

MAVEN
Formation



SOMMAIRE

- + Introduction à Apache Maven
- + Installation de Maven
- + Le POM (Modèle Objet de Projet)
- + Cycle de vie du build
- + Profils de Build
- + Propriétés et filtrage des ressources
- + Options d'exécution
- + Configuration avancée
- + Génération de Site / rapports
- + Création de Plugins
- + Support de Maven par les IDE
- + Maven et l'Intégration Continue



INTRODUCTION

+ Approche fonctionnelle

- ▀ Maven est un outil de build
- ▀ Maven est un gestionnaire de packages

+ Approche technique

- ▀ Maven est un micro noyau extensible par plugins
- ▀ Convention over configuration

+ Synthèse

- ▀ Synthèse de la philosophie Maven
- ▀ Documentation de référence

+ Approche comparative

- ▀ Maven vs Ant
- ▀ Les alternatives

MAVEN

est un outil de

BUILD

INTRODUCTION

Maven : Approche fonctionnelle

+ Maven est un outil de build

« Build » : ensemble d'opérations destinées à produire un artefact logiciel



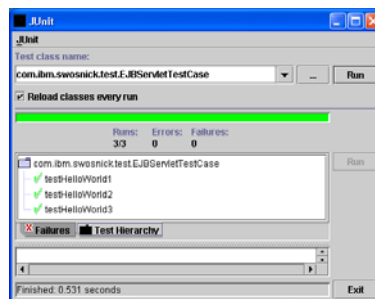
Compiler le code source en bytecode

INTRODUCTION

Maven : Approche fonctionnelle

+ Maven est un outil de build

« Build » : ensemble d'opérations destinées à produire un artefact logiciel



Passer les tests automatisés et vérifier le résultat

INTRODUCTION

Maven : Approche fonctionnelle

+ Maven est un outil de build

« Build » : ensemble d'opérations destinées à produire un artefact logiciel



Analyser la qualité du code :

- Respect des normes
- Détection d'anti-patterns et de bugs

INTRODUCTION

Maven : Approche fonctionnelle

+ Maven est un outil de build

« Build » : ensemble d'opérations destinées à produire un artefact logiciel



Packager l'ensemble des ressources en un artefact conforme :

- Fichier JAR respectant la spécification JAR
- Fichier WAR respectant la norme JEE

-- ...

INTRODUCTION

Maven : Approche fonctionnelle

+ Maven est un outil de build

Le build peut se faire manuellement

- en ligne de commande
- avec des fichiers de scripts (shells scripts, cmd windows)
- via les menus de votre IDE



C'EST MAL

- fastidieux
- source d'erreurs
- perte de temps et d'énergie
- résultat non prédictible et non reproductible



INTRODUCTION

Maven : Approche fonctionnelle

+ Maven est un outil de build

Le build peut s'automatiser avec un outil dédié



C'EST BIEN

- rapide
- résultat reproductible
- non coûteux
- déclenchable sur conditions (... intégration continue ...)



INTRODUCTION

Maven : Approche fonctionnelle

+ Maven est un outil de build

maven *Permet d'automatiser la phase de build*



Comme ANT avant lui

MAIS

En beaucoup mieux

INTRODUCTION

Maven : Approche fonctionnelle

MAVEN est un gestionnaire de PACKAGES

INTRODUCTION

Maven : Approche fonctionnelle

+ Maven est un gestionnaire de packages

Un gestionnaire de paquets est un outil automatisant le processus d'installation [...] de logiciels sur un système informatique.

maven Gère des paquets



RPM PACKAGE
MANAGEMENT

*Comme les gestionnaires de package
utilisés sous Linux pour installer des
fonctionnalités*



DEBIAN / UBUNTU
PACKAGE MANAGEMENT

INTRODUCTION

Maven : Approche fonctionnelle

+ Maven est un gestionnaire de packages

Un gestionnaire de paquets est un outil automatisant le processus d'installation [...] de logiciels sur un système informatique.

maven Gère des paquets spécifiques



*Des librairies JAVA
- Des fichiers JAR donc
- Appelées artefacts*

INTRODUCTION

Maven : Approche fonctionnelle

+ Maven est un gestionnaire de packages

Comme tout gestionnaire de packages, il s'appuie sur un **repository** accessible via Internet et dans lequel il va chercher les paquets **requis**.



