Vacademy

vacademy.cz vzdělání IT profesionálů

Maven

příručka programátora

Libor Jelínek

Od instalace, vysvětlení základů, správy závislostí po použití pluginů a integraci s Jenkins

Vítá vás Maven - příručka programátora!

Vítáme budoucí i současné uživatele a programátory v této příručce Maven. Tato publikace může být skvělým doplňkem k našemu školení Maven, ale je koncipována jako zcela samostatná. Nabízíme ji zdarma všem návštěvníkům a samozřejmě i studentům tohoto kurzu. Budeme rádi, když vám pomůže naučit se Maven.

> **Tip**
>
> Líbí se vám tato knížka? Přijďte na školení od autorů této příručky na Vacademy.cz!

Licenční ujednání

Copyright © 2018 Virtage Software. Tato příručka je publikována pod licencí CC BY-NC-SA 4.0 (Uveďte původ-Neužívejte dílo komerčně-Zachovejte licenci 4.0 Mezinárodní). Tato licence dovoluje text sdílet a distribuovat v jakémkoli formátu nebo médiu *kromě použití pro výdělečné účely*. Text můžete upravovat a pozměňovat, pokud zachováte stejnou licenci. Vystavitel licence může tyto podmínky v budoucnu upravovat. Při sdílení a šíření je nutné uvést původ např. URL odkazem.

1. Úvod

1.1. Proč Maven a srovnání s Ant

- *„convention over configuration"*
- nezávislost na jazyku, typu projektu ap.
- nezávilost na IDE
- správa závilostí
- standardizuje software lifecycle management
- vnucuje test-driven přístup
- automatizované generování dokumentace, buildů atp. – vhodné pro CI
- modulárně navržený

1.1.1. Historie Maven

- 2002 - vznikl jako subprojekt Jason van Zyla v rámci Apache Turbine
- 2003 - Apache top-level project
- 2004 - Maven 1
- 2005 - Maven 2 - zásadní význam. Maven, tak jak ho známe nyní.
- 2010 - Maven 3 - prakticky zpětně kompatibilní s Maven 2

1.1.2. Srovnání Ant a Maven

Občas srovnáván, ale jedná se o dva poměrně odlišné nástroje.

	Ant	**Maven**
vznik	pokračovatel např. Make z C	založil novou generaci nástrojů
obvyklé jméno skriptu	`build.xml`	`pom.xml`
syntaxe skriptu	XML	XML
obsah skriptu	*jak* něco dělat	*co* dělat bez jak
podpora v IDE	perfektní	velmi dobrá

	Ant	Maven
nový vývojář projektu musí znát	konkrétní `build.xml`	Maven, nikoli konkrétní `pom.xml`

1.2. Instalace Maven

Instalace Mavenu v Linuxu, Mac OS X i Windows je velmi podobná a spočívá jen v rozbalení archivu a nastavení několika proměnných prostředí (environment variables).

1. Stáhneme si a rozbalíme archív Mavenu. Typické umístění bývá
    - v Linuxu /usr/local/apache-maven/apache-maven-<verze>
    - na Windows např. C:\Program Files\Apache Software Foundation\apache-maven-<verze>.
2. Nastavíme proměnnou prostředí `M2_HOME` na tuto složku.
3. Nastavíme proměnnou prostředí `M2` na `M2_HOME/bin/`.
4. Volitelně můžeme do proměnné prostředí `MAVEN_OPTS` zadat parametry pro JVM volanou Maven
5. Přidáme `M2` do `PATH`.

Důležité

Maven rovněž vyžaduje nastavení systémové proměnné `JAVA_HOME`. Měla by ukazovat na složku JDK (JRE nestačí).

Výpis `mvn –version` nám řekne, jestli jsme vše nastavili správně:

```
$ mvn --version
Apache Maven 3.0.4 (r1232337; 2012-01-17 09:44:56+0100)
Maven home: /usr/share/maven
Java version: 1.7.0_45, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.7.0_45.jdk/Contents/Home/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.8.5", arch: "x86_64", family: "mac"
Restarujte terminál/znovu se přihlašte a vyzkoušejte mvn –version.
```

1.2.1. Linux

V Ubuntu a dalších linuxech tyto kroky znamenají přidat do `~/.bash_aliases` (soubor vytvoříme, pokud neexistuje) řádky:

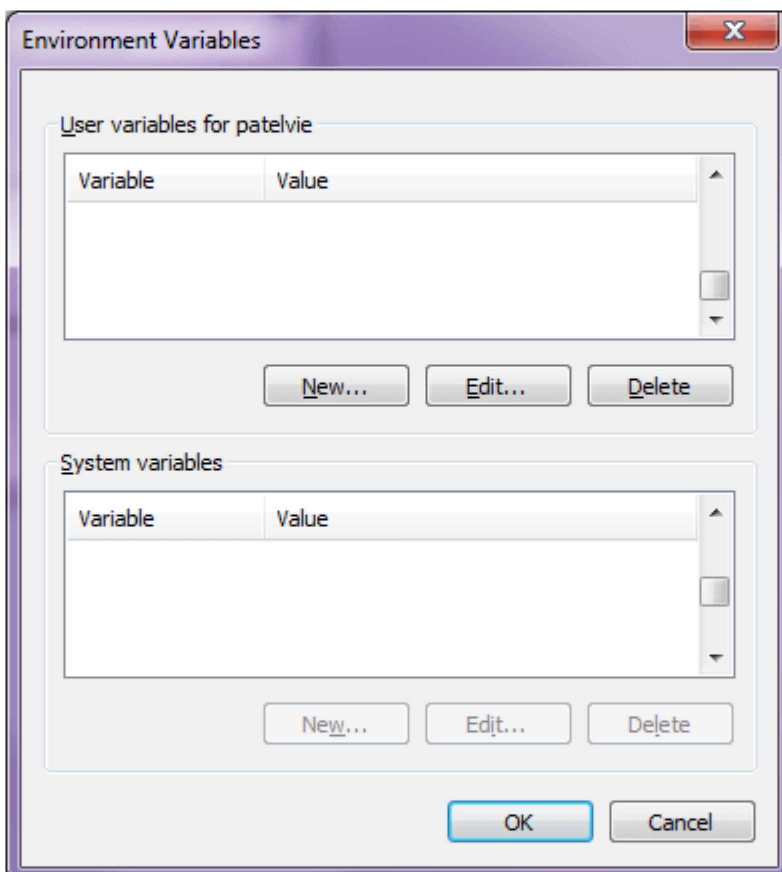
```
export M2_HOME=/usr/local/apache-maven/apache-maven-3.0.4/
export M2=$M2_HOME/bin/
export PATH=$PATH:$M2
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-i386/
```

1.2.2. Mac

Pro uživatele Mac OS X Lion (10.7) a vyšší je dobrou zprávu, že Maven 3 je již součástí jejich operačního systému ve složce `/usr/share/maven/`.

