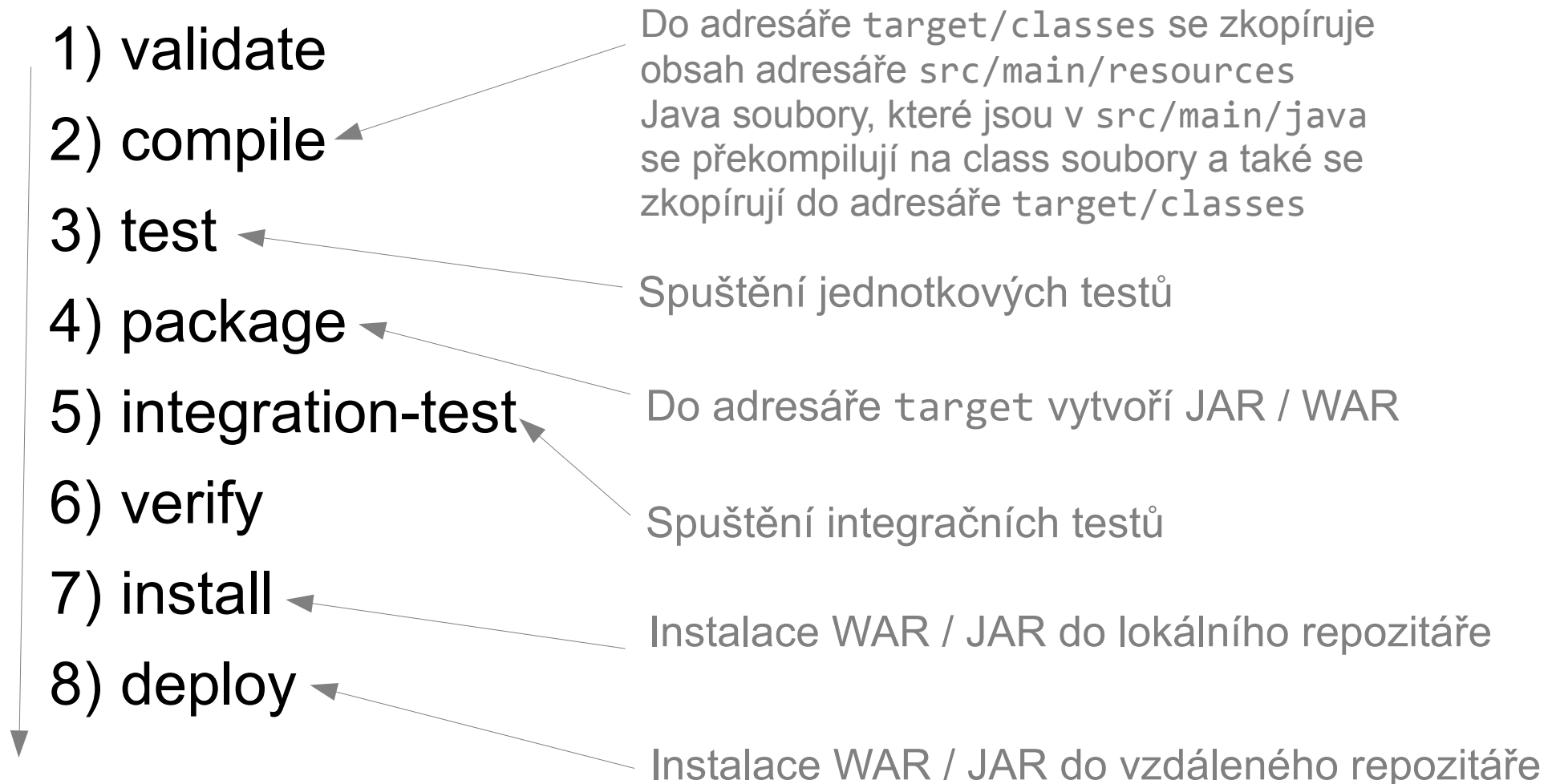
- Veřejné webové stránky jsou standardně v adresáři src/main/webapp



Když se například spustí  
`mvn package` (4. fáze),  
pak se vyvolají fáze 1 – 4!

# Build fáze I.

- Každý Maven projekt má následující základní fáze buildování:



