eBook Gratuit

APPRENEZ Apache Maven

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#maven

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: apache-maven

It is an unofficial and free Apache Maven ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Apache Maven.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec Apache Maven

Remarques

Comme décrit dans son guide de démarrage officiel :

> Maven est une tentative d'appliquer des **modèles** à l'infrastructure de construction d'un projet afin de promouvoir la compréhension et la productivité en fournissant un chemin clair dans l'utilisation des **meilleures pratiques** .

Maven est essentiellement un outil de gestion et de compréhension de projets et fournit ainsi un moyen d'aider à gérer:

- Construit
- Documentation
- Rapport
- Les dépendances
- Contrôle de version
- Les rejets
- Distribution

D'où la prise en charge des développeurs dans de nombreuses phases de l'ensemble du cycle de vie du développement logiciel (SDLC).

Cette philosophie fait partie intégrante de Maven: le mot *maven* signifie *accumulateur de connaissances* (en yiddish).

Maven concerne l'application de **modèles** afin de créer une infrastructure présentant les caractéristiques de visibilité, de réutilisabilité, de maintenabilité et de compréhensibilité.

- Maven est né de la volonté très concrète de faire fonctionner plusieurs projets de la même manière, comme l'affirme la déclaration de philosophie officielle de Maven .
- Les développeurs pouvaient se déplacer librement entre les projets, en sachant clairement comment ils travaillaient tous en comprenant comment fonctionnait l'un d'eux.
- La même idée s'étend aux tests, à la génération de documentation, à la génération de métriques et de rapports et au déploiement

Versions

Version	Annoncer	Commentaire	Date de sortie
1.0-beta-2	annoncer	Première version (bêta)	2002-03-30
1.0	annoncer	Première sortie officielle	2004-07-13
2.0	annoncer	**Version** officielle **2.0**	2005-10-20

Version	Annoncer	Commentaire	Date de sortie
3.0	annoncer	**Version** officielle **3.0**	2010-10-08

Examples

Installation ou configuration

Les versions binaires de Maven peuvent être téléchargées sur le site Web de Maven .

Le binaire se présente sous la forme d'une archive zip ou d'une archive tar.gz. Après le téléchargement, les instructions de la page d'installation peuvent être suivies:

- Assurez-vous que la variable d'environnement `JAVA_HOME` est définie et pointe vers votre installation JDK (pas JRE). Par exemple, sur un ordinateur Windows, ce dossier d'installation peut correspondre à `C:\Program Files\Java\jdk1.8.0_51` .
- Extrayez l'archive de distribution dans le répertoire de votre choix.
- Ajoutez le répertoire `bin` répertoire créé (nommé `apache-maven-3.3.9` pour Maven 3.3.9) à la variable d'environnement `PATH` . (Référence pour le changer sous Windows).
- Vérifiez que la configuration est correcte en exécutant `mvn -version` sur la ligne de commande.

Il n'est pas nécessaire de définir la variable d'environnement `M2_HOME` ou `MAVEN_HOME` .

Installation sur Ubuntu

1. Dans un terminal, lancez `sudo apt-get install maven`

2. Une fois l'installation terminée, vérifiez qu'elle fonctionne correctement avec `mvn -v` La sortie devrait ressembler à `mvn -v` :

```
Apache Maven 3.3.9
Maven home: /usr/share/maven
Java version: 1.8.0_121, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-8-openjdk-amd64/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.8.0-parrot-amd64", arch: "amd64", family: "unix"
```

Si cela ne fonctionne pas, assurez - vous que vous avez un JDK installé dans votre environnement `javac -version`

Configuration des paramètres de proxy

Si votre connexion Internet est fournie via un proxy, Maven ne sera pas en mesure de télécharger des fichiers JAR à partir de référentiels distants, un problème courant pour les entreprises.

Pour résoudre ce problème, Maven doit disposer des détails et des informations d'identification du proxy en accédant à *{Maven install location}* → *confi* → `settings.xml` . Faites défiler jusqu'à la

`<proxies>` et entrez les détails ici, en utilisant le format mentionné dans les commentaires.

Pour les utilisateurs d'Eclipse

Eclipse utilise son propre fichier `settings.xml` pour exécuter Maven, dont l'emplacement peut être trouvé dans le menu *Fenêtre → Préférences → Maven → Paramètres utilisateur → Paramètres utilisateur:.* Si le fichier n'est pas disponible à l'emplacement mentionné, créez-le simplement ou créez un duplicata du fichier à l'emplacement ci-dessus *{emplacement d'installation Maven} → confi →* `settings.xml` .

Pour les utilisateurs d'IntelliJ

Ouvrez les paramètres et accédez à Maven -> Importer. (Cela peut être imbriqué sous Build, Execution, Deployment -> Build Tools ->, selon la version d'IntelliJ que vous utilisez.)

Définissez le champ nommé "Options de la machine virtuelle pour l'importateur" comme suit:

```
-DproxySet=true -DproxyHost=<HOST> -DproxyPort=<PORT>
-DproxySet=true -DproxyHost=myproxy.com -DproxyPort=8080
```

Appliquez et redémarrez IntelliJ.