1.2.3. Windows

Ve Windows použijeme ovládací panel Environment Variables (Proměnné prostředí) a nastavíme výše uvedené proměnné prostředí.



Dialog pro nastavení proměnných prostředí ve Windows

1.3. Konfigurace Maven

Konfigurační soubor `settings.xml` určuje nastavení specifické pro konkrétní prostředí. Z toho důvodu by neměl být sdílen mezi vývojáři.

Maven lze konfigurovat z globálního (společný všem uživatelům) nebo uživatelského (jen pro daného uživatele) souboru `settings.xml`. Existují-li oba soubory, Maven nastavení sloučí s předností pro uživatelské nastavení.

globální nastavení (system-wide)	`$M2_HOME/conf/settings.xml`
uživatelské nastavení (user-wide)	`~/.m2/settings.xml`

Elementy v settings.xml odpovídají jednotlivým možnostem nastavení, které popíšeme za okamžik.

Ukázka `settings.xml` (zkráceno)

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository>...</localRepository>
  <interactiveMode>...</interactiveMode>
  <usePluginRegistry>...</usePluginRegistry>
  <offline>...</offline>
  <pluginGroups>...</pluginGroups>
  <servers>...</servers>
  <mirrors>...</mirrors>
  <proxies>...</proxies>
  <profiles>...</profiles>
  <activeProfiles>...</activeProfiles>
</settings>
```

Podívejme se stručně ty nejdůležitější:

- `<localRepository>` – umístění lokálního repozitáře (standardně v `~/.m2/` (pro Maven 2 i 3))
- `<offline>` – build bude pracovat vždy v offline režimu
- `<pluginGroups>` – doplnění groupId pro pluginy, které ho neuvádějí (automaticky obsahuje org.apache.maven.plugins a org.codehaus.mojo).

- `<servers>` – nastavení cesty URL, hesel serverů ap. ve kterých pro stahování a deployment určený elementy `<repositories>` a `<distributionManagement>` POM souboru.
- `<activeProfiles>` – jména aktivních profilů (používá-li `pom.xml` profily)

Detailní popis najdeme v manuálu Maven.

Tip

Ještě lepší popis významu jednotlivých elementů najdeme přímo v souboru `M2_HOME/conf/settings.xml`. Tento soubor taky můžeme použít jako šablonu pro naše uživatelské nastavení a odmazat z něj elementy, které nepotřebujeme.

2. Základy Maven

2.1. Závilost, plugin, artifact

Jednou z hlavních příčin úspěchu Maven je správa závilostí. Váš program si určuje jaký další software

- potřebuje jako závilosti (*dependency, závilost*)
- potřebuje pro sestavení (*plugin* nebo někdy také *MOJO*).

Váš sestavený program se může stát závilostí pro další program atp. Stejně tak program, který píšete může být plugin pro Maven. Proto se obecně setkáváme s pojmem *artifact (artefakt)* zahrnující jak závilosti, tak pluginy.

Pluginy bychom mohli ještě rozdělit na pluginy potřebné k sestavení projektu a pro generování reportů (viz manuál Maven).

2.2. Coordinates

Pro jednoznačnou identifikaci artifactů v Maven repozitáři slouží tzv. coordinates (souřadnice)

1. ve zkráceném tvaru *groupId*:*artifactId*:*version* nebo
2. v úplném *groupId*:*artifactId*:*packaging*:*classifier*:*version*.

Příklady zkrácených coordinates

groupId	artifactId	version
org.eclipse.jetty	jetty-annotations	9.0.3.v20130506
org.apache.maven.doxia	doxia	1.0-alpha-9

Příklady úplných coordinates

groupId	artifactId	packaging	classifier	version
org.eclipse.jetty	jetty-annotations			9.0.3.v2013
org.apache.maven.doxia	doxia	pom	experimental	1.0-alpha-9

Všimněte si, že mohou chybět vlastnosti

1. *packaging* neboli typ balení, pak se přepokládá hodnota jar
2. *classfier* pro upřesnění jinak kolidujících artifactů (bez defaultní hodnoty)

Více v manuálu Maven.

2.3. Závilosti

2.3.1. Přidání závilosti

Závilost na běžném artifactu i na pluginu do Maven projektu přidáváme v souboru `pom.xml` jako element `<dependency>`. V `pom.xml` souboru vypadá definice závilosti na pluginu takto:

Příklad definice závilosti

```
<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">
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.0</version>
      <type>jar</type>
      <scope>test</scope>
      <optional>true</optional>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

Příklad definice pluginu

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

```
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.0</version>
        <extensions>false</extensions>
        <inherited>true</inherited>
        <configuration>
          <classifier>test</classifier>
        </configuration>
        <dependencies>...</dependencies>
        <executions>...</executions>
      </plugin>
    </plugins>
  </build>
</project>
```

2.3.2. Transitive závilost

(Užitečný český překlad nás nenapadá.) Závilosti obvykle mají další závilosti, ty zase další atd. atd. Stejně tak i pluginy. Skvělou vlastností Maven je, že se postará o tyto „skryté“ tzv. transitive závilosti, tedy závilosti našich závilostí. Transitive závilosti musí být pochopitelně k nalazení ve známých repozitářích.

