# Memo Maven

## Créer un projet

|  |  |  |
| --- | --- | --- |
|  | | **commande console** |
| Créer un projet maven (se placer dans le repertoire voulu d’abord) | | mvn archetype :generate [nom de l’archetype] |
| Exemple archetype  groupId // artifactId // laisser vide les 2 autres | mvn archetype:generate -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.1  com.truc.truc // mon-appli | |
| compiler | | mvn package |

## Configuration pom.xml

**information general**

// Un projet Maven est identifier par ses coordonnées tel que **groupId :artifactId :version**

// Une version avec le suffixe –SNAPSHOT : est une version en cours de développement, si on retire le suffixe = release

// Les properties sont des sortes de constantes que l’on peut utiliser dans le fichier de configuration pom.xml en les appelant avec ${ maPropertie }, cela permet de ne définir qu’une seul fois une variable et évite des erreurs

|  |  |
| --- | --- |
| **Propriétés prédéfinis** | |
| project.basedir | chemin vers le repertoire de base du projet |
| project.baseUri | chemin du repertoire de base du projet sous forme URI |
| maven.build.timestamp | horodatage du lancement du build Maven |
| env. | renvoi la valeur d’une variable d’enviranement (par exemple ${env.PATH}) |
| project. | renvoi la valeur d’une balise du fichier pom.xml (exemple : ${ project.organization.name } ) |
| settings. | renvoi la valeur d’une balise dans le fichier settings.xml |
| java. | renvoie la valeur d’une propriété système de java (exemple ${ java.version }) |

**Build**

// le build permet de paramétrer différents éléments du processus de construction de Maven, par exemple on peut rendre le JAR généré exécutable en demandant a Maven d’indiquer la classe Main dans le manifest du JAR

// ont peut filtrer des fichiers ressource qui n’ont pas a etre compilées mais simplement copiés dans le livrable généré par Maven, par convention, ces fichier se trouvent dans le repertoire src/main/ressources.

// Il est possible de dire a Maven de remplacer des propriété par leur valeurs dans des fichiers en filtrant ceux-ci. Exemple : dans un fichier mon.fichier se situant dans mon dossier ressource on aura une ligne org.exemple.demo.version=${project.version} puis dans le fichier pom.xml dans ressources>ressource>directory qui sera le chemin vers le repertoire concerné >filtering à true

**profil**

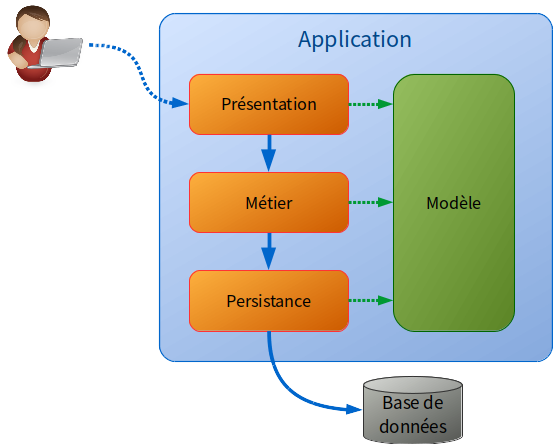
// les profils permettent de créer des options dans le build Maven, ont peut par exemple avoir deux environnements cible (un de test et l’autre de production) et embarquer dans le JAR généré, des fichiers de configurations différent en fonction de la cible

// pour déclarer un profil : balises profiles>profile>id***(son nom)***+build>resources>ressource>directory***(le repertoire)***

// pour construire un livrable pour l’environnement choisis : en console : mvn package –P monprofil

// on peut choisir de choisir un profil automatiquement en fonction de certains critère que l’on définis dans <activation>

## de MVC à une architecture multi-tiers

**Couche présentation** : cette couche contient les parties Controleur et vue du patron MVC, une fois l’action utilisateur identifiés, la couche Presentation fait appel à :

**Couche Métier** : cette couche est responsable de la logique métier de l’application, c’est

à dire de l’implémentation des regles de gestion foncitonnelles.