Installation sur Mac OSX avec Brew

1. Dans un terminal, exécuter le `brew install maven`
2. Une fois l'installation terminée, vérifiez que maven fonctionne correctement avec `mvn -v` . La sortie devrait ressembler à quelque chose comme:

```
Apache Maven 3.3.9
Maven home: /usr/local/Cellar/maven/3.3.9/libexec
Java version: 1.8.0_121, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.12.4", arch: "x86_64", family: "mac"
```

Si cela ne fonctionne pas, assurez - vous que vous avez un JDK installé dans votre environnement `javac -version`

Lire Démarrer avec Apache Maven en ligne: https://riptutorial.com/fr/maven/topic/898/demarrer-avec-apache-maven

Chapitre 2: Accédez aux informations Maven dans le code

Introduction

Il est parfois utile d'obtenir les propriétés Maven, telles que la version actuelle, dans le code. Voici quelques moyens d'y parvenir.

Examples

Obtenir le numéro de version depuis un pot

Si vous empaquetez votre application dans un `jar` à l'aide du `maven-jar-plugin` ou du `maven-jar-plugin` `maven-assembly-plugin` , un moyen simple d'obtenir la version actuelle de pom consiste à ajouter une entrée dans le manifeste, qui est alors disponible à partir de Java.

Le secret est de définir l'indicateur `addDefaultImplementationEntries` sur true (et l' `addDefaultSpecificationEntries` vous avez également besoin de l'ID d'artefact).

Configuration du plugin jar :

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <configuration>
        <archive>
          <manifest>
            <mainClass>...</mainClass>
            <addDefaultImplementationEntries>
              true
            </addDefaultImplementationEntries>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Configuration du plugin d'assemblage :

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
```

```
        <archive>
            <manifest>
              <addDefaultImplementationEntries>true</addDefaultImplementationEntries>
            </manifest>
        </archive>
    </configuration>
    <executions>
        <execution .../>
    </executions>
</plugin>
```

`addDefaultImplementationEntries` demande à Maven d'ajouter les en-têtes suivants au `MANIFEST.MF` de votre `MANIFEST.MF` jar:

```
Implementation-Title: display-version
Implementation-Version: 1.0-SNAPSHOT
Implementation-Vendor-Id: test
```

Vous pouvez maintenant utiliser cette ligne de code n'importe où dans votre fichier jar pour accéder au numéro de version:

```
getClass().getPackage().getImplementationVersion()
```

Plus d'informations ici et ici .

Garder un fichier de propriétés en synchronisation en utilisant le mécanisme de filtrage des propriétés de maven

Comme l'explique cette documentation ,

> Parfois, un fichier de ressources doit contenir une valeur qui ne peut être fournie qu'au moment de la construction. Pour ce faire, placez une référence à la propriété qui contiendra la valeur dans votre fichier de ressources en utilisant la syntaxe `${<property name>}` . La propriété peut être l'une des valeurs définies dans votre `pom.xml` , une valeur définie dans le fichier `settings.xml` l'utilisateur, une propriété définie dans un fichier de propriétés externe ou une propriété système.

A titre d'exemple, créons un simple `info.txt` dans `src/main/resources` contenant la version de pom et le temps de compilation.

1. créer un `src/main/resources/info.txt` avec le contenu suivant:

   version = $ {pom.version} build.date = $ {horodatage}

2. demander à Maven d' *étendre* les propriétés en définissant le `filtering` sur true:

   ```
   <build>
       <resources>
           <resource>
              <directory>src/main/resources</directory>
              <filtering>true</filtering>
   ```

```
        </resource>
    </resources>
</build>
```

3. avec cela, la version sera mise à jour, mais malheureusement un bogue dans Maven empêche la propriété `${maven.build.timestamp}` d'être transmise au mécanisme de filtrage des ressources (plus d'informations ici). Alors, créons une propriété d' `timestamp` comme solution de contournement! Ajoutez ce qui suit aux propriétés de pom:

```
<properties>
    <timestamp>${maven.build.timestamp}</timestamp>
    <maven.build.timestamp.format>yyyy-MM-dd'T'HH:mm</maven.build.timestamp.format>
</properties>
```

4. lancez maven, vous devriez trouver un `info.txt` dans `target/classes` avec un contenu comme:

```
version=0.3.2
build.date=2017-04-20T13:56
```

Lecture d'un fichier pom.xml à l'exécution à l'aide du plugin maven-model

Les autres exemples peuvent être le meilleur moyen et le plus stable d'obtenir un numéro de version dans une application de ***manière statique*** . Cette réponse propose une alternative montrant comment le faire ***dynamiquement*** pendant l'exécution, en utilisant la bibliothèque maven *maven-model* .

Ajoutez la dépendance:

```
<dependency>
  <groupId>org.apache.maven</groupId>
  <artifactId>maven-model</artifactId>
  <version>3.3.9</version>
</dependency>
```

En Java, créez un `MavenXpp3Reader` pour lire votre pom. Par exemple:

```
package de.scrum_master.app;

import org.apache.maven.model.Model;
import org.apache.maven.model.io.xpp3.MavenXpp3Reader;
import org.codehaus.plexus.util.xml.pull.XmlPullParserException;

import java.io.FileReader;
import java.io.IOException;

public class MavenModelExample {
    public static void main(String[] args) throws IOException, XmlPullParserException {
        MavenXpp3Reader reader = new MavenXpp3Reader();
        Model model = reader.read(new FileReader("pom.xml"));
        System.out.println(model.getId());
        System.out.println(model.getGroupId());
```