2.3.3. Scopes (obor platnosti)

Možná jste si povšimli elementu `<scope>` ve výše uvedených příkladech závislostí. Každá závilost platí pouze v určitném scope (oboru platnosti). Základními zabudovanými pěti typy oborů platnosti jsou:

Zabudované scope (obory platnosti) závilostí

Obor platnosti	Popis
compile	Nejčastější. Závilost je potřebná pro zkompilování.
test	Závilost potřebná jen pro spuštění testů.
provided	Potřebná pro kompilaci, ale nebude součástí výsledného JAR/WAR/... Závilost bude během spuštění dostupná na classpath (poskytne aplikační kontejner ap.)
system	Potřebná pro kompilaci, ale cestu musíme zadat z místa na disku. V praxivhodná jen při mavenizaci.
import	Speciální obor platnosti pro multi-module projekty (typ balení pom).

2.3.4. Pořadí závislostí

Pozor na to, že na pořadí závilostí záleží! Tak, jak je napíšeme v `pom.xml`, v takovém pořadí budou v classpath. To může být důležité např. pro knihovny upravující třídy (weaving) jako třeba při testování s knihovnou JMockit.

2.4. Lifecycle, phase, plugin, goal

Maven rozděluje provádění sestavování, generování dokumentace a dalších operací do čtyř úrovní hierarchie:

- **lifecycle (životní cyklus)** – nejvyšší rozdělení zobecňuje životní cyklus jakéhokoli software projektu bez ohledu na programovací jazyk
- **phase (fáze)** – lifecycle obsahuje kroky zvané phases
- **plugin** – jednotka balení a distribuce (typicky JAR soubor) obsahující jeden nebo více goalů
- **goal (cíl)** – konkrétní jednotlivá úloha identifikovaná ve formátu plugin:goal (např. `tomcat7:run`, `jar:sign`)

Tyto vztahy bychom tedy mohli znázornit jako

lifecycle → phases → plugin → goals

Všechny Maven projekty mají tři životní cykly:

1. clean (čistící)
2. default (výchozí, hlavní)
3. site (webové sídlo)

Jednotlivé fáze životních cyklu jsou pro jakýkoli Maven projekt vždy stejné. Podle typu balení artifactu (`<packaging>` v POM souboru) se liší napojené goals.

Příklad připojených goalů stejné phase lišících se podle packaging typu

Packaging	Lifecycle	Phase	Připojený goal
`jar`	default	package	`jar:jar`

Packaging	Lifecycle	Phase	Připojený goal
`ear`	default	package	`ear:ear`
`pom`	default	package	`site:attach-descriptor`

Více opět manuál Mavenu.

2.4.1. Clean lifecycle

Jak napovídá název, clean lifecycle vymaže všechny vygenerované a zkompilované soubory (`.class` ap.) z výstupní složky (`target/`).

Phase	Popis
`pre-clean`	
`clean`	

2.4.2. Default lifecycle

Phase	Popis
`validate`	
`initialize`	
`generate-sources`	
`process-sources`	
`generate-resources`	
`process-resources`	
`compile`	Zkompiluje nalezené zdrojové kódy
`process-classes`	
`generate-test-sources`	

Phase	Popis
`process-test-sources`	
`generate-test-resources`	
`process-test-resources`	
`test-compile`	Zkompiluje jednotkové (unit) testy
`test`	Spustí jednotkové (unit) testy
`prepare-package`	
`package`	Vytvoří distribuovatelnou podobu projektu (JAR, WAR, EAR ap.)
`pre-integration-test`	
`integration-test`	Spustí integrační testy. Pokud je třeba, provede nasazení (deloy) distribuovatelné podoby projektu do testovacího prostředí.
`post-integration-test`	
`verify`	
`install`	Spoustí kontroly ověřující, že distribuovatelná podoba projektu (balíček) je platná (validní)
`install`	Umístí balíček do lokálního Maven repozitáře
`deploy`	Nahraje balíček do vzdáleného Maven repozitáře

2.4.3. Site lifecycle

Phase	Popis
`pre-site`	
`site`	Vygeneruje informační web a HTML reporty
`post-site`	
`site-deploy`	Nahraje vygenerovaný web na web server

2.4.4. Provádění phase nebo goal

Z příkazové řádky můžeme vyvolat provádění phase nebo goalu. Více phases/goalů oddělujeme mezerou mezi sebou.

Syntaxe pro spuštění phase je jen její název (např. `package`), syntaxe pro goal je *<plugin>*:*<goal>*. Např.:

```
$ mvn clean jar:test-jar
```

provede postupně

1. všechny goals fáze clean patřící do cyklu clean
2. pouze goal test-jar z pluginu jar patřící do cyklu default

Důležité

Přímým vyvoláním goalu přeskočíme všechny fáze cyklu a provede se opravdu jen zadaný goal. Naopak vyvoláním celé fáze provede i všechny předchozí goals.

Následující příklad postupně provede všechny fáze (validate, initialize, generate-sources, process-sources, generate-resources, process-resources) až po samotné compile:

```
$ mvn compile
```

2.5. Soubor pom.xml

Podle přítomnosti souboru `pom.xml` (nebo jen POM) poznáme, že projekt je Maven projekt. *POM neboli Project Object Model* je reprezentací Maven projektu XML syntaxí. Slovo „projekt“ zde má velmi široký pojem a neznamená pouze zdrojový kód. V POMu evidujeme také řadu dalších informací o projektu jako např.