směr provádění jednotlivých fází

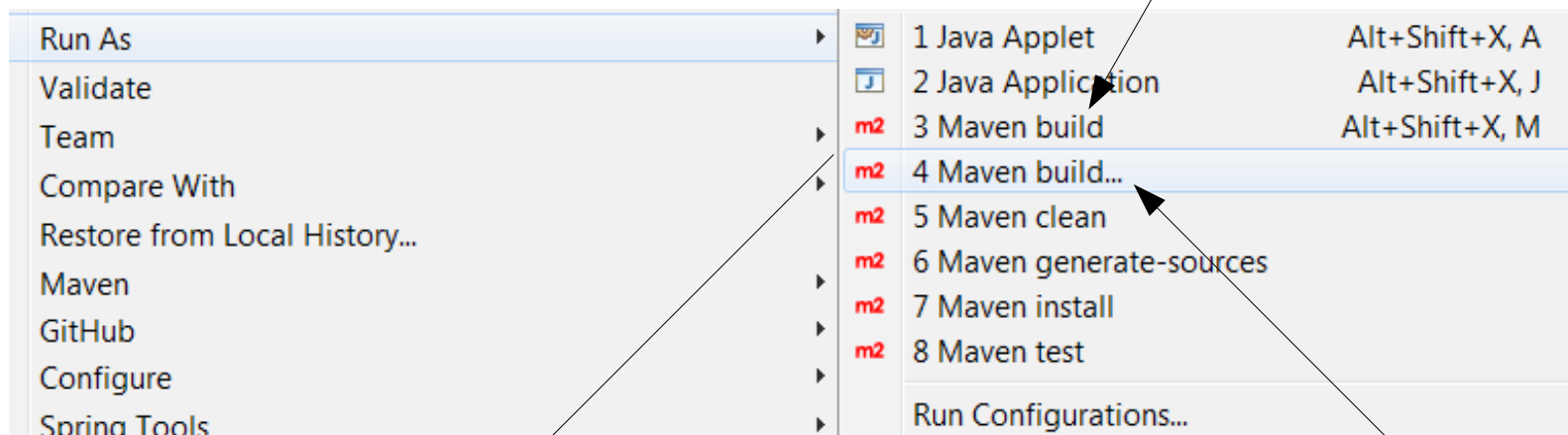
# Build fáze II.

- Build fází je celá řada, víc než jsem uvedl. K těm nejdůležitějším se postupně dostaneme. Celý seznam je zde:
  - [http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle\\_Reference](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference)

# Build fáze III.

- Spouštění fází:
  - Z příkazové řádky: `mvn package`
  - V Eclipse:

Pokud není vytvořena žádná spouštěcí konfigurace, pak ji vytvoří. Pokud je vytvořena jenom jedna, pak ji spustí. Pokud jich je víc, pak nabídne dialog pro výběr spouštěcí konfigurace



Goals:

Vytvoří novou spouštěcí konfiguraci

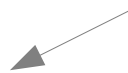
# Build fáze IV.

při spouštění  
Mavenu z konzole



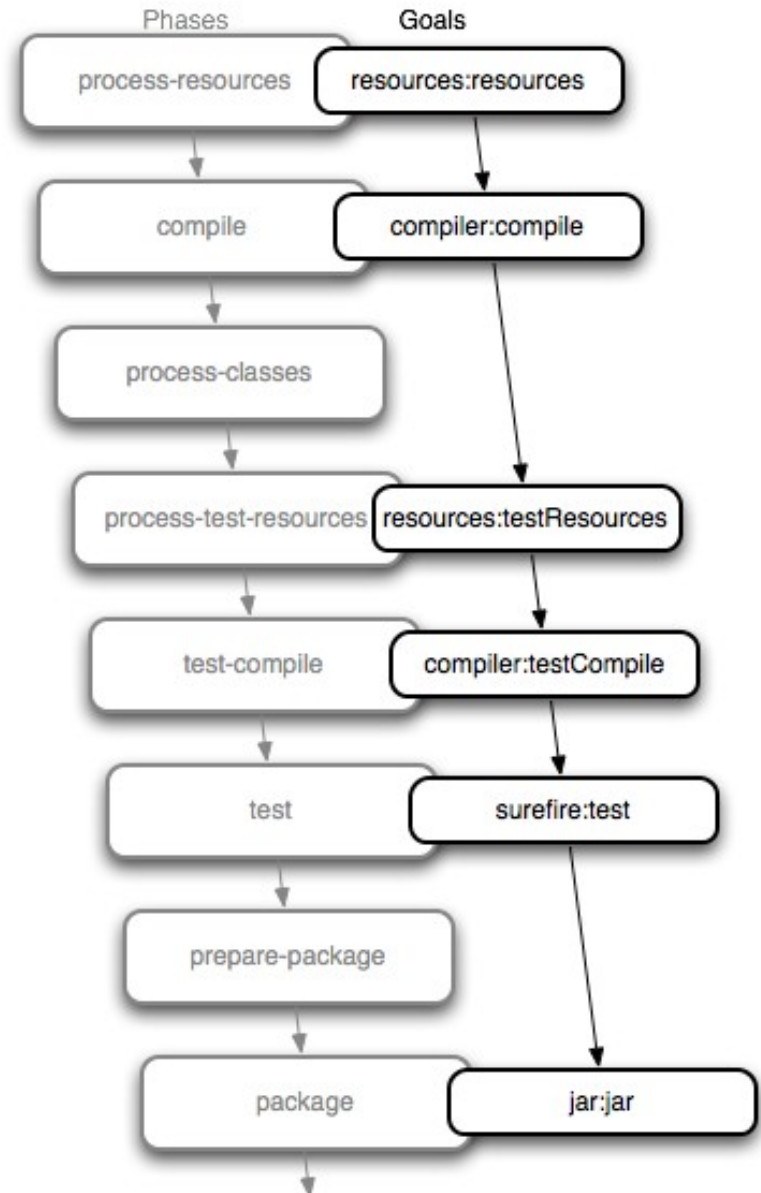
- Build fáze je možné volat následujícím způsobem:
  - `mvn [název build fáze]`
- Build fáze, která smaže obsah adresáře target:
  - `mvn clean`
- Jednotlivé build příkazy je možné řetězit:
  - `mvn clean package`
- Maven je rozšiřitelný pomocí pluginů, požadované pluginy se definují v `pom.xml` souboru.
- Plugin se spustí zavoláním:
  - `mvn [název pluginu]:[goal]`

co bude  
plugin dělat



# Build fáze vs. goal

- Často se setkáte s pojmem „goal“. Navíc se občas nesprávně slučuje goal a build fáze, ale je mezi nimi důležitý rozdíl. Build fáze jsou jednoznačně dané. Goaly jsou akce, které můžete volat zvlášť, nebo dokonce mohou být na build fáze „navěšené“.
- Standardně je to tak i nastavené, na jednotlivých fázích jsou „navěšené“ různé goaly podle toho, o jaký typ projektu se jedná.