Si des acces à la base de données sont necessaires, alors la couche métier appelle

**Couche Persistance** : c’est dans cette couche que l’on retrouve le patron DAO

toutes ces couches partagent une « vision commune » du domaine fonctionnel en s’appuyant sur le

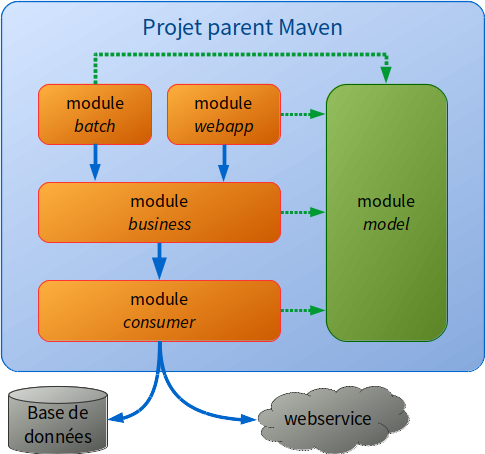
**modele**, qui contient les JavaBeans manipulés dans l’application ( à ne pas confondre avec le modele

MVC, ce dernier est en fait réparti ici sur les couche métier, persistance et modele)

Chaque couche n’appelle que la couche immédiatement en dessous d’elle et n’a aucune connaissance

des couches supérieures.

## separer les couches applicatives avec Maven – projet multi-modules



// On peut matérialiser le découpage de l’architecture multi-tiers grace à des

modules Maven. Chaque couche de l’architecture fait alors l’objet d’un module dédié.

// les modules sont des sortes de « sous-projet » Maven, ratachés à un projet

Maven principal (appelé project parent). les modules fournissent chacun leurs

propres livrables et il est possible de créer des dépendances entres eux.

//Ci-contre un exemple d’architecture multi-tiers découpés en module, ici

nous avont deux couche présentation suivant sous quel support tournera l’app.

## Créer un projet multi modules

1 ) créer un projet comme d’habitude, suprimer le dossier src, dans le fichier pom.xml changer le packaging en pom, et suprimer la dépendance vers JUnit

2 ) créer dans le repertoire de notre app chaque module (donner le meme groupId que le parent, changer le package en ajoutant .nomdumodule a la fin et artifactId = le nom du module) exemple :

mvn archetype:generate -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.1 -DartifactId=mon-app-batch –Dpackage=com.maven.training.batch –DgroupId=com.maven.training

ou encore pour un archetype webapp

mvn archetype:generate -DarchetypeArtifactId=maven-archetype-webapp -DartifactId=mon-app-webapp -Dpackage=com.maven.training.webapp

3 ) définir les dépendance entre les modules :

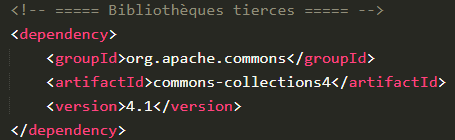
* Dans le pom du projet parent : lister dépendances et leur version de manière global pour le projet :

ajouter dans balise dependencyManagement>dependencies : pour chaque module : dependency>groupId + artifactId + version

* dans le pom de chaque module, definir quelles sont les dépendance à utiliser

ajouter dans balise dependency : pour chaque dependance : groupId + artifactId

## Gérez les dépendances



On ajoute la dépendance dans le module souhaité, puis on lance

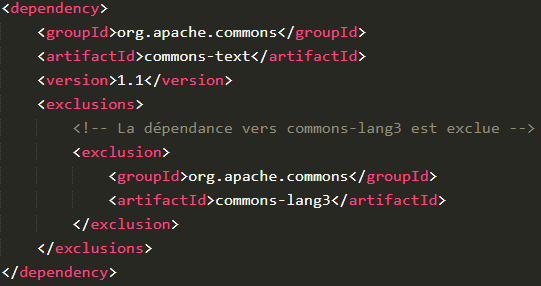
un packaging du projet, Maven ajoutera la dépendance au projet.