- konfigurační soubory
- jména a role programátorů
- issue trackery
- umístění a typ verzovacího systému

2.5.1. Ukázka pom.xml

Minimálním povinným základem každého POM je určení coordinates, tedy elementy

- `<groupId>` – ID skupiny, bývá nejčastěji organizace nebo organizace a projekt v Java package notaci (např. `cz.virtage.utility`)
- `<artifactId>` – ID tohoto artifactu neboli jméno projektu (např. `diskcleaner`)
- `<version>` – verze artifaktu (projektu) (např. `1.5.2a`)

K tomu je třeba určit verzi POM modelu 4.0.0 platí pro Maven 2 a 3. Minimální platný POM by vypadal např. takto

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>cz.virtage.utility</groupId>
    <artifactId>diskcleaner</artifactId>
    <version>1.0-SNAPSHOT</version>
</project>
```

Zkrácená ukázka pom.xml (většina elementů je nepovinných):

```
<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>

  <!-- The Basics -->
```

```
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <packaging>...</packaging>
  <dependencies>...</dependencies>
  <parent>...</parent>
  <dependencyManagement>...</dependencyManagement>
  <modules>...</modules>
  <properties>...</properties>

  <!-- Build Settings -->
  <build>...</build>
  <reporting>...</reporting>

  <!-- More Project Information -->
  <name>...</name>
  <description>...</description>
  <url>...</url>
  <inceptionYear>...</inceptionYear>
  <licenses>...</licenses>
  <organization>...</organization>
  <developers>...</developers>
  <contributors>...</contributors>

  <!-- Environment Settings -->
  <issueManagement>...</issueManagement>
  <ciManagement>...</ciManagement>
  <mailingLists>...</mailingLists>
  <scm>...</scm>
  <prerequisites>...</prerequisites>
  <repositories>...</repositories>
  <pluginRepositories>...</pluginRepositories>
  <distributionManagement>...</distributionManagement>
  <profiles>...</profiles>
</project>
```

2.5.2. Dědičnost POM

Pokud některé nastavení v POM projektu vynecháme, Maven slučováním postupně hledá v

- rodičovském POM (pro multi-module Maven projekty)
- „Super POM“, když rodičovský chybí nebo po té co nenajde všechno nastavení v rodičovském

Super POM mj. určuje URL The Central Repository, standardní adresářovou strukturu Maven projektu atd. Jeho obsah je „zadrátován“ ve zdrojovém kódu Mavenu a může se pro každou verzi mírně lišit.

Tip

Super POM vašeho Maven si můžeme prohlédnout příkazem `mvn help:effective-pom`.

Viz manuál Maven.

2.5.3. Properties (vlastnosti)

Properties jsou velmi podobné stejné funkci v Antu nebo v operačním systému. Jsou to dvojice klíč-hodnota, obojí typu řetězec, které nadefinujeme na jednou místě a opakovaně použijeme v POM souboru. Hodí se pro uložení takových hodnot jako cesty, URL, uživatelská jméno ap.

Všechny vlastnosti se používají zápisem `${_property_}` a vytvářejí se v elementu `<properties>`:

Definice:

```
<properties>
    <color>green</color>
</properties>
```

Použití:

```
<build>
    <finalName>${artifactId}-${version}-${color}</finalName>
</build>
```

Některé užitečné zabudované properties

Vlastnosti	Popis
`${project.basedir}`	Složka s POM souborem.
`${project.artifactId}`	Hodnota z `<artifactId>`
`${project.groupId}`	Hodnota z `<groupId>`

Vlastnosti	Popis
`${project.version}`	Hodnota z `<version>`
`${project.build.directory}`	Výstupní složka (obvykle `target/`).
`${project.build.sourceEncoding}`	Kódování zdrojových souborů (dnes obvykle UTF-8). Maven varuje, pokud není definováno.
`${env._<proměnná>_}`	Vyhodnotí proměnnou prostředí OS, např. `${env.HOME}` vrátí domovskou složku uživatele.

Např. velmi často potřebnou proměnnou pro správné fungování kompilátoru je project.build.sourceEncoding, tedy kódování zdrojových `.java` souborů.

2.6. Repozitáře

Repozitář je složka na lokálním nebo vzdáleném souborovém systému, kterou Maven udržuje a používá k vyhledávání a ukládání závilostí a pluginů. Obsahem složky repozitáře je mnoho dalších složek a souborů se speciálním významem pro Maven.

2.6.1. Lokální repozitář

Vždy přítomný lokální repozitář je v `~/.m2/repository/`, který funguje v zásadě jako cache a najdete zde dříve použité stažené artefakty ze vzdálených repozitářů.

2.6.2. The Central Repository

Druhý repozitář, který nemusíme nastavovat (definuje ho Super POM) a přesto v něm Maven bude hledat se nazývá The Central Repository

- URL pro člověka (webový prohlížeč): http://search.maven.org/
- URL pro Maven: http://repo.maven.apache.org/maven2

V něm jsou prakticky všechny myslitelné knihovny a závilosti, které můžeme potřebovat.

Občas přesto narazíme na to, že nějaký software v The Central Repository chybí (buď třetí strany, proprietární nebo náš vlastní software) a je třeba ho tzv. mavenizovat.

2.6.3. Vlastní repozitář

Je možné vytvářet vlastní repozitáře, např. v rámci firmy. Lze to sice jen z příkazové řádky, ale to se hodí jen pro několik málo spravovaných závislostí/pluginů. Pro velké množství položek se v praxi používají nástroje třetích stran jako Nexus.

2.6.4. Přidání repozitáře do POM

Příklad přidání nového vzdáleného repozitáře do POM projektu:

```
<repositories>
     <repository>
             <id>ourcompany-repo</id>
             <url>https://somelocalserver</url>
     </repository>
</repositories>
```

3. Projekty

3.1. Vytvoření Maven projektu

To, že v Maven je opravdu všechno plugin dokazuje, že i vytváření Maven projektů má na starosti plugin s jménem archetype, resp. jeho goal generate. Ten očekává zadání groupId a artifactId v parametrech `-DgroupId` a `-DartifactId`:

```
$ mvn archetype:generate -DgroupId=... -DartifactId=...
```

Poznámka

Ve starších návodech se setkáme s archetype:create namísto archetype:generate. Goal archetype:create byl zavržen (deprecated) a neměli bysme ho již používat.

Když neurčíme žádný archetyp, Maven založí jednoduchou Java aplikaci „Hello world“ a test (ve skutečnosti použije zabudovaný archetyp maven-archetype-quickstart). Hello world většinou není, co chceme a proto se naučíme třetí důležitý parametr a to `-DarchetypeArtifactId`. Např. vytvoření jednoduché webové Java aplikace:

```
$ mvn archetype:generate -DgroupId=... -DartifactId=.. -DarchetypeArtifactId=ma
```

Pokud nezadáme úplně všechny argumenty goalu archetype:generate spustí se *interaktivní mód*, kdy se nás Maven postupně dotazuje na chybějící údaje. Dokonce lze napsat jen

```
$ mvn archetype:generate
```

a postupně vyplňovat jednotlivé údaje. Výstup variant příkazu achetype:generate bude podobný tomuto:

```
$ mvn archetype:generate -DgroupId=org.virtage.maven -DartifactId=defaultGoal
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-archetype-plugin:2.2:create (default-cli) @ standalone-pom ---
[WARNING] This goal is deprecated. Please use mvn archetype:generate instead
[INFO] Defaulting package to group ID: org.virtage.maven
[INFO] ------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Old (1.x) Archetype
[INFO] ------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: org.virtage.maven
[INFO] Parameter: packageName, Value: org.virtage.maven
[INFO] Parameter: package, Value: org.virtage.maven
[INFO] Parameter: artifactId, Value: defaultGoal
[INFO] Parameter: basedir, Value: /Users/libor/Ubuntu One/Trainings/courseware/
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: /Users/libor/Ubuntu One
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.065s
[INFO] Finished at: Tue Oct 22 17:11:33 CEST 2013
[INFO] Final Memory: 11M/112M
[INFO] ------------------------------------------------------------------------
```

Příkaz vytvoří složku pojmenovanou jako groupId a v ní standardní adresářovou strukturu:

```
└── defaultGoal
    ├── pom.xml
    └── src
        ├── main
        │   └── java
        │       └── org
        │           └── virtage
        │               └── maven
        │                   └── App.java
        └── test
            └── java
                └── org
                    └── virtage
                        └── maven
                            └── AppTest.java
```

3.2. Archetypy

Archetypy jsou v terminologii Maven šablony pro vytváření nových projektů.

3.2.1. Zabudované archetypy

Když zkusíme interaktivní vytvoření zjistíme, že z centrálního repozitáře Maven máme archetypů k dispozici mnohem víc. Samotný Maven obsahuje jen několik málo archetypů. Podívejme se na některé užitečné z nich:

- `maven-archetype-j2ee-simple` – velmi jednoduchá vzorová J2EE aplikace
- `maven-archetype-quickstart` – javovský Maven project s „Hello world“ jako ukázkou
- `maven-archetype-simple` – prázdný javovský Maven projekt
- `maven-archetype-webapp` – jednoduchá Java servlet aplikace (WAR)

Seznam a podrobnosti zabudovaných archetypů hledejte na manuálu Maven.

3.2.2. Vlastní archetyp

Můžeme rovněž definovat vlastní nový archetyp, ale to je mimo rozsah této učebnice. Zájemce odkážeme na manuál Maven.

3.3. Mavenizace

Pod pojmem mavenizace (angl. podst jm. „mavenization“, resp. slov. „to mavenize“) myslíme proces převodu knihovny nebo programu do podoby Maven artifactu. Tedy, aby mohl být použit jako závilost v Maven projektu.

Důležité

Nejprve se ubezpečíme, že knihovna ještě není mavenizována prohledáním Maven Central. Když ji tady nenajdeme, budeme pátrat

na webu projektu a zkusíme také vyhledávač s „<knihovna> maven“ ap. Ne všechny projekty posílají své artefakty do Maven Central a hostují si Maven repozitář sami.

Způsobů jak mavenizovat JAR se nabízí více. Ukážeme si několik možných postupů.

Tip

Ať zvolíme jakoukoli z následujících možností, nezapomeneme umístit JAR soubor do verzovacího systému, aby byl dostupný všem.

3.3.1. Repozitář v projektu (in-project repository)

Nejpracnější, avšak profesionální postup, který nemá nevýhody zbývajících technik využívá toho, že repozitář je běžná složka a může být umístěn i v kořenové složce Maven projektu.

Postup vytvoření repozitáře v projektu

Vytvoříme repozitář goalem install:install-file do složky např. `jar-repo` nastavenou v parametru `-DlocalRepositoryPath`. Musíme zadat i další údaje budoucího Maven artifactu (ve Windows uvedeme příkaz na jediné řádce bez `/`):

```
$ mvn install:install-file -DlocalRepositoryPath=jar-repo -Dfile=pre-maven-proj
-DgroupId=org.virtage -DartifactId=premaven -Dpackaging=jar -Dversion=1.0
```

Maven pro nás vytvoří tuto strukturu:

```
jar-repo/
└── org
    └── virtage
        └── premaven
            ├── 1.0
            │   ├── premaven-1.0.jar
```

```
            │   └── premaven-1.0.pom
            └── maven-metadata-local.xml
```

Varování

Příklad funguje s maven-install-plugin od verze 2.3.1. Pokud máme problémy, nahradíme příkaz na `org.apache.maven.plugins:maven-install-plugin:2.3.1:install-file`. Případně na aktuální verzi v době čtení (viz stránky maven-install-plugin).

Nyní do POM přidáme repozitář uvnitř projektu:

```
<repositories>
    <repository>
        <id>jar-repo</id>
        <url>file://${basedir}/jar-repo/</url>
    </repository>
</repositories>
```

A samozřejmě nakonec samotnou závilost:

```
<dependency>
    <groupId>org.virtage</groupId>
    <artifactId>premaven</artifactId>
    <version>1.0</version>
    <scope>compile</scope>
</dependency>
```

3.3.2. Instalace do lokálního repozitáře

Nejrychlejší a nejjednodušší je instalace JARu do lokálního repozitáře (`~/.m2/repository/`) s pomocí goalu install:install-file (na Windows na jednom řádku bez `/`):

```
$ mvn install:install-file -Dfile=<path-to-file> -DgroupId=<group-id> \
    -DartifactId=<artifact-id> -Dversion=<version> -Dpackaging=<packaging-type>
```

Upozornění

Tento přístup je vhodný jen pro lokální testování. Zásadní nevýhodou je, že aby sestavení proběhlo úspěšně i u dalšího programátora, ten musí JAR nějak získat a pak ručně nainstalovat do svého lokálního repozitáře.

3.3.3. System scope

Druhou a **ještě méně doporučeníhodnou cestou** je využití oboru platnosti `system`. Obor platnosti `system` umožňuje zadat libovolnou cestu na disku v elementu <systemPath>, která se připojí do kompilační classpath.