# „Navěšení“ goalu na build fázi

```
<plugin>
```

```
  <artifactId>maven-antrun-plugin</artifactId>
```

```
  <version>1.6</version>
```

```
  <executions>
```

```
    <execution>
```

```
      <phase>install</phase>
```

Na jakou fázi se navěší goal

```
      <configuration>
```

```
        <target> <echoproperties /> </target>
```

```
      </configuration>
```

```
      <goals>
```

```
        <goal>run</goal>
```

Jaký goal tohoto pluginu  
bude navěšen na výše uvedenou fázi

```
      </goals>
```

```
    </execution>
```

```
  </executions>
```

```
</plugin>
```



# Super POM, Effective POM

- Každý `pom.xml` soubor je potomkem výchozího, který je v Maven instalaci. Tento výchozí `pom.xml` soubor se nazývá Super POM.
- V Super POM je mapování goal na build fáze, definování základních pluginů, výchozí adresářová struktura a URL Maven Central repozitáře.
- Veškeré nastavení v Super POM je možné ve Vašem POM změnit.
- Spojení Super POM a Vašeho POM se nazývá Effective POM, který se ve finále použije.

# Maven Plugins

- Existuje celá řada pluginů, které je možné použít ve Vašich Maven projektech.
- Jestli znáte Ant, pak pluginy jsou prakticky jako předprogramované targety (mimochodem pomocí antrun pluginu můžete volat Ant skripty).
- Pluginy můžete volat buď samostatně, nebo je můžete zapojit na nějakou Maven fázi (viz. příklad na zobrazení všech properties při spuštění install fáze, který byl uveden před několika snímky).
- Základní Maven pluginy naleznete zde:
  - <http://maven.apache.org/plugins/index.html>
- Další hromada Maven pluginů je zde:
  - <http://mojo.codehaus.org/plugins.html>
- Některé projekty mají své vlastní pluginy, které naleznete pouze na web. stránkách příslušného projektu:
  - <https://maven-glassfish-plugin.java.net/>

# Maven Profiles I.

- Maven aplikace je možné lehce buildovat kdekoli kde běží JDK. V některých situacích ale není možné zajistit 100% přenositelnost buildu – často je vývojové, testovací a produkční prostředí malinko jiné a z toho důvodu vyžaduje například použití jiných cest ke konfiguračním souborům, jiné závislosti atd.
- Maven obsahuje profily, které umožňují změnit výsledný build (JAR / WAR soubor).
- Dále se podíváme na typické použití profilů, všechny použití naleznete v dokumentaci:
  - <http://maven.apache.org/guides/introduction/introduction-to-profiles.html>

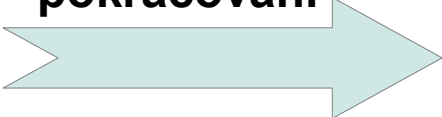
# Maven Profiles II.

- Při vývoji můžete chtít používat testovací databázi (např. HSQLDB), zatímco na produkci chcete používat produkční databázi (např. MySQL). Přidejte do pom.xml:

```
<profiles>
  <profile>
    <activation> <activeByDefault>true</activeByDefault> </activation>
    <id>dev</id>
    <dependencies>
      <dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>2.3.0</version>
      </dependency>
    </dependencies>
  </profile>
```

Název profilu

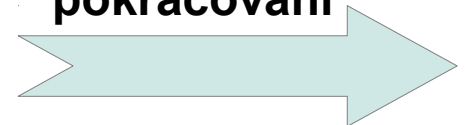
Výchozí profil (pokud neřekneme jinak)