```
        System.out.println(model.getArtifactId());
        System.out.println(model.getVersion());
    }
}
```

Le journal de la console est le suivant:

```
de.scrum-master.stackoverflow:my-artifact:jar:1.0-SNAPSHOT
de.scrum-master.stackoverflow
my-artifact
1.0-SNAPSHOT
```

Lire Accédez aux informations Maven dans le code en ligne:
https://riptutorial.com/fr/maven/topic/9773/accedez-aux-informations-maven-dans-le-code

Chapitre 3: Créer un plugin Maven

Introduction

Maven vous permet d'implémenter et d'utiliser des plugins personnalisés. Ces plug-ins permettent à un comportement supplémentaire d'être lié à n'importe quelle phase du cycle de vie de Maven.

Chaque objectif Maven est créé en implémentant un objet Java MOJO (Maven Ordinary Java Object): une classe Java implémentée avec des annotations qui décrivent comment l'invoquer.

Le préfixe d'objectif d'un plugin est dérivé de son nom d'artefact. Un `hello-world-plugin` artefact `hello-world-plugin` crée un préfixe d'objectif `hello-world` . Le but `hello` peut alors être exécuté avec `mvn hello-world:hello` .

Remarques

Un plugin Maven est un JAR contenant un `maven/plugins.xml` qui décrit les métadonnées du plugin. Ce fichier est généré par le `maven-plugin-plugin` .

Examples

Déclarer un artefact Maven en tant que plugin Maven

Un artefact construit par Maven peut être déclaré en tant que plug-in Maven en spécifiant le `maven-plugin` comme `maven-plugin` dans le `pom.xml` .

```
<packaging>maven-plugin</packaging>
```

Vous devez déclarer une dépendance à l'API du plug-in et aux annotations.

```
<dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-plugin-api</artifactId>
    <version>3.3.9</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.maven.plugin-tools</groupId>
    <artifactId>maven-plugin-annotations</artifactId>
    <version>3.5</version>
    <scope>provided</scope>
</dependency>
```

Vous devez ajouter un plug-in pour générer les métadonnées.

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

```
    <artifactId>maven-plugin-plugin</artifactId>
    <version>3.5</version>
</plugin>
```

Créer un objectif

Les objectifs sont mis en œuvre en créant un MOJO. Ceci est un fichier de classe annoté avec des annotations de `maven-plugin-annotations` .

```
@Mojo(name = "hello")
public final class HelloWorldMojo extends AbstractMojo {

    public void execute() throws MojoExecutionException, MojoFailureException {
         getLog().info("Hello world");
    }
}
```

Utiliser la configuration du plugin

Les plugins peuvent être configurés en annotant des champs avec `@Parameter` . Le MOJO est ensuite injecté avec la configuration.

```
@Mojo(name = "greet")
public final class GreetMojo extends AbstractMojo {

    @Parameter(required = true)
    public String name;

    public void execute() throws MojoExecutionException, MojoFailureException {
        getLog().info("Hello " + name);
    }
}
```

Le paramètre `name` peut être configuré dans le POM:

```
<plugin>
    <groupId>com.mattunderscore</groupId>
    <artifactId>hello-world-plugin</artifactId>
    <version>1.0-SNAPSHOT</version>
    <configuration>
        <name>Matt</name>
    </configuration>
</plugin>
```

Si l'objectif `greet` est exécuté en tant qu'objectif autonome, le paramètre `name` peut être défini en tant que propriété sur la ligne de commande:

```
mvn <plugin name>:greet -Dname=Geri
```

Accéder aux informations du projet

Le plug-in peut, entre autres, accéder à des informations sur le projet Maven en cours de

construction.

```
@Mojo(name = "project")
public final class ProjectNameMojo extends AbstractMojo {

    @Parameter(defaultValue = "${project}", readonly = true, required = true)
    private MavenProject project;

    public void execute() throws MojoExecutionException, MojoFailureException {
        getLog().info("Hello, this is " + project.getName());
    }
}
```

L'exemple ci-dessus imprimera dans la console le nom du projet Maven sur lequel il est exécuté, qui est spécifié dans l'élément `<project>/<name>` de son POM.

La classe `MavenProject` utilisée dans le plugin nécessite une dépendance de `maven-core` avec une étendue de `compile` (par défaut) dans le POM du plugin:

```
<dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-core</artifactId>
    <version>3.3.9</version>
</dependency>
```

De plus, l' utilisation des annotations nécessite la dépendance suivante dans le POM du plugin:

```
<dependency>
  <groupId>org.apache.maven.plugin-tools</groupId>
  <artifactId>maven-plugin-annotations</artifactId>
  <version>3.5</version>
  <scope>provided</scope> <!-- annotations are needed only to build the plugin -->
</dependency>
```

Déclarez une phase par défaut pour un objectif

```
@Mojo(name = "hi", defaultPhase = LifecyclePhase.COMPILE)
```

Récupère le répertoire de construction en tant que fichier

```
@Parameter(defaultValue = "${project.build.directory}")
private File buildDirectory;
```

Lire Créer un plugin Maven en ligne: https://riptutorial.com/fr/maven/topic/8635/creer-un-plugin-maven

Chapitre 4: Cycle de construction Maven

Introduction

Vous trouverez ci-dessous une liste complète des phases de cycle de vie de construction par défaut de Maven. Chacune de ces phases est appelée en l'ajoutant à la commande `mvn` , par exemple `mvn install` .

Examples

Maven construit des phases de cycle de vie

```
validate
```

Valide si le projet est correct et si toutes les informations requises sont disponibles pour la génération.