```
<dependency>
    <groupId>org.virtage.maven</groupId>
    <artifactId>mavenizing.systemscope</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <scope>system</scope>
    <systemPath>/home/libor/workspace/mavenizing.systemscope/some-dirty.jar</sy
</dependency>
```

Cesta v <systemPath> musí být vždy absolutní, tj. platná jen pro daný počítač.

Zmírněním je možnost použív cestě zabudovanou property `${project.basedir}` ukazující do složky s POMem:

```
<systemPath>${project.basedir}/jars/dirty.jar</systemPath>
```

System scope svou podstatou zaručuje základní výhodu Maven v podobě transitive závislosti. Proto jsou také zavržené ???(deprecated) a je pravděpodobné, že budou v příštích verzí Maven úplně odstraněny.

3.4. Multi-module Maven projekt

Maven podporuje projekt složený z více samostatných podprojektů zvaných *moduly*. Je to podobné jako dědičnost POMů. Multi-module projekt se výborně hodí, když máme velkou aplikaci skládající se z částí jako např.

- desktop frond-end napsaný ve Swingu
- webový front-end napsaný v servletech a JSP
- společnou knihovnu (JAR)
- back-end běžící na serveru
- dokumentace a nápověda projektu

Můžeme pracovat samostatně na jednotlivých částech, ale když chceme odeslat zákazníkovi novou verzi aplikace, potřebujeme všechny její části (moduly) sestavit. Maven zjistí vzájemné závilosti modulů, sestaví správné pořadí sestavení??? a vytvoří potřebné výstupy. Tento mechanismus se nazývá *reactor*. Můžeme si tohoto názvu všimnout při sestavení celého multi-module projektu:

```
[INFO] Reactor Summary:
[INFO]
[INFO] parent .................................. SUCCESS [0.002s]
[INFO] richclient .............................. SUCCESS [6.043s]
[INFO] webclient ............................... SUCCESS [1.324s]
```

3.4.1. Top-level POM

Multi-module projekt je sám Maven projekt a tudíž má pom.xml, kterému můžeme říkat top-level POM. Ten v sekci `<modules>` odkazuje na jednotlivé moduly. Jméno modulu musí odpovídat složce ve které je modul umístěn.

Top-level POM

```
<project>
    ...
    <modules>
        <module>richclient</module>
        <module>webclient</module>
    </modules>
    ...
</project>
```

Top-level projekt vytvoříme stejně jako běžný Maven projekt:

```
mvn archetype:generate -DgroupId=org.virtage.maven.multimodule -DartifactId=par
```

Založí se složka s názvem stejným jako artifactId. V ní vymažeme složku `src/`, protože ji nebudeme potřebovat.

V top-level POM změníme

1. hodnotu v `<packaging>` z `jar` na `pom`
2. nastavíme jakékoli závilosti, pluginy ap. které mají být společné všem modulům - např. pro JUnit, změnit verzi Java kompilátoru ap.

Vytvoříme potřebný počet modulů, neboli Maven projektů uvnitř jiného Maven projektu:

```
$ mvn archetype:generate -DgroupId=org.virtage.maven.multimodule -DartifactId=r
$ mvn archetype:generate -DgroupId=org.virtage.maven.multimodule -DartifactId=w
$ mvn archetype:generate -DgroupId=org.virtage.maven.multimodule -DartifactId=.
```

Maven vytvoří složky shodné s artifactId modulů:

```
├── pom.xml
├── richclient
│   ├── pom.xml
│   └── src
│       └── ...
└── webclient
    ├── pom.xml
    └── src
        └── ...
```

Tip

V případě, že nemůžou být moduly podsložky rodičovského POM můžeme použít `<relativePath>` unvitř `<parent>` k určení relativní cesty k rodiči. Po této možnosti bysme však měli sáhnout jen v opodstatněném případě.

Nastavíme referenci na rodičovský POM pomocí elementu `<parent>`:

POM modulu

```
<project>
    ....
    <parent>
        <groupId>org.virtage.maven.multimodule</groupId>
        <artifactId>parent</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    ...
```

Ostatní elementy POMu zůstavají stejné.

> **Poznámka**
>
> Dokonce je možné, aby rodič a potomek měli různé groupId, ale není to rozhodně doporučeno.

Nyní můžeme provádět jakýkoli goal nebo phase na každém modulu zvlášť nebo v kořenovém projektu a Maven vždy provede operace ve správném pořadí dle vzájemných závislostí modulů.

3.5. Packaging (typ balení)

Packaging (a odpovídající element `<packaging>` v POM) určují typ balení neboli jaký bude výsledek sestavení Maven projektu pomocí `mvn package`. Pokud není specifikován přepokládá se `jar`.