**pokračování** 

# Maven Profiles III.

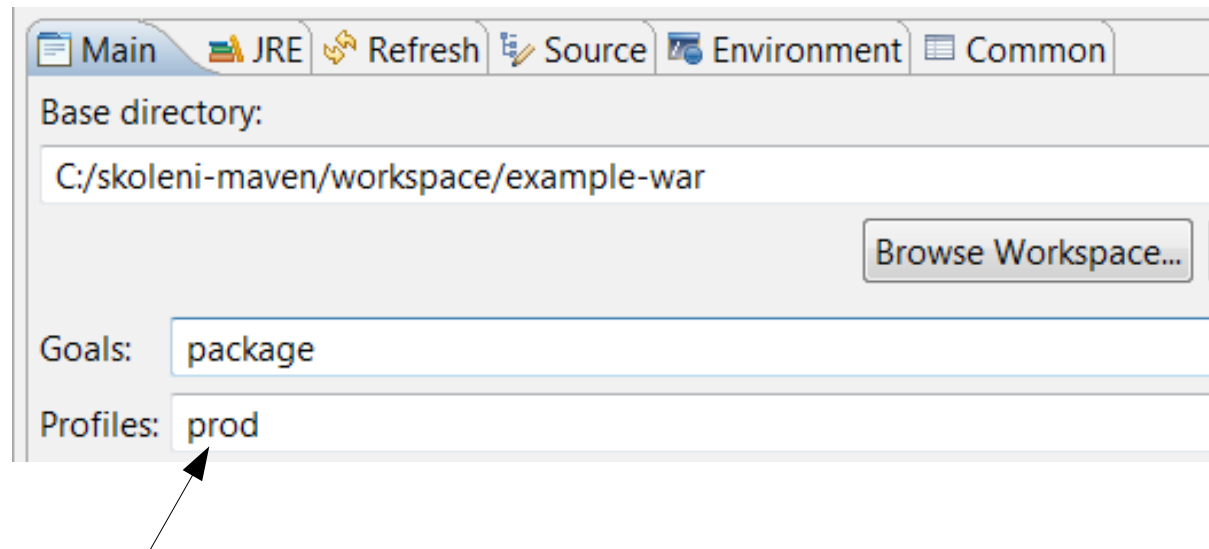
```
<profile>
  <id>prod</id>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.26</version>
    </dependency>
  </dependencies>
</profile>
</profiles>
```

**pokračování**



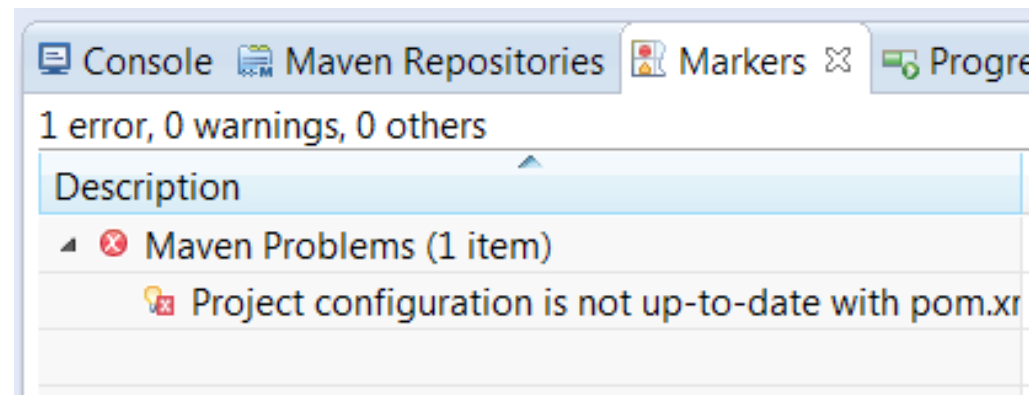
# Maven Profiles - spuštění

- Změnit profil můžete:
  - V příkazové řádce: `mvn package -P prod`
  - V Eclipse při tvorbě spouštěcí konfigurace:

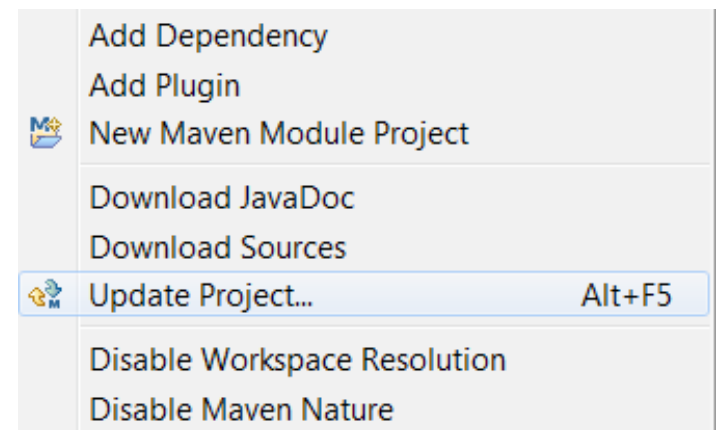


# Eclipse tip: Update Maven Project

- Při velké změně `pom.xml` (jako je například změna verze Javy pod kterou se bude projekt kompilovat) se zobrazí tato chyba:



- Buď na ni klikněte a dejte CTRL + 1 a potvrďte Quick Fix, nebo použijte kdekoli klávesovou zkratku ALT + F5 nebo klikněte na projekt a pravým tlačítkem a vyberte Maven → Update Project:



# Scope I.

- Co když máte webovou aplikaci a na produkčním serveru máte nějaké dependency ve sdílených knihovnách? Pak je nechcete vkládat do pom.xml souboru. Také knihovna JUnit má význam pouze při testování aplikace, ale při běhu je zbytečná.
- K těmto situacím slouží tag `scope`, který je možné přidat dovnitř tagu `dependency`:

```
<dependency>  
  <groupId>mysql</groupId>  
  <artifactId>mysql-connector-java</artifactId>  
  <version>5.1.26</version>  
  <scope>provided</scope>  
</dependency>
```