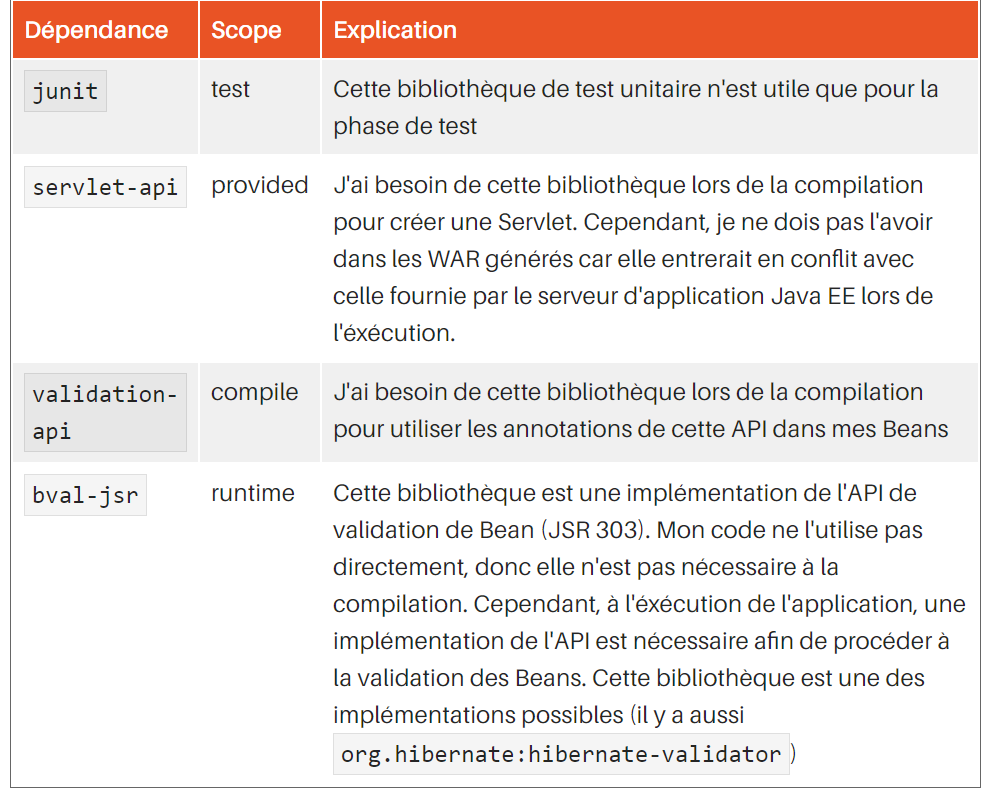
La dependance sera ajouter dans le dossier :

monmodule > target > monmodule.war

La transitivité des dépendance

La transitivité des dépendance est la capacité de maven d’ajouter automatiquement les dépendances requises par les dépendance que l’on ajoute. Il est cependant possible d’excure certaine transitivité



Définir la porté d’une dépendance

On peut définir la porté du dépendance grace au scope. il en existe 4

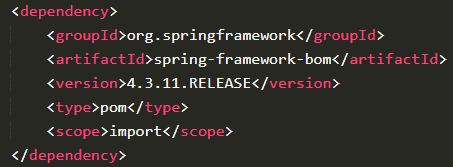
//compile : scope par défaut : accessible dans tout les context

// test : accessible que lors de la compilation des test et leur execution

// provided : disponible à la compilation, mais devra etre fournie par le contexte d’execution (le serveur d’application par exemple)

// runtime : pas disponible lors de la compilation mais disponible à l’execution

gérer les dépendance de manière global

Afin de gérer plus efficacement nos dépendance comme leur version et leur scope, il faut les déclarer dans notre projet pincipale, puis dans chaque module nous n’indiquerons que le groupId ainsi que artifactId

On peut utiliser ce qu’on appel une Bill Of Materials (BOM) qui est un fichier POM spécifique, mis à disposition des mainteneur de Framework afin de telecharger toute les dépendance indispensable au bon fonctionnement de celui-ci. Dans ce cas la on utilisera le scope import

Version SnapShot et Release

Les version Realease sont immuable, une fois qu’elle sont publié aucune modification ne sera apporté, a moins de faire une nouvelle version, alors que les snapshot sont des version en cours de developpement.

## Cycle de vie du build

Afin d’automatiser la construction d’un projet, Maven s’appuie sur des cycles de vie de construction appelés build lifecycle.

Il y as 3 build lifecycle :

// default : qui permet de construire et deplayer le projet

//clean qui permet de nettoyer le projet en suprimant les elements issus de la construction de celui-ci

//site qui permet de créer un site web pour le projet.

Ces build lifecycle sont découpés en phases qui sont executées sequentiellement les unes à la suite des autres, par exemple dans le build lifecycle default en retrouve entre autre les phases :

//validate : verifie que la configuration est correct (POM, pas d’element manquant…)

//compile : compile les sources du projet

//test : test le code compilé avec les classes de test unitaire contenue dans le projet

//package : package les elements issus de la compilation dans un format distribuable (jar, war…)

//install : installe le package dans notre repository local

//deploy : envoie le package dans le repository distant définis dans le POM

### lancer une build lifecycle

//Quand on lance une construction Maven en ligne de commande, on precise une phase d’un des build lifecycles et maven se charge d’executer, dans l’ordre, toutes les phases qui composent le build lifecycle jusqu’à la phase indiquée.