```
<project>
    ...
    <packaging>war</packaging>
    ...
</project>
```

Zabudové packaging typy jsou:

- jar - klasický Java archiv
- war - Web ARchiv, Java EE webová aplikace

- ear - Enterprise ARchiv
- pom - typ balení kořenového Maven projektu v multi-module projektu

Typ balení je důležitý, protože určuje jaké goals se provedou při spuštění dané phase. Např. pro balení jar se na fázi package vyvolá goal jar:jar, pro balení pom je to site:attach-descriptor.

3.6. Standardní adresářová struktura

Když vytvoříme Maven projekt založí se zároveň *standardní adresářová struktura (standard folder layout)*. Tu mají všechny Maven projekty stejnou a proto nový programátor nemusí studovat, kam se ukládají jaké soubory zrovna ve vaší aplikaci.

- `pom.xml`
- `src/` – složka ze zdrojovými soubory (sources) a zdroji (resources)
  - `main/`
    - `java/` - zdrojové kódy aplikace
    - `config/` - konfigurační soubory
    - `resources/` - zdroje (ne-Java soubory jako ikony ap.)
    - `webapp/` - složka pro JSP a HTML soubory Java web aplikace
  - `test/`
    - `java/` - zdrojové kódy testů
    - `resources/` - zdroje potřebné pro testy. Zkopírují se do `target/test-classes/`.
- `target/` - výstupní složka
  - `classes/` - zkompilované třídy programu (`.class` soubory)
  - `test-classes/` - zkompilované třídy testů a testovací zdroje
  - `site/` – vygenerovaný web projektu

Poznámka

Váš projekt nemusí mít všechny uvedené složky nebo jich mít naopak více. Např. `src/main/webapp` najdeme jen v javovské webové aplikaci. `target/site/` jen, když site nastavíme, aby se generoval.

4. Nástroje a integrace

4.1. Maven a unit testy

Maven nás doslova „nutí“ do testování a psaní testů tím, že za nás vytvoří složky pro testy (`src/test/java/`) a testy spouští ještě před zabalením výsledného projektu (`mvn package`). Navíc standardně, pokud testy selžou nebo skončí chybou, projekt se nesestaví.

Jak už je u Maven obvyklé, testy provádí ve skutečnosti plugin maven-surefire-plugin. Ten podporuje JUnit i TestNG framework. Výsledkem testu je report v textové a XML podobě v `target/surefire-reports/` určený k dalšímu zpracování.

4.1.1. HTML report z výsledků

Tyto zdrojové soubory výsledků testů nejčastěji chceme zobrazit jako HTML report. K tomu slouží další plugin maven-surefire-report-plugin.

Pokud chceme tento výstup vytvořit v rámci lifecycle site (neboli `mvn site`), přidejte do `pom.xml`:

```
<reporting>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-report-plugin</artifactId>
            <version>2.16</version>   <!-- latest at the time of writing -->
        </plugin>
        ...
     </plugins>
     ...
</reporting>
```

Jen HTML výsledky testů můžete vygenerovat pomocí

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

4.1.2. Přidání/vyloučení testů

Během test phase se Surefire pokusí provést testy v `src/test/java/`. Pomocí elementů `<includes>` a `<excludes>` v `<build>` POM souboru

můžeme jednotlivé testy dodatečně přidávat nebo odebírat. Je povoleno používat zástupné znaky

- `**` – kdekoli v hiearchii složek
- `*` – libovolných nebo více znaků

Např. vyloučit všechny testy začínající na `Dummy`:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</group>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.16</version>
            <configuration>
                <excludes>
                    <exclude>**/Dummy*.java</exclude>
                </excludes>
            </configuration>
        </plugin>
        ...
    </plugins>
    ...
</build>
```

Důležité

Pozor na to, že Maven surefire plugin defaultně spouští jen testy (třídy testů), které se začínají na `Test`, končí na `Test` nebo `TestCase`! Nemusíme nic dělat, pokud dodržujeme konvenci a třídy pojmenováváme `NěcoTest`.

Pokud chceme testovat všechny Java soubory v `src/test/java/` bez ohledu na jejich název můžeme změnit nastavení surefire pluginu:

```
<configuration>
    <!-- Run Java classes of any name, not only *Test, Test* or *TestCase
    <includes>
        <include>**/*.java</include>
    </includes>
</configuration>
```

4.1.3. Přeskočení testů

Použijte volbu `-Dmaven.test.skip=true`, např.:

mvn clean package -Dmaven.test.skip=true

4.1.4. Spuštění jen jednoho testu

Pro spuštění jen jediného testu použijte:

```
mvn -Dtest=TridaTestu test
```

4.2. Maven a Jenkins CI

Maven tvoří s CI serverem Jenkins skvělý pár. Díky tomu, že Maven sjednocuje build jakéhokoli projektu na několik standardizovaných fází může Jenkins snáze nabídnout podporu Maven projektů. Maven plugin je navíc již dodáván přímo s Jenkins.

Postup není složitý:

1. Stáhneme a spustíme Jenkins z WAR souboru:

   ```
   $ java -jar jenkins.war
   ```

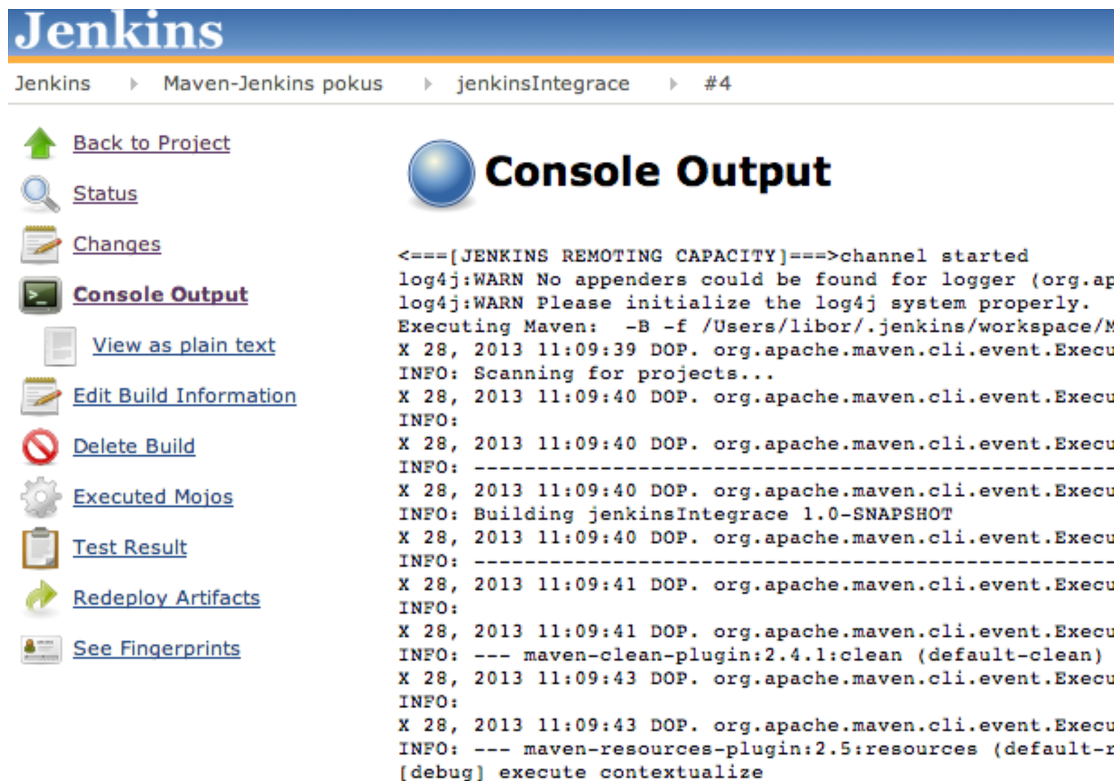
   nebo nainstalujeme balíček pro svůj OS.