# Scope II.

- K dispozici jsou tyto scopes:
  - `compile` – výchozí scope, který se použije, když není žádný scope specifikován. Dependency je k dispozici ve všech classpath projektu a propagují se do výsledného JAR / WAR souboru.
  - `provided` – Dependency je k dispozici pouze při kompilaci a vykonávání testů, nepropaguje se do výsledného JAR / WAR souboru (u webových aplikací typicky servlet, jsp API, sdílené knihovny atd.).
  - `test` – Podobné jako `provided`, ale říkáte, že se tato dependency používá v testech (typicky JUnit knihovny).
  - a další:
    - <http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>

# Servlet & JSP Java EE 6 dependencies

- Při tvorbě webových aplikací obvykle chcete používat třídy jako `HttpServletRequest` apod. a k tomu musíte mít v classpath při vývoji Servlet & JSP dependency:

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.0.1</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>javax.servlet.jsp-api</artifactId>
  <version>2.2.1</version>
  <scope>provided</scope>
</dependency>
```

# Optional Dependencies

- Některé artifacty mají optional dependency:
  - <http://repo1.maven.org/maven2/cz/jiripinkas/jsitemapgenerator/2.2/jsitemapgenerator-2.2.pom>
  - K čemu je to dobré? Optional dependency je k dispozici při kompilaci aplikace, ale do projektu který používá knihovnu obsahující optional dependency se neuloží jako tranzitivní dependency (čili když ji chceme použít, musíme ji do pom.xml explicitně přidat).

# Oracle JDBC driver I.

- Kvůli licenčním omezením není Oracle JDBC ovladač v Central repozitáři Mavenu. Ale ve spoustě aplikací chcete pracovat s Oracle databází :-) ... jak tento problém vyřešit?
  - Buď instalací JAR knihovny do Artifactory / Nexus repozitáře – toto je preferovaný způsob, zejména pokud aplikaci vyvíjí více než jeden vývojář (i pokud vyvíjíte aplikaci sami, tak je lepší tento způsob – tedy stáhnout si Artifactory / Nexus repozitář, lokálně ho spustit na svém počítači, nainstalovat JAR knihovnu do něj a používat ji z něj).
    - Jak na to je popsáno v přednášce o Artifactory.
  - Nebo pokud z nějakého důvodu první způsob používat nechcete, pak je možné nainstalovat JAR knihovnu do lokálního repozitáře Mavenu.
    - Jak na to je na dalším snímku.

# Oracle JDBC driver II.

- Dejte do Googlu oracle jdbc driver, stáhněte JAR soubor, v adresáři s tímto souborem otevřete cmd a v příkazové řádce spusťte:

```
mvn install:install-file -Dfile=NAZEV_SOUBORU.jar  
-DgroupId=com.oracle -DartifactId=ojdbc6  
-Dversion=11.2.0 -Dpackaging=jar
```

- Je úplně jedno jaké hodnoty nastavíte do groupId, artifactId a version.
- V pom.xml poté tuto dependency můžete použít:

```
<dependency>  
    <groupId>com.oracle</groupId>  
    <artifactId>ojdbc6</artifactId>  
    <version>11.2.0</version>  
</dependency>
```

# JUnit dependencies

- Při tvorbě a běhu JUnit testů musíte mít v classpath JUnit dependency:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
```

# JUnit testy

- Před fází package se volá fáze test, ve které se spouští JUnit testy (třídy v `src/test/java` s příponou `Test`).
- Jednotkové testy volá Maven Surefire Plugin. Když spustí JUnit test, pak do adresáře `target/surefire-reports` vygeneruje XML a text report.

# Ignorování JUnit testů

- Pokud nějaký z testů neprojde, pak se nepodaří vytvořit WAR / JAR soubor! Jak testy ignorovat?
- Jednorázově v příkazové řádce: `mvn package -DskipTests=true`
- Pomocí pluginu:

```
<plugin>  
  <groupId>org.apache.maven.plugins</groupId>  
  <artifactId>maven-surefire-plugin</artifactId>  
  <version>2.16</version>  
  <configuration>  
    <skipTests>true</skipTests>  
  </configuration>  
</plugin>
```

Obecně ale toto nedoporučuji ;-)