Le paquet qu'on lui demande de mettre à disposition

ET

Les paquets dont a besoin le paquet demandé pour fonctionner (ses dépendances)

ET

Les paquets dont ont besoin les paquets dont a besoin le paquet demandé pour fonctionner

etc ...

INTRODUCTION

Maven : Approche fonctionnelle

Gestion transitive des dépendances



+ Maven est un gestionnaire de packages

*Comme tout gestionnaire de packages, il s'appuie sur un **repository** accessible via Internet et dans lequel il va chercher les paquets **requis**.*



Exposé sur Internet

Une interface web permet d'en browser le contenu : <http://search.maven.org/>

TOUS LES PROJETS OPEN SOURCE IMPORTANTS ont adopté Maven et déposent leurs jars et la description de leurs propres dépendances sur ce repository

C'est un standard de l'industrie

Il existe d'autres repositories gérés par différentes organisations (peu utilisés)

MAVEN est un micro kernel extensible via plugins

INTRODUCTION

Maven : Approche technique

+ Maven est un micro kernel (noyau in french)



Le cœur de Maven ne pèse rien (comme nous le verrons lors de son installation)

Le cœur de Maven a très peu de fonctions.

Il doit être étendu par des composants spécialisés, spécifiquement conçus pour être invocables par son noyau.



Ces composants sont appelés PLUGINS ou MOJO

 The Central Repository

*Ces plugins sont des artefacts particuliers.
Comme tous les artefacts ils sont dans le repository central
Le noyau les télécharge dès qu'il en a besoin*

INTRODUCTION

Maven : Approche technique

+ Maven est un micro kernel (noyau in french)



Chaque plugin expose un certain nombre de services invocables

*Chacun de ces services est appelé **GOAL***

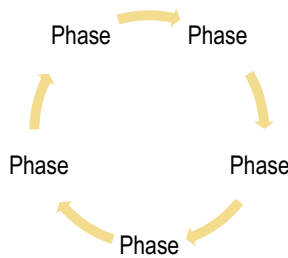
Ces GOALS peuvent être invoqués via la ligne de commande

```
>mvn plugin:goal
```

INTRODUCTION

Maven : Approche technique

+ Maven est un micro kernel (noyau in french)



Au cœur de Maven on trouve la notion de **ProjectLifeCycle**

Un **ProjectLifeCycle** est une succession ordonnée de **phases**

Le noyau Maven pilote des ProjectLifeCycle et comprend la définition de 3 ProjectLifeCycle, dont un par défaut.

Le cycle de vie projet par défaut est le cycle de vie d'un build ; il décrit les phases successives d'un processus de build

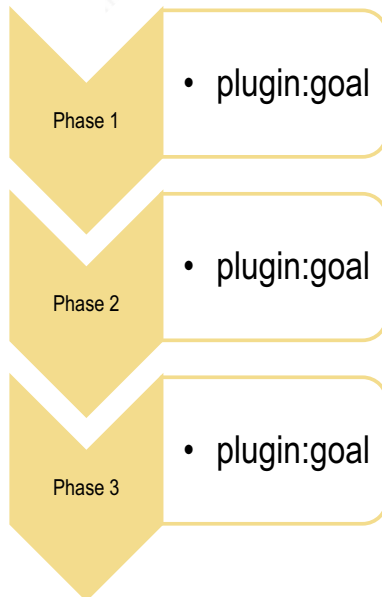
INTRODUCTION

Maven : Approche technique

https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.htm#Lifecycle_Reference

Clean Lifecycle	
pre-clean	executes processes needed prior to the actual project cleaning
clean	remove all files generated by the previous build
post-clean	executes processes needed to finalize the project cleaning
Default Lifecycle	
validate	validate the project is correct and all necessary information is available.
initialize	initialize build state, e.g. set properties or create directories.
generate-sources	generate any source code for inclusion in compilation.
process-sources	process the source code, for example to filter any values.
generate-resources	generate resources for inclusion in the package.
process-resources	copy and process the resources into the destination directory, ready for packaging.
compile	compile the source code of the project.
process-classes	post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.
generate-test-sources	generate any test source code for inclusion in compilation.
process-test-sources	process the test source code, for example to filter any values.
generate-test-resources	create resources for testing.
process-test-resources	copy and process the resources into the test destination directory.
test-compile	compile the test source code into the test destination directory.
process-test-classes	post-process the generated files from compilation, for example to do bytecode enhancement on Java classes. For Maven 2.0.5 and above.
test	run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
prepare-package	perform any operations necessary to prepare a package before the actual packaging. This often results in an unprocessed, processed version of the package. (Maven 2.1 and above)
package	take the compiled code and package it in its distributable format, such as a JAR.
pre-integration-test	perform actions required before integration tests are executed. This may involve things such as setting up the required environment.
integration-test	process and deploy the package if necessary into an environment where integration tests can be run.
post-integration-test	perform actions required after integration tests have been executed. This may include cleaning up the environment.
verify	run any checks to verify the package is valid and meets quality criteria.
install	install the package into the local repository, for use as a dependency in other projects locally.
deploy	done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.
Site Lifecycle	
pre-site	executes processes needed prior to the actual project site generation
site	generates the project's site documentation
post-site	executes processes needed to finalize the site generation, and to prepare for site deployment
site-deploy	deploys the generated site documentation to the specified web server

+ Maven est un micro kernel (noyau in french)



*Maven associe un goal d'un plugin à chaque phase de chaque cycle de vie projet > notion de **binding***

Ces GOALS peuvent être déclenchés via la ligne de commande

```
>mvn phase  
(ex : mvn package)
```

Maven exécute successivement le goal associé à chaque phase de rang inférieur ou égal à la phase passée en argument

Voilà comment le noyau Maven permet de piloter des builds

Convention over Configuration

INTRODUCTION

Maven : Approche technique

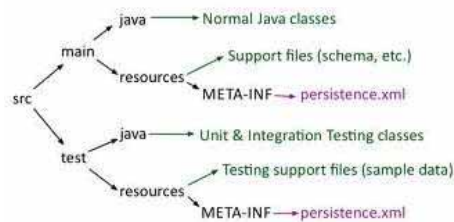
+ Convention over configuration

Les différents goals invoqués durant le build vont avoir besoin de certaines informations essentielles :

- Où se trouve le code source ?
- Où ranger les classes compilées ?
- Où déposer l'artefact généré ?

Pour éviter à chaque projet de devoir fournir ces informations, Maven définit un **layout standard**, aka un modèle de répertoires.

En suivant ces **conventions**, tout projet peut bénéficier directement de Maven, sans devoir passer par une fastidieuse phase de **configuration**



INTRODUCTION

Maven : Synthèse

Synthèse

On a donc :

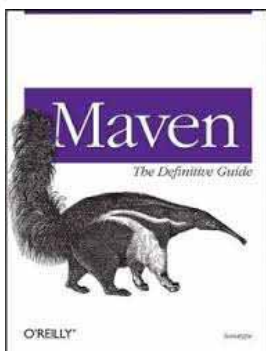
- un **repository** central sur Internet
- une série de **plugins**, téléchargés au premier usage, qui sont des composants spécifiques et capables de certaines activités (compilation, lancement de tests, packaging ...) exposées sous forme de **goals**
- un **cycle de vie** par défaut qui définit le cycle de build de tout projet
- Une **association** (binding) entre des **goals** et les **phases** du cycle de vie
- Un **layout standard**
- Une **interface en ligne de commande** qui permet d'exécuter un goal d'un plugin (mvn plugin:goal) ou tous les goals associés aux phases successives d'un cycle de vie projet (mvn phase)

Il nous manque encore :

- la notion de **repository local**, qui est un répertoire local dans lequel maven recopie les plugins et autres artefacts téléchargés depuis le repository central
- le **descripteur de projet** (pom.xml) qui est un ensemble de meta-données sur le projet qui sont consommées par Maven et permettent de
 - spécialiser le comportement des différents plugins invoqués lors des différentes phases via des variables d'environnements (propriétés)
 - intervenir sur les bindings par défaut, pour par exemple faire déclencher d'autres goals (d'autres plugins) lors du passage sur une phase, en fonction des besoins spécifiques du projet
 - définir les dépendances de premier niveau (paquets à télécharger)

Une fois ceci assimilé, le reste n'est que du détail de mise en œuvre

- ✓ *La configuration de Maven est à connaître*
- ✓ *Une connaissance des différents plugins et de leurs goals est requise*
- ✓ *Une connaissance du format du descripteur de projet est requise*
- ✓ *Quelques subtilités et bonnes pratiques sont à assimiler*



Livre de référence sur Maven. Disponible gratuitement

Traduction française :

<http://maven-guide-fr.erwan-alliaume.com/maven-guide-fr/site/reference/public-book.html>

Site web, excellente documentation, informations extensives sur les plugins

<http://maven.apache.org/index.html>

Pourquoi MAVEN a supplanté ANT

	Maven	Ant
Layout	Standardisé	Spécifique à chaque projet
Cycle de build	Standardisé et prédéfini A l'état de l'art	Spécifique à chaque projet Plus ou moins bien pensé
Gestion des dépendances	Transitive, configurable (gestion des conflits)	Rien de base, mais possible via une intégration Ivy
Adoption	Très large soutien	En perte de vitesse
Richesse	Moindre que ANT mais en progression constante	Très large catalogue de Tasks spécifiques
Souplesse	Moindre que ANT, par définition	Totale

Maven offre également un plugin qui permet d'exécuter des Tasks ANT pour faciliter la transition et profiter du large catalogue de Tasks ANT

INTRODUCTION

Maven : Approche comparative

+ Quelques pistes de réflexion sur les alternatives



Groovy Ant scripting, no dependency management or extras.

If you're still used to Ant build files and presumably doing most of the dependency management by hand, then Gant is probably a great choice for you. It hides the Ant XML that can become very unwieldy and syntax heavy in favor of more focussed Groovy.



Gradle combines the power and flexibility of Ant with the dependency management and conventions of Maven into a more effective way to build. Powered by a Groovy DSL and packed with innovation, Gradle provides a declarative way to describe all kinds of builds through sensible defaults.



The agile dependency manager

Apache Ivy™ is a popular dependency manager focusing on flexibility and simplicity.

INSTALLATION

INTRODUCTION

Maven : Installation



Downloadez depuis le site maven : <http://maven.apache.org/download.cgi>

Dézippez l'archive où bon vous semble (éviter les espaces blancs dans le chemin)

Ajoutez le répertoire « bin » de la distribution Maven à votre PATH système

Déclarez une variable `JAVA_HOME` qui pointe sur votre JDK.

Déclarez une variable d'environnement `M2_HOME` qui pointe sur la racine du dossier d'install (optionnel)

```
>mvn -version
```



LE POM (MODÈLE OBJET DE PROJET)

Sommaire

- + Introduction**
- + Le POM**
- + Dépendances d'un projet**
- + Relations entre projets**
- + Les bonnes pratiques du POM**

LE POM (MODÈLE OBJET DE PROJET)

Sommaire

- + Introduction**
 - ▀ Problématique projet
 - ▀ Un fichier pour centraliser toutes les informations
- + Le POM**
- + Dépendances d'un projet**
- + Relations entre projets**
- + Les bonnes pratiques du POM**

LE POM (MODÈLE OBJET DE PROJET)

Introduction – problématique projet

+ Tout projet informatique est en général constitué de :

- ▀ Code source
- ▀ Documentation
- ▀ Ressources (images, vidéos, fichiers de configuration...)
- ▀ Dépendances externes (bibliothèques de programmation tierce-partie)

+ Pour être exploités, ces constituants doivent être :

- ▀ compilés (code source java, .Net...)
- ▀ vérifiés
- ▀ packagés (jar, zip, ...)
 - + Parfois de façons différentes en fonction de l'usage (clients différents, environnements techniques différents...)
- ▀ Livrés / déployés aux utilisateurs finaux

LE POM (MODÈLE OBJET DE PROJET)

Introduction – problématique projet

+ Chaque nouvel incrément du projet livré possède un numéro de « version »

- ▀ Ce numéro augmente en général avec le temps, selon des règles spécifiques à chaque projet
- ▀ Il permet d'identifier avec précision de quelle « version » du projet on parle

+ Culturellement, un projet est souvent « rattaché » à une organisation

- ▀ « Windows » -> « Microsoft »
- ▀ « Tomcat » -> « Apache »
- ▀ « Java » -> « Sun »

LE POM (MODÈLE OBJET DE PROJET)

Introduction – Un fichier pour centraliser toutes les informations

+ Maven propose de formaliser et recenser un certain nombre de ces informations dans un « modèle objet de projet »

- Où sont les sources ?
- Quelle est le numéro de version du projet ?
- Quelles sont les dépendances du projet ?
- ...

+ L'objectif étant de centraliser toutes ces informations utiles pour pouvoir les exploiter facilement et effectuer des tâches du cycle de vie projet

- Compilation, packaging, exécution de tests automatisés, copie sur une machine distante...

LE POM (MODÈLE OBJET DE PROJET)

Introduction – Un fichier pour centraliser toutes les informations

+ Ce modèle objet de projet (POM – Project Object Model) est décrit dans un fichier xml de configuration associé au projet

- Ce fichier fait partie du code source de l'application, la plupart du temps à la racine de celui-ci
- C'est ce fichier que va lire maven pour prendre connaissance de tous les aspects utiles à son fonctionnement dans le cadre du projet
- Ce fichier permet aussi d'activer et configurer certaines tâches (effectuées par les plugins) – **mais ce n'est pas un fichier de scripting, comme ant !**

LE POM (MODÈLE OBJET DE PROJET)

Sommaire

+ Introduction

+ Le POM

- Introduction et exemple de POM minimaliste
- Coordonnées du projet
- Le Super POM
- Le POM effectif
- Contenu du POM

+ Dépendances d'un projet

+ Relations entre projets

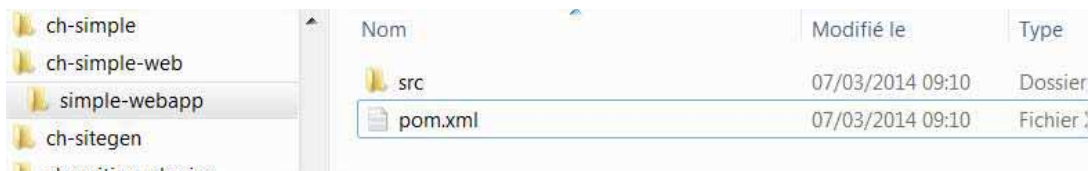
+ Les bonnes pratiques du POM

LE POM (MODÈLE OBJET DE PROJET)

Introduction

+ Le POM est donc décrit dans un fichier xml

- Nommé pom.xml depuis la version 2 de maven
- Chaque projet, ou sous-module de projet (nous y reviendrons), possède son propre pom, la plupart du temps à la racine des sources



LE POM (MODÈLE OBJET DE PROJET)

Exemple de pom minimaliste

+ Exemple de fichier pom minimaliste

- ▀ C'est un fichier xml encodé en UTF-8
- ▀ Normalisé par un schéma xsd
 - + Les parties plus claires sont à reprendre systématiquement tel quel

```
<?xml version="1.0" encoding="UTF-8"?>
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.sqli.training</groupId>
  <artifactId>maven-example</artifactId>
  <version>1</version>

</project>
```

LE POM (MODÈLE OBJET DE PROJET)

Coordonnées du projet

+ On trouve dans ce pom minimaliste :

- ▀ groupId : permet d'identifier le projet comme appartenant à un groupement plus vaste (nom d'une organisation, nom d'un service, d'un projet plus global dans le cas d'un sous projet...)
- ▀ artifactId : détermine le nom précis du projet au sein de ce groupId
- ▀ version : quelle est la version courante du projet

+ Ce sont les seules informations à faire figurer dans un pom pour que maven puisse fonctionner :

- ▀ Certaines informations ou actions à effectuer lors d'un build, comme la localisation des fichiers sources à compiler, ou le type de packaging à produire, sont pré-configurées par défaut (nous verrons où et comment)

LE POM (MODÈLE OBJET DE PROJET)

Coordonnées du projet

+ Le triplet `groupId:artifactId:version` forme ce que l'on appelle les **coordonnées du projet**

- ▀ Dans l'exemple précédent, notre projet est donc identifié