2. Běžne na http://localhost:8080/.

3. V Manage Jenkins ‣ Configure System ‣ Maven Installations nastavíme cestu k existující instalaci Maven nebo využijeme schopnost Jenkins - stáhnout si Maven automaticky.

Nastavení cesty k instalaci Maven v Jenkins CI

4. Založíme nový projekt typu Maven 2/3 běžným způsobem.
5. Funkčnost si zkontrolujeme v konzolovém výstupu jobu.



Příklad výstupu sestavování Maven projektu v Jenkins CI

5. Tipy a triky

5.1. Nastavení verze Java kompilátoru

Přidáme do sekce `<plugins>` POM souboru:

```
...
<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
...
```

5.2. Nastavení výchozího goal nebo phase

Voláme stále Maven se stejným cílem nebo fází? Např.

```
$ mvn clean package
```

Díky nastavení defaultního cíle nebo fáze (od Maven 2) můžeme psát jen

```
$ mvn
```

pro totéž. V `pom.xml` v `<defaultGoal>` elementu musíme nastavit nejčastější goal nebo fázi:

```
<build>
    ...
    <defaultGoal>clean package</defaultGoal>
    ...
```

5.3. Varování „Using platform encoding, build is platform dependent“

Pokud není explicitně určeno v jakém kódování jsou zdrojové soubory, Maven nás varuje hláškou podobnou této:

```
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources,
i.e. build is platform dependent!
```

Znamená to, že použije kódování aktuálního počítače, ale to nemusí být to, ve kterém byly soubory vytvořeny a tak může na jiném PC build selhat.

Je třeba nastavit kódování zdrojových souborů vlastnost `project.build.sourceEncoding`:

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</propeties>
```

5.4. Spuštění Java programu z Maven

Sám Maven neumí spustit Java program (resp. třídu se `main()` metodou-). Pokud to potřebujeme poslouží exec-maven-plugin:

```
mvn exec:java -Dexec.mainClass=org.virtage.maven.App
```

5.5. Získávání nápovědy

Maven obsahuje příkazy (resp. goaly) pluginu help, který nám pomůže zjistit řadu důležitých informací pro práci.

5.5.1. Zjištění goals určitého pluginu

Jaké goaly plugin nabízí k použití zjistíme příkazem

```
$ mvn help:describe -Dplugin=<plugin-name>
```

Příklad výpisu pro plugin jar (zkráceno):

```
$ mvn help:describe -Dplugin=jar
[INFO] org.apache.maven.plugins:maven-jar-plugin:2.4

Name: Maven JAR Plugin
Description: Builds a Java Archive (JAR) file from the compiled project
classes and resources.
Group Id: org.apache.maven.plugins
Artifact Id: maven-jar-plugin
Version: 2.4
Goal Prefix: jar

This plugin has 5 goals:

jar:help
Description: Display help information on maven-jar-plugin.
    Call
    mvn jar:help -Ddetail=true -Dgoal=<goal-name>
    to display parameter details.

jar:jar
Description: Build a JAR from the current project.

jar:sign
...

...
```

5.5.2. Zjištění výsledného POM

Výsledný POM vzniklý dědičností POMů zjistíme pomocí

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

5.5.3. Zjištění aktivních profilů

Vypíše všechny profily aktivované manuálně i automaticky.

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

5.5.4. Závislosti

Závilosti jsou často problematické. Různé verze, závilosti závilostí atd. Maven přichází na pomoc s řadou goalů pro analýzu závilostí.

Vypíše strom (hierarchii) závislostí:

```
mvn dependency:tree
```

Vypíše závislosti v abecedním pořadí:

```
mvn dependency:resolve
```

Analýza závilostí, vypíše všechny nepoužitené a nedeklarované závilosti:

```
mvn dependency:analyze
```

5.6. Debugging (ladění) Maven

Maven nabízí řadu možností, co dělat při problémech.

> **Poznámka**
>
> Zabudovanou nápovědu k pluginům, analýzu závilostí ap. najdeme na stránce Získávání nápovědy.

5.6.1. Full stack trace výjimek (exceptions)

Pokud Maven plugin nebo Maven samotný skončí výjimkou, můžeme vynutit full stack trace volbou `-e`, např.:

```
mvn clean package -e
```

5.6.2. Vypisovat debug info

Volbou -X nebo -debug přinutíme Maven vypisovat všechny detaily toho, co provádí. Pozor, výpis bude velmi dlouhý!

```
mvn <goal> -X
```

5.6.3. Debug Maven nebo pluginů

Velmi pokročilý způsob ladění Maven představuje možnost krokovat provádění pomocí JPDA debuggeru (např. z vašeho IDE jako IntelliJ IDEA).

Místo příkazu `mvn` použijeme `mvnDebug`. Ve výchozí konfiguraci bude Maven čekat na připojení debuggeru na portu 8000:

```
$ /usr/share/maven/bin/mvnDebug
Preparing to Execute Maven in Debug Mode
Listening for transport dt_socket at address: 8000
...
```