```
initialize
```

Initialise l'environnement de génération, par exemple, définit les propriétés ou crée des répertoires.

```
generate-sources
```

Génère le code source à traiter dans la phase de compilation.

```
process-sources
```

Traite le code source au cas où un filtre doit être appliqué.

```
generate-resources
```

Génère des ressources à inclure dans l'artefact.

```
process-resources
```

Traite et copie les ressources dans le répertoire de sortie (`${basedir}/target/classes`).

```
compile
```

Compile le code source du projet dans le répertoire source (`${basedir}/src/main/[java|groovy|...]`) dans le répertoire de sortie (`${basedir}/target/classes`).

```
process-classes
```

Traite les fichiers `.class` générés lors de la phase de `compile` , par exemple pour effectuer des améliorations de code octet.

```
generate-test-sources
```

Génère le code source du test à traiter dans la phase de `test-compile` .

```
process-test-sources
```

Traite le code source du test au cas où un filtre doit être appliqué.

```
generate-test-resources
```

Génère des ressources pour les tests.

```
process-test-resources
```

Traite et copie les ressources de test du répertoire de ressources (`${basedir}/src/main/resources`) dans le répertoire de sortie de test (`${basedir}/target/test-classes`).

```
test-compile
```

Compile le code source dans le répertoire source du test ('$ {basedir} / src / test / [java | groovy | ...]') dans le répertoire de sortie de test (`${basedir}/target/test-classes`).

```
process-test-classes
```

Traite les fichiers de test `.class` générés lors de la phase de `test-compile` , par exemple pour effectuer des améliorations du bytecode (Maven 2.0.5 et supérieur).

```
test
```

Exécute des tests en utilisant un cadre de test approprié. Remarque: ces cas de test ne sont pas pris en compte pour le conditionnement et le déploiement.

```
prepare-package
```

Effectue les modifications finales et les validations avant la création définitive du package.

```
package
```

Emballe le code compilé et testé avec succès dans un format distribuable tel que JAR, WAR, EAR dans le répertoire cible (`${basedir}/target`).

```
pre-integration-test
```

Effectue des actions avant que les tests d'intégration ne soient exécutés s'ils nécessitent d'appliquer des modifications dans l'environnement pour l'application.

```
integration-test
```

Traite et déploie éventuellement l'application dans un environnement où les tests d'intégration peuvent être exécutés.

```
post-integration-test
```

Effectue des actions après les tests d'intégration, comme le nettoyage de l'environnement créé lors de la phase de `pre-integration-test` .

```
verify
```

Vérifie si un paquet est valide et répond aux critères de qualité requis.

```
install
```

Installe l'artefact dans le référentiel local. Tout autre projet local peut utiliser cet artefact comme l'une de ses dépendances (si de toute façon votre IDE ne prend pas en *charge la résolution de dépendance de l'espace de travail*).

```
deploy
```

Copie le package dans un référentiel distant pour le rendre disponible pour les autres développeurs.

Lire Cycle de construction Maven en ligne: https://riptutorial.com/fr/maven/topic/9679/cycle-de-construction-maven

Chapitre 5: Effectuer une libération

Introduction

Le plugin Maven standard utilisé par un Release Process est le plugin maven-release - la configuration de ce plugin est minimale:

SCM dans le pom de Maven: Le processus de publication interagira avec le contrôle de code source du projet - cela signifie que nous devons définir l'élément "scm" dans notre fichier pom.xml. L'élément "scm" pour une version release doit contenir suffisamment d'informations Découvrez l'étiquette qui a été créée pour cette version.

Remarques

Remarque: veillez à utiliser le plug-in 2.5 ou une version ultérieure de maven pour éviter les problèmes liés au mode Maven. Le processus de libération

```
mvn release:clean
```

La commande ci-dessus effectuera les opérations suivantes: Supprimez le descripteur de version (release.properties) supprimez tous les fichiers POM de sauvegarde.