# Integrační testy

- Integrační testy (třídy v src/test/java s příponou IT) se spouští ve fázi integration-test nebo verify. K tomu je nutné přidat plugin:

```
<plugin>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>2.6</version>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

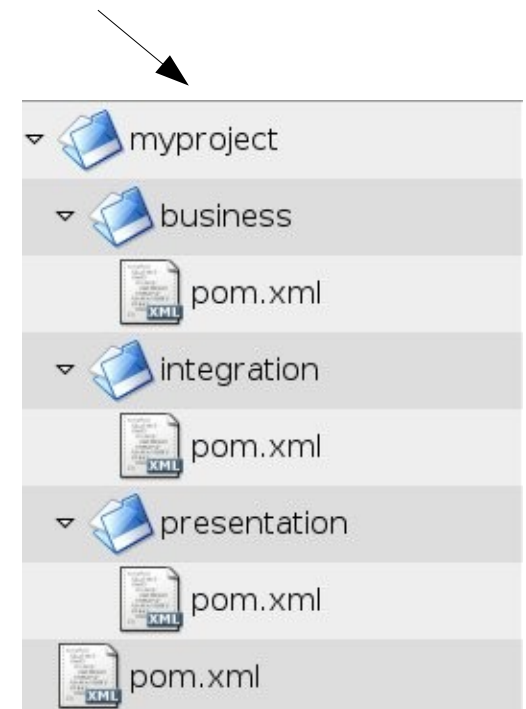
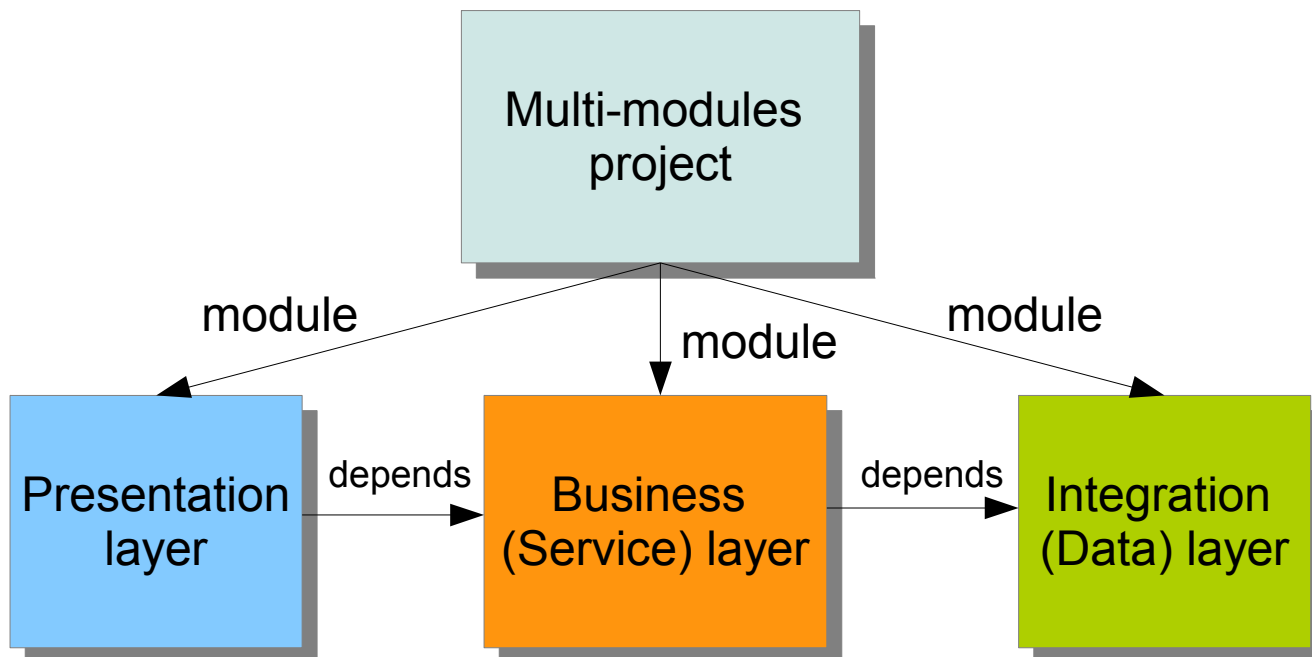
# Archetypes

- Maven má na svých serverech řadu tzv. archetypů, což jsou prakticky šablony projektů a jsou výborným způsobem jak rychle začít vyvíjet nějakou aplikaci.
- Všechny archetypy získáte pomocí příkazové řádky:
  - `mvn archetype:generate`
    - Zobrazí se všechny archetypy, můžete zúžit jejich výběr tím že zadáte část názvu archetype.

# Multi-modules projects I.

- Pokud se projekt skládá z více vrstev (např. webové aplikace jsou obvykle rozdělené do tří vrstev – tzv. třívrstvá architektura), pak je vhodné ho rozdělit do jednotlivých modulů, které odpovídají jednotlivým vrstvám:

Multi-modules project musí mít tuto adresářovou strukturu



# Parent project

- Hlavní (parent) projekt je typu pom a obsahuje seznam modulů (které jsou zároveň podadresáře):

```
<packaging>pom</packaging>

<modules>

  <module>eshop-repository</module>

  <module>eshop-service</module>

  <module>eshop-web</module>

</modules>
```

- Zároveň obsahuje dependencies a pluginy společné všem modulům.

# Module project I.

- Jednotlivé moduly neobsahují groupId, ale zato obsahují artifactId a definici parent projektu:

```
<parent>  
  <groupId>cz.jiripinkas.example</groupId>  
  <artifactId>eshop-spring</artifactId>  
  <version>0.0.1-SNAPSHOT</version>  
</parent>  
  
<artifactId>eshop-repository</artifactId>
```

# Module project II.

- Také obsahují závislosti na jiné moduly:

```
<parent>

  <groupId>cz.jiripinkas.example</groupId>

  <artifactId>eshop-spring</artifactId>

  <version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>eshop-service</artifactId>