// pour les projet multi-modules, quand on lance la commande mvn sur le projet parent, maven lance le build lifecycle dans chaque sous-projet (module), les uns à la suite des autre, en respectant l’ordre des dépendances inter-modules.

// il est aussi possible de chainer l’ecxecution de plusieurs build lifecycles dans une seul commande Maven. par exemple si on lance un mvn clean package, maven va tout d’abord netoyer le projet avec clean, puis lancer le build lifecycle default jusqu’à la phase package.

### les goals

Les phases d’un build lifecycle sont découpées en taches, chaque tache est assurée par un plugin Maven, dans le jargon Maven, ces taches s’appellent des goals. Par exemple la phase test est réalisée par défaut par le goal surefire :test, c’est-à-dire le goal test du plugin surefire. Maven fournit de base une certain nombre de plugins.

## Plugins

// Le but des plugins est de pouvoir personnaliser la construction du projet, en enrichissant le POM afin de coller le plus possible à notre besoin et ainsi d’automatiser au maximum la construction et la génération des livrables de notre projet.

// suivant le build lifecycle et le packagins utilisés, différents goals sont cablés par defaut aux differente phase.

### Affiner la configuration de la compilation

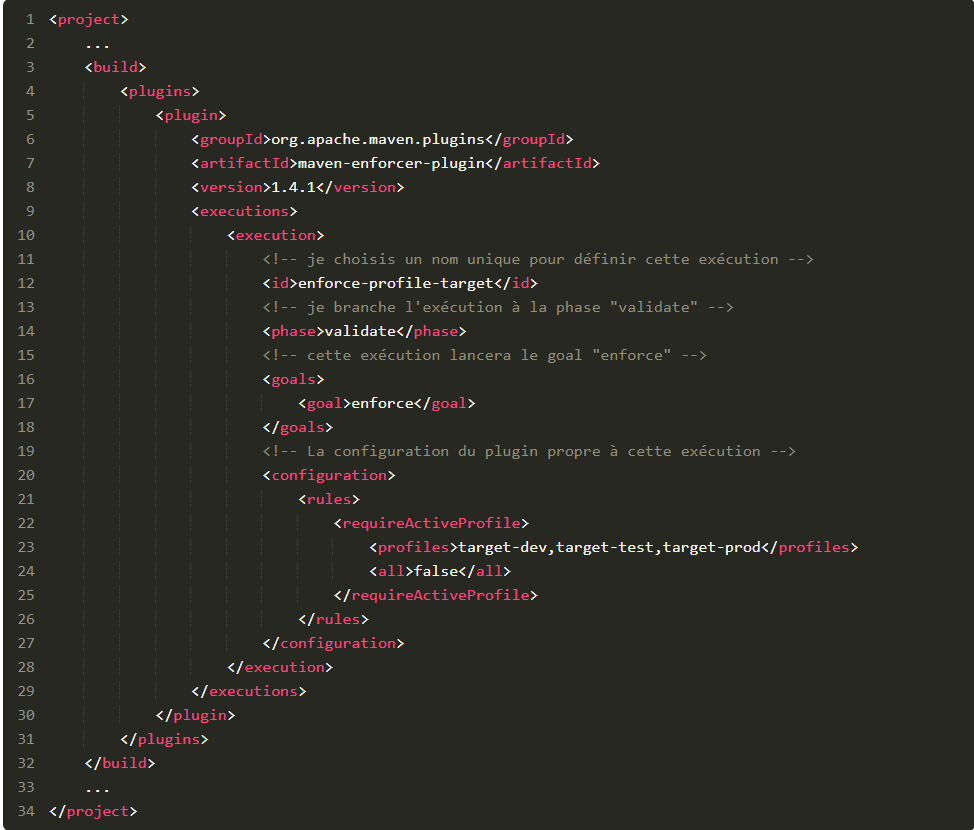
// Par exemple, lors de la compilation, on peut paramétrer un pluging afin qu’il affiche les utilisations de méthodes dépréciées, ainsi que des avertissements de compilation. Dans la documentation la phase compile est assurée par le goal compile du plugin org.apache.maven.plugins :maven-compiler-plugin, je vais donc dans la documentation du plugin et j’obtiens ainsi une description du goal et la liste des options de configuration qui lui sont applicables.