```
mvn release:prepare
```

La prochaine partie du processus de publication est la préparation de la version; Cela va: effectuer des vérifications - il ne doit y avoir aucune modification non validée et le projet ne doit dépendre d'aucune dépendance SNAPSHOT modifier la version du projet dans le fichier pom en un numéro de version complet (supprimer le suffixe SNAPSHOT) - exécuter les suites de test du projet valider et pousser les modifications créer la balise hors de ce code non versionné SNAPSHOT augmenter la version du projet dans le pom - dans notre exemple - 0.0.2-SNAPSHOT valider et pousser les modifications

```
mvn release:perform
```

La dernière partie du processus de publication est la réalisation de la version; Cela va: extraire le code de la version de SCM et déployer le code publié Cette deuxième étape du processus repose sur la sortie de l'étape Prepare - the release.properties.

Examples

POM.xml pour effectuer la publication dans le référentiel Nexus

```
<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.codezarvis.artifactory</groupId>
<artifactId>nexusrelease</artifactId>
<version>0.0.5-SNAPSHOT</version>
<packaging>jar</packaging>

<name>nexusrelease</name>
<url>http://maven.apache.org</url>

<scm>
<connection>scm:git:git@github.com:isudarshan/nexuspractice.git</connection>
<url>scm:git:git@github.com:isudarshan/nexuspractice.git</url>
<developerConnection>scm:git:git@github.com:isudarshan/nexuspractice.git</developerConnection>
<tag>HEAD</tag>
</scm>

<distributionManagement>
<!-- Publish the versioned snapshot here -->
<repository>
<id>codezarvis</id>
<name>codezarvis-nexus</name>
<url>http://localhost:8080/nexus/content/repositories/releases</url>
</repository>

<!-- Publish the versioned releases here -->
<snapshotRepository>
<id>codezarvis</id>
<name>codezarvis-nexus</name>
<url>http://localhost:8080/nexus/content/repositories/snapshots</url>
</snapshotRepository>
</distributionManagement>

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

<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>

<build>
<pluginManagement>
<plugins>
<plugin>
<artifactId>maven-release-plugin</artifactId>
<version>2.5.2</version>
<executions>
<execution>
<id>default</id>
<goals>
<goal>perform</goal>
</goals>
<configuration>
```

```
<pomFileName>${project.name}/pom.xml</pomFileName>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</pluginManagement>
</build>
</project>
```

Lire Effectuer une libération en ligne: https://riptutorial.com/fr/maven/topic/9680/effectuer-une-liberation

Chapitre 6: Générer des rapports FIXME / TODO à l'aide du taglist-maven-plugin

Introduction

Ceci est un petit code (xml) extrait pour mettre en évidence l'utilisation du taglist-maven-plugin pour générer des rapports personnalisés (de TODO, FIXME, ...)

Examples

pom.xml pour générer un rapport FIXME

```
<build>
    <plugins>
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>project-info-reports</artifactId>
                <version>2.9</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>index</goal>
                        </goals>
                        <phase>site</phase>
                    </execution>
                </executions>
            </plugin>
    </plugins>
</build>

<reporting>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-project-info-reports-plugin</artifactId>
                <version>2.9</version>
                <reportSets>
                    <reportSet>
                    <reports>
                        <report>index</report>
                        <report>issue-tracking</report>
                    </reports>
                    </reportSet>
                </reportSets>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>taglist-maven-plugin</artifactId>
                <version>2.4</version>
                <configuration>
                    <tagListOptions>
                        <tagClasse>
                            <displayName>FIXME Work</displayName>
```

```
                        <tags>
                            <tag>
                                <matchString>FIXME</matchString>
                                <matchType>ignoreCase</matchType>
                            </tag>
                            <tag>
                                <matchString>@fixme</matchString>
                                <matchType>ignoreCase</matchType>
                            </tag>
                        </tags>
                    </tagClasse>
                    <tagClasse>
                        <displayName>TODO Work</displayName>
                        <tags>
                            <tag>
                                <matchString>TODO</matchString>
                                <matchType>ignoreCase</matchType>
                            </tag>
                            <tag>
                                <matchString>@todo</matchString>
                                <matchType>ignoreCase</matchType>
                            </tag>
                        </tags>
                    </tagClasse>
                </tagListOptions>
            </configuration>
        </plugin>
    </plugins>
</reporting>
```

Puis courir