<dependencies>

  <dependency>

    <groupId>cz.jiripinkas.example</groupId>

    <artifactId>eshop-repository</artifactId>

    <version>0.0.1-SNAPSHOT</version>

  </dependency>

</dependencies>
```

# Module project III.

- Také mohou obsahovat další závislosti, které se nepoužívají v ostatních modulech.
- Příklad na uvedenou třívrstvou architekturu je zde:
  - <https://github.com/jirkapinkas/example-eshop-spring>
- Jak na automatickou změnu verzí všech projektů? Jsou dvě možnosti:
  - Při používání Maven Release Plugin se o to nemusíte starat, ten se o změnu verzí stará sám.
  - Také můžete v parent pom.xml přidat do tagu properties:

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

A poté všude místo konkrétní verze používat `${current.version}`

# Parent project (nikoli multi-module)

- Pokud ve firmě tvoříte neustále podobné projekty, které používají stále stejné dependency a pluginy, pak můžete vytvořit projekt typu pom ve kterém je nastavíte a Váš projekt nastavíte jako potomka tohoto parent projektu:

```
<parent>  
  
  <groupId>cz.jiripinkas</groupId>  
  
  <artifactId>webapp</artifactId>  
  
  <version>0.0.1-SNAPSHOT</version>  
  
</parent>
```

- Tento parent není v nadřazeném adresáři projektu, nejedná se o multi-module projekt!
- Bohužel jeden projekt může mít max. jednoho předka, čili není to moc flexibilní. Pro větší flexibilitu se dá k obdobnému účelu použít skupina dependencies (viz. následující snímek) – u ní ale není možné definovat pluginy.



# DependencyManagement

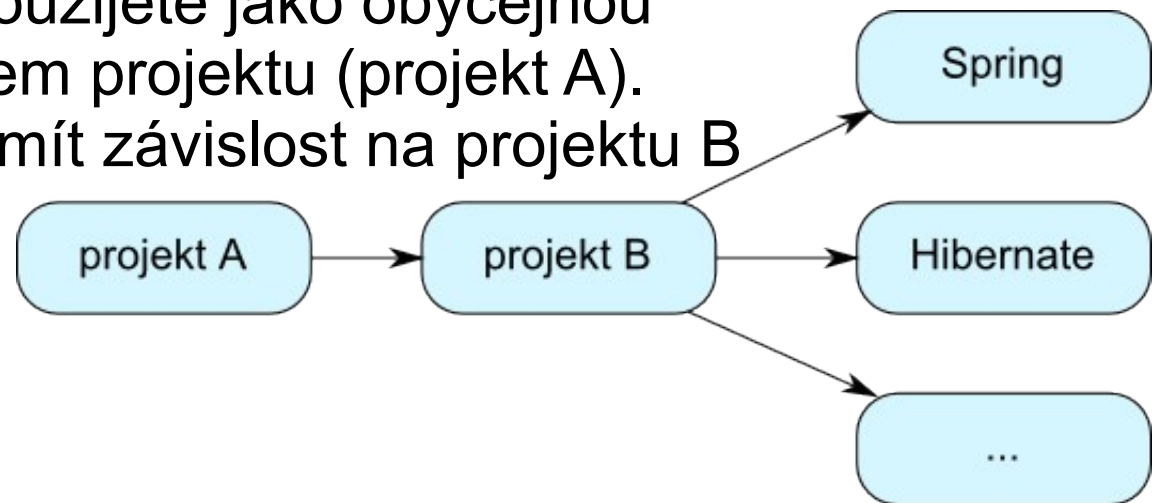
- V předkovi může být tag `<dependencyManagement>`, ve kterém mohou být definované verze dependencies.
- V potomkovi se poté čísla verzí neuvádí:

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

Poznámka: Typické použití je vidět u Spring Boot aplikací

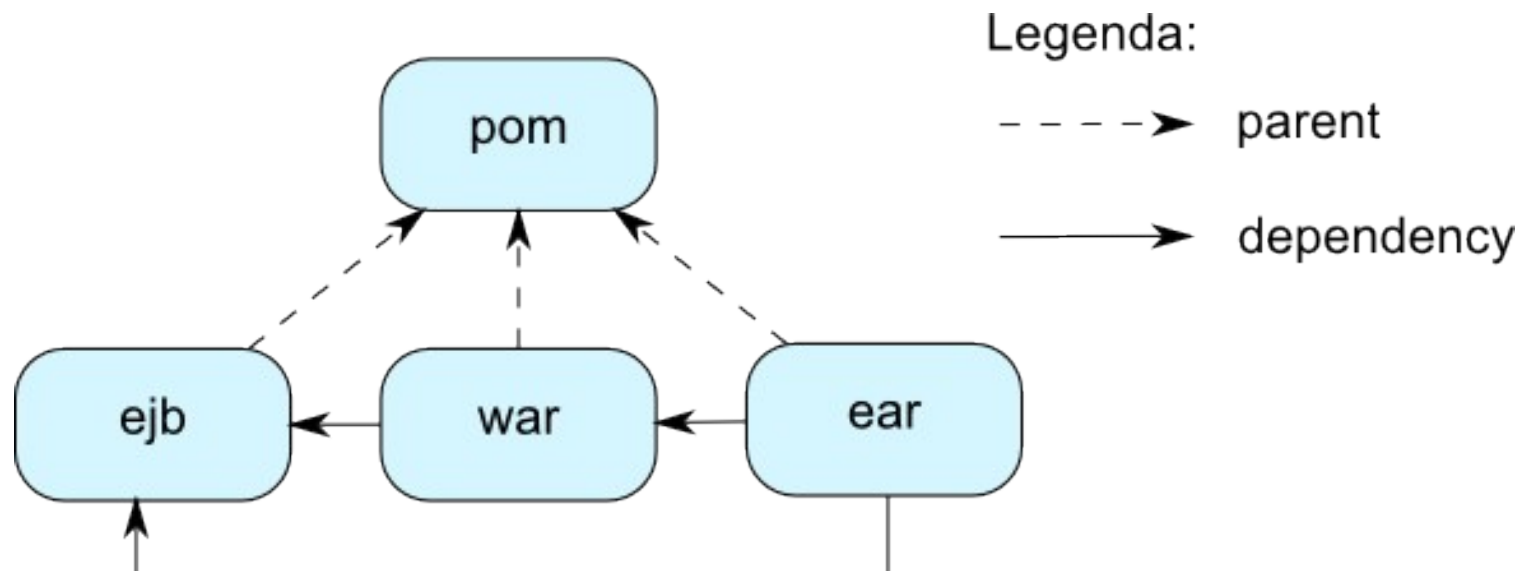
# Skupina dependencies

- Pokud máte více projektů, které používají stále stejné dependency (například Hibernate + Spring + slf4j + log4j + hsqldb), pak tyto dependency nemusíte specifikovat u každého projektu zvlášť, ale můžete vytvořit skupinu dependencies, kterou pak přiřadíte k projektu.
  - Skupina dependencies je prakticky Maven projekt typu pom (projekt B) v němž jsou definované jednotlivé dependency (Hibernate, Spring, ...).
  - Tento projekt poté použijete jako obyčejnou dependency ve Vašem projektu (projekt A). Projekt A tedy bude mít závislost na projektu B a tranzitivně tedy i na dependencies, které jsou v projektu B definovány.



# EAR archetype

- Pro jednoduché vytvoření EAR projektu je například tento archetype:
  - `jboss-javaee6-ear-webapp`
- EAR projekt je prakticky multi-module Maven projekt (pom), který se skládá ze tří projektů (ejb, war, ear), které mají mezi sebou závislosti:



# Filtering

- Někdy je nutné mít jiné nastavení context parametrů ve web.xml v development, produkčním, ... prostředí (například při použití JSF). Jak na to?
- Můžete použít filtering, pomocí kterého místo skutečné hodnoty do web.xml nastavíte proměnnou pomocí expression language a při buildování ji nahradíte za skutečnou hodnotu například z properties souboru.
- Příklad:
  - <https://community.jboss.org/wiki/HowToConfigureJavaEEApplicationToApplyDifferentSettingsinWebxmlEtcForVariousEnvironmentsByMaven>

Na následujících stránkách je rozchození filteringu při použití embedded Jetty serveru (to je složitější než pro JBoss).

# web.xml – development vs. production – Jetty I.

Dovnitř tagu build!!!:

```
<filters>

  <filter>${basedir}/src/main/filters/${filter.name}.properties</filter>

</filters>

<resources>

  <resource>

    <directory>src/main/webapp/WEB-INF</directory>

    <filtering>true</filtering>

    <targetPath>../jettyFilteredResources</targetPath>

  </resource>

  <resource>

    <directory>src/main/resources</directory>

    <targetPath>../classes</targetPath>

  </resource>

</resources>
```

# web.xml – development vs. production – Jetty II.

- Použití Jetty pluginu (uvnitř build tagu):

```
<plugin>

  <groupId>org.eclipse.jetty</groupId>

  <artifactId>jetty-maven-plugin</artifactId>

  <version>9.1.3.v20140225</version>

  <configuration>

    <webAppConfig>

      <descriptor>target/jettyFilteredResources/web.xml</descriptor>

    </webAppConfig>

    <scanIntervalSeconds>3</scanIntervalSeconds>

  </configuration>

</plugin>
```

# web.xml – development vs. production – Jetty III.

- Dvnitř tagu project:

```
<profiles>
  <profile>
    <id>dev</id>
    <activation> <activeByDefault>true</activeByDefault> </activation>
    <properties> <filter.name>dev</filter.name> </properties>
  </profile>
  <profile>
    <id>prod</id>
    <properties> <filter.name>prod</filter.name> </properties>
  </profile>
</profiles>
```

# web.xml – development vs. production – Jetty IV.

- src/main/filters/dev.properties:

```
jsf.projectStage=Development
```

```
jsf.faceletsRefreshPeriod=0
```

- src/main/filters/prod.properties:

```
jsf.projectStage=Production
```

```
jsf.faceletsRefreshPeriod=-1
```



# web.xml – development vs. production – Jetty V.

- web.xml:

```
<context-param>
```

```
    <param-name>javax.faces.FACELETS_REFRESH_PERIOD</param-name>
```

```
    <param-value>${jsf.faceletsRefreshPeriod}</param-value>
```

```
</context-param>
```

```
<context-param>
```

```
    <param-name>javax.faces.PROJECT_STAGE</param-name>
```

```
    <param-value>${jsf.projectStage}</param-value>
```

```
</context-param>
```

# web.xml – development vs. production – Jetty VI.

- Spuštění:

```
mvn jetty:run
```

```
mvn package -P prod
```