// il y as deux façon de configurer les option des plugins =============🡺

### gerer les plugins de manière globale

Il est fortement recommandé de définir les versions des plugins utilisés. Comme pour les dépendance il est possible de le faire de manière globale au projet dans le POM parent. On peut ainsi fixer les versions des plugins mais aussi leur configuration de manière global. Certaine configuration ne peut se faire que dans les modules concernés, par exemple quand ont défini une classe main, qui sera différente pour tous les modules.

### Ajouter de nouveaux plugins dans les phases



// Par exemple, je créer 3 profils, qui seront : target-dev, target-test et target-prod

// Lors du build avec Maven je veux m’assurer qu’au moins un des profils est activé.

// J’utilise dans ce cas le goal enforce du plugin maven-enforcer-plugin.

// Maintenant si je lance un mvn package sans définir de profile j’aurais une erreur, je doit alors ajouter un parametre tel que :

mvn package –Ptarget-dev

## Organisez et packagez une application

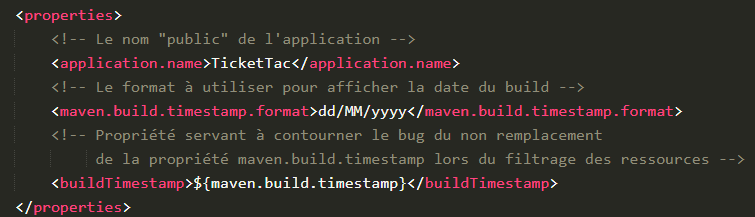
### generer un WAR de l’application Web

// On utilise une extension WAR réalisé par le plugin maven-war-plugin afin de pouvoir déployer une application web. Je veux par exemple ajouter certaines informations du projet dans les JSP et m’assurer qu’aucune version SNAPSHOT ne soit envoyée en production.

// Afin d’ajouter automatiquement certaines informations du projet dans les JSP, je vais utiliser le même mécanisme que pour les fichiers de ressource classique : le filtrage, cependant les JSP ne sont pas des ressources classiques, elles doivent être ajoutée au répertoire WAB-INF du WAR. Le plugin maven-war-plugin fait déjà cela et permet également de mettre en place le filtrage de ces web ressources.

// Dans le POM parent j’ajoute donc le plugin maven-war-plugings (groupId + artifactId + version) , puis dans mon module avec l’archetype web app, je configure le filtrage des web ressources ===========================🡺

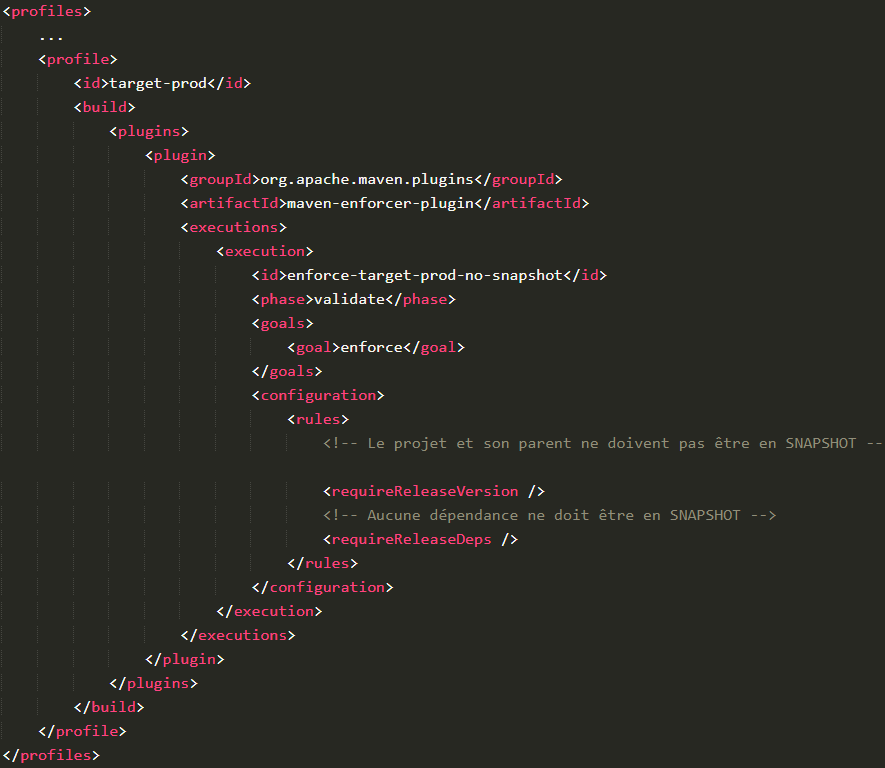
// Puis je définie certaines propriétés :