```
mvn clean site:site
```

Lire Générer des rapports FIXME / TODO à l'aide du taglist-maven-plugin en ligne: https://riptutorial.com/fr/maven/topic/10110/generer-des-rapports-fixme---todo-a-l-aide-du-taglist-maven-plugin

Chapitre 7: Intégration Eclipse

Examples

Installer Maven dans Eclipse

Vous pouvez tirer parti des puissantes fonctionnalités d'Apache Maven dans Eclipse en installant la fonctionnalité M2Eclipse . Suivez ces étapes pour installer Maven dans Eclipse:

1. Ouvrez Eclipse et sélectionnez *Aide* → *Installer un nouveau logiciel…*
2. Dans la boîte de dialogue ouverte, sélectionnez le bouton `Ajouter ...` pour ajouter un nouveau référentiel.
3. Remplissez le formulaire avec les informations ci-dessous et confirmez avec `OK` :

   Nom: `M2Eclipse`

   Lieu: `http://download.eclipse.org/technology/m2e/releases`
4. Une fois la mise en *attente* terminée, `sélectionnez Tout` et sélectionnez `Suivant` .
5. ⊙ `acceptez les termes du contrat de licence et sélectionnez Terminer`.
6. A la fin de l'installation, il vous sera demandé de redémarrer Eclipse. Sélectionnez `Oui` pour effectuer le redémarrage.

Vérifiez si le support de M2Eclipse Maven est déjà installé sur Eclipse

Allez dans *Aide* → *À propos d'Eclipse* → Vérifiez si la *fonctionnalité m2e* est présente: .

Configurer une installation Maven personnalisée dans Eclipse

Eclipse fournirait son propre environnement Maven intégré prêt à l'emploi, ce qui n'est pas recommandé lorsqu'une certaine version de Maven doit être utilisée ou que d'autres configurations doivent être effectuées (proxy, miroirs, etc.): pour un contrôle total sur quel environnement Maven serait utilisé par l'EDI.

- Sélectionnez *Fenêtre* → *Préférences* → *Maven* → *Installations*
- Sélectionnez `Ajouter ..` pour ajouter une installation Maven personnalisée / locale
- Fournissez les informations nécessaires et sélectionnez `Terminer` :

  Installation à domicile: `... your Maven home ... Annuaire ...`

  Nom d'installation: `Apache Maven xyz`

- It sélectionnez-le par défaut (au lieu de la version *EMBEDDED* par défaut) et confirmez avec `OK` .

Lire Intégration Eclipse en ligne: https://riptutorial.com/fr/maven/topic/2315/integration-eclipse

Chapitre 8: Maven installer dans la fenêtre

Introduction

comment installer maven dans la fenêtre 7

Remarques

comment installer maven dans la fenêtre 7 étapes:

1. téléchargez le formulaire maven https://maven.apache.org/download.cgi (site Web poffice) 2.zipez le dossier binaire maven et enregistrez-le dans n'importe quel floder (bon: enregistrez-le dans les fichiers du programme dans le lecteur c)
2. Vérifiez la valeur de la variable d'environnement invite de commande ouverte et tapez echo% java_home% son doit afficher le chemin du fichier jdk comme: C: \ Program Files \ Java \ jdk1.8.0_102 si non affiché définissez la variable d'environnement java_home

Définissez les variables d'environnement à l'aide du système m2_home: définissez le chemin du dossier où son maven_home stocké: idem que ci-dessus définissez le chemin Ajout à PATH: Ajoutez le répertoire bin de la distribution décompressée à votre variable d'environnement PATH utilisateur:% m2_home% \ bin

Pour vérifier, ouvrez l'invite de commande et tapez mvn -version doit afficher ce message Apache Maven 3.3.3 Version Java: 1.8.0_45, fournisseur: Oracle Corporation Java x ", version:" 10.8.5 ", arch:" x86_64 ", famille:" mac

un certain temps, il ne s'affichera pas, car le dossier mvn ne fonctionnant pas avec l'accès administrateur rend l'exécution en tant qu'administrateur

Examples

installer

Vérifiez la valeur de la variable d'environnement, par exemple echo% JAVA_HOME% C: \ Program Files \ Java \ jdk1.7.0_51

Lire Maven installer dans la fenêtre en ligne: https://riptutorial.com/fr/maven/topic/10813/maven-installer-dans-la-fenetre

Chapitre 9: Maven Tomcat Plugin

Examples

Démarrez tomcat en utilisant le plugin maven.

Dans l'exemple, nous lancerons tomcat 7 en utilisant le plugin maven, éventuellement en ajoutant une protection utilisateur / mot de passe pour le point de fin REST. Ajoutant également la caractéristique de la guerre de construction.

Ajouter ci-dessous la section dans la section plugin de pom pour tomcat

```
             <plugin>
                <groupId>org.apache.tomcat.maven</groupId>
                <artifactId>tomcat7-maven-plugin</artifactId>
                <version>2.2</version>
                <configuration>
                    <url>http://localhost:8090/manager</url>
                    <server>localhost</server>
                    <port>8191</port>
                    <path>/${project.build.finalName}</path>
                    <tomcatUsers>src/main/tomcatconf/tomcat-users.xml</tomcatUsers>
                </configuration>
            </plugin>
```

Vérifiez que le plugin maven war est ajouté et que web.xml est présent à l'emplacement /src/main/webapp/WEB-INF/web.xml. Ci-dessous, un exemple de plugin de guerre.

```
<plugin>
            <artifactId>maven-war-plugin</artifactId>
            <version>2.3</version>
            </plugin>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                    <webResources>
                        <resource>
                            <!-- this is relative to the pom.xml directory -->
                            <directory>/src/main/webapp/WEB-INF/web.xml</directory>
                        </resource>
                    </webResources>
                </configuration>
            </plugin>
```

Si vous le souhaitez, ajoutez tomcat-users.xml à l'emplacement src / main / tomcatconf. Il sera copié automatiquement lorsque tomcat démarrera.

```
<tomcat-users>
    <user name="user" password="password" roles="admin" />
```

```
</tomcat-users>
```

Si vous le souhaitez, ajoutez l'entrée ci-dessous dans web.xml pour protéger l'URL REST.

```
    <!-- tomcat user -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Wildcard means whole app requires authentication</web-resource-
name>
            <url-pattern>/helloworld/*</url-pattern>
            <http-method>GET</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>admin</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>NONE</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
    </login-config>
```

Créez un nouveau build de maven à partir d'eclipse. Sélectionnez le projet de guerre et dans la section Objectifs, ajoutez la commande ci-dessous.

```
tomcat7:run
```

vous verrez un message.

[INFO] --- tomcat7-maven-plugin: 2.2: exécuter (default-cli) @ web-service-ldap2 --- [INFO] Exécuter la guerre sur http: // localhost: 8191 /

Lire Maven Tomcat Plugin en ligne: https://riptutorial.com/fr/maven/topic/6292/maven-tomcat-plugin

Chapitre 10: Plugin de montage Maven

Examples

Créer un fichier .jar avec toutes les dépendances du projet

Pour créer un fichier JAR contenant toutes ses dépendances, il est possible d'utiliser les `jar-with-dependencies` au format descripteur intégré. L'exemple suivant configure une exécution du plug-in Assembly lié à la phase du `package` , en utilisant ce descripteur intégré et en déclarant une classe principale de `com.example` :

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.6</version>
  <executions>
    <execution>
      <id>make-assembly</id>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <archive>
          <manifest>
              <mainClass>com.example</mainClass>
          </manifest>
        </archive>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Fonctionnement:

```
mvn clean package
```

sur la ligne de commande, les jar-with-dependencies seront créés et attachés au projet.

Si vous avez besoin de plus de contrôle sur cet uber-jar, rendez- vous au plug-in Maven Shade .

Lire Plugin de montage Maven en ligne: https://riptutorial.com/fr/maven/topic/2308/plugin-de-montage-maven

Chapitre 11: Plugin Maven EAR

Introduction

Voici un exemple de configuration pour un plugin maven ear de base pour empaqueter les artefacts .war et .jar

Examples

Une configuration EAR de base

```
<dependencies>
    <dependency>
        <groupId>{ejbModuleGroupId}</groupId>
        <artifactId>{ejbModuleArtifactId}</artifactId>
        <version>{ejbModuleVersion}</version>
        <type>ejb</type>
    </dependency>
    <dependency>
        <groupId>{webModuleGroupId}</groupId>
        <artifactId>{webModuleArtifactId}</artifactId>
        <version>{webModuleVersion}</version>
        <type>war</type>
    </dependency>
</depencencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-ear-plugin</artifactId>
                <version>2.9.1</version>
                <configuration>
                    <version>1.4</version><!-- application.xml verion -->
                    <modules>
                        <ejbModule>
                            <groupId>{ejbModuleGroupId}</groupId>
                            <artifactId>{ejbModuleArtifactId}</artifactId>
                        </ejbModule>
                        <webModule>
                            <groupId>{webModuleGroupId}</groupId>
                            <artifactId>{webModuleArtifactId}</artifactId>
                            <contextRoot>/custom-context-root</contextRoot>
                        </webModule>
                    </modules>
                </configuration>
        </plugin>
    </plugins>
</build>
```

Une fois `mvn clean install` compilée, génère un artefact *.ear* dans le répertoire *cible* contenant à la fois le module ejb (.jar) et le module Web, ainsi que le fichier de description JEE *application.xml* .

Lire Plugin Maven EAR en ligne: https://riptutorial.com/fr/maven/topic/10111/plugin-maven-ear

Chapitre 12: Plugin Surefire Maven

Syntaxe

- test mvn
- mvn -Dest = test com.example.package.ExampleTest

Examples

Test d'une classe Java avec JUnit et le plugin Maven Surefire

Le plug-in Maven Surefire s'exécute pendant la phase de test du processus de génération Maven ou lorsque le `test` est spécifié comme objectif Maven. La structure de répertoires suivante et le fichier `pom.xml` minimum configurent Maven pour exécuter un test.

Structure du répertoire dans le répertoire racine du projet:

```
– project_root
  ├– pom.xml
  ├– src
  |  ├– main
  |  |  └– java
  |  └– test
  |     └– java
  └– target
     └– ...
```

`pom.xml` :

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>company-app</artifactId>
  <version>0.0.1</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Créez un fichier appelé `PlusTenTest.java` avec le contenu suivant dans le répertoire `src/test/java/com/example/app` :

```
package com.example.app;

import static org.junit.Assert.assertEquals;
```

```
import org.junit.Test;

public class PlusTenTest {

    @Test
    public void incrementTest() {
        int result = PlusTen.increment(10);
        assertEquals("PlusTen.increment(10) result", 20, result);
    }
}
```

L'annotation `@Test` indique à JUnit qu'elle doit exécuter `incrementTest()` comme test pendant la phase de `test` du processus de construction de Maven. Maintenant, créez `PlusTen.java` dans `src/main/java/com/example/app` :

```
package com.example.app;

public class PlusTen {
    public static int increment(int value) {
        return value;
    }
}
```

Exécutez le test en ouvrant une invite de commande, en accédant au répertoire racine du projet et en appelant la commande suivante:

```
mvn -Dtest=com.example.app.PlusTenTest test
```

Maven compilera le programme et exécutera la méthode de test `incrementTest()` dans `PlusTenTest` . Le test échouera avec l'erreur suivante:

```
...
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.005 sec <<< FAILURE! - in
com.example.app.PlusTenTest
incrementTest(com.example.app.PlusTenTest)  Time elapsed: 0.004 sec  <<< FAILURE!
java.lang.AssertionError: PlusTen.increment(10) result expected:<20> but was:<10>
at org.junit.Assert.fail(Assert.java:88)
at org.junit.Assert.failNotEquals(Assert.java:743)
at org.junit.Assert.assertEquals(Assert.java:118)
at org.junit.Assert.assertEquals(Assert.java:555)
at com.example.app.PlusTenTest.incrementTest(PlusTenTest.java:12)