// Il ne reste plus qu’a utiliser les propriétés dans les JSP afin que leurs valeurs soient injectées lors du filtrage par le plugin maven-war-plugin : j’ajoute pour accéder a ces propriété dans mes jsp : ${application.name} et ${maven.build.timestamp}

// Maintenant si je veux vérifier qu’aucune version SNAPSHOT ne soit envoyée en production je vais utiliser le plugin maven-enforcer-plugin.

//Comme cette vérification concerne uniquement les constructions ayant pour cible la production, je vais ajouter l’exécution de ce plugin dans le profil de production.

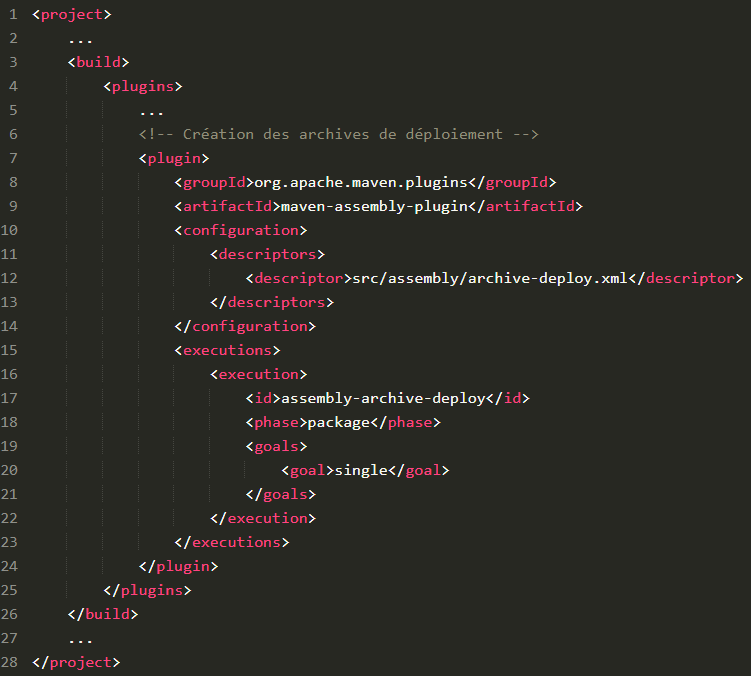


### Générer une archive (un fichier TAR.GZ + ZIP)



1. Etant donné que je veux fournir les JAR des dépendances à coté du JAR du module batch, j’ajoute le classpath au manifest de ce JAR. (attention meme si la balise classpathPrefix est vide il faut la mettre)
2. Afin de générer les fichier d’archive on utilise le plugin maven-assembly-plugin, et la définission se fait grace à des fichiers XML dans le repertoire src/assembly

* id : l'identifiant de l'assemblage. Maven va se servir de cet identifiant dans le nom des fichiers générés : <project.finalName>-<id>.<format>. Ce qui donnerait par exemple : ticket-batch-1.0-SNAPSHOT-archive-deploy.tar.gz.
* formats : les formats de fichier à générer. Ici un fichier *tar.gz* et un fichier *zip*.
* dependencySets : ajout d'un ensemble de dépendances. Ici, j'ajoute toutes les dépendances de *runtime* (scope *runtime* et *compile*) du projet dans le répertoire de destination lib.
* fileSets : ajout d'un ensemble de fichiers. Ici, j'ajoute deux ensembles de fichiers :
  + les scripts de lancement des batchs contenus dans le répertoire src/data/scripts, dans le répertoire de destination bin. J'y positionne les droits Unix, notamment celui d'exécution : <fileMode>0755</fileMode> (-rwxr-x-rx).
  + les fichiers de configuration contenus dans le répertoire src/data/conf, dans le répertoire de destination conf.



1. il ne reste plus qu’a cabler le génération des fichiers d’archive à la phase package

## Generez un site pour le projet