Results :

Failed tests:
  PlusTenTest.incrementTest:12 PlusTen.increment(10) result expected:<20> but was:<10>

Tests run: 1, Failures: 1, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.749 s
[INFO] Finished at: 2016-09-02T20:50:42-05:00
[INFO] Final Memory: 14M/209M
```

```
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.19.1:test
(default-test) on project app: There are test failures.
...
```

Le plug-in Maven Surefire crée un `/target/surefire-reports/` dans le répertoire de votre projet contenant les fichiers `com.example.app.PlusTenTest.txt` et `TEST-com.example.app.PlusTenTest.xml` contenant les détails d'erreur du début. de la sortie ci-dessus.

En suivant le modèle de développement piloté par les tests, modifiez `PlusTen.java` pour que la méthode `increments()` fonctionne correctement:

```
package com.example.app;

public class PlusTen {
    public static int increment(int value) {
        return value + 10;
    }
}
```

Invoquer à nouveau la commande:

```
mvn -Dtest=com.example.app.PlusTenTest test
```

Le test réussit:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.example.app.PlusTenTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.028 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.753 s
[INFO] Finished at: 2016-09-02T20:55:42-05:00
[INFO] Final Memory: 17M/322M
[INFO] ------------------------------------------------------------------------
```

Toutes nos félicitations! Vous avez testé une classe Java à l'aide de JUnit et du plug-in Maven Surefire.

Lire Plugin Surefire Maven en ligne: https://riptutorial.com/fr/maven/topic/5876/plugin-surefire-maven

Chapitre 13: POM - Modèle d'objet de projet

Examples

Structure POM

Project Object Model est l'unité de base de Maven et définit la structure du projet, les dépendances, etc.

Les éléments suivants sont très minimes pour créer un POM:

- racine du `project`
- `modelVersion` - devrait être mis à `4.0.0`
- `groupId` - l'ID du groupe du projet
- `artifactId` - l'ID de l'artefact (projet)
- `version` - la version de l'artefact sous le groupe spécifié

`groupId` , `artifactId` et `version` sont appelés *coordonnées Maven* et parfois abrégés avec *GAV* . Ils identifient de manière unique l'artefact résultant d'un projet dans un référentiel Maven (et *devraient le* faire dans tout l'univers).

Un exemple de POM minimal ressemble à:

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

  <groupId>com.sample</groupId>
  <artifactId>sample-app</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</project>
```

Héritage POM

L'héritage est le principal atout du POM, où les éléments suivants peuvent être gérés, du super POM au POM enfant.

- dépendances
- développeurs et contributeurs
- listes de plugins (y compris les rapports)
- exécutions de plugin avec les identifiants correspondants
- configuration du plugin

Ce qui suit permet l'héritage

```
  <parent>
    <groupId>com.sample</groupId>
    <artifactId>sample-app-parent</artifactId>
    <version>1.0.0</version>
```

```
  </parent>
```

La structure de POM ressemble à

```
<project>
  <parent>
    <groupId>com.sample</groupId>
    <artifactId>sample-app-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.sample</groupId>
  <artifactId>sample-app</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</project>
```

Agrégation POM

Les modules d'un projet multi-modules sont agrégés à partir d'une structure hiérarchique.

Le `pom` racine devrait ressembler à:

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

La structure de répertoire du projet sera la suivante:

```
 |-- sample-app
  \   `-- pom.xml
   |-- sample-module-1
   |   `-- pom.xml
   |-- sample-module-2
   |   `-- pom.xml
```

POM racine:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.sample</groupId>
  <artifactId>sample-app</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>

  <modules>
    <module>sample-module-1</module>
    <module>sample-module-2</module>
  </modules>
  <dependencyManagement>
    ...
  </dependencyManagement>
</project>
```

Lire POM - Modèle d'objet de projet en ligne: https://riptutorial.com/fr/maven/topic/2310/pom---modele-d-objet-de-projet

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Apache Maven	A_Di-Matteo, Adonis, Community, darkend, Nate Vaughan, ngreen, Ray, Stephen Leppik, tobybot11, Tunaki
2	Accédez aux informations Maven dans le code	Derlin
3	Créer un plugin Maven	Gerold Broser, Matt Champion, Tunaki, Vince
4	Cycle de construction Maven	Gerold Broser, isudarsan
5	Effectuer une libération	isudarsan
6	Générer des rapports FIXME / TODO à l'aide du taglist-maven-plugin	Mahieddine M. Ichir
7	Intégration Eclipse	A_Di-Matteo, Gerold Broser, karel, kartik, Radouane ROUFID
8	Maven installer dans la fenêtre	shashigura
9	Maven Tomcat Plugin	kartik
10	Plugin de montage Maven	Jean-Rémy Revy, Tunaki, wallenborn, zygimantus
11	Plugin Maven EAR	Mahieddine M. Ichir
12	Plugin Surefire Maven	Gerold Broser, Nate Vaughan
13	POM - Modèle d'objet de projet	Gerold Broser, JF Meier, Radouane ROUFID, VinayVeluri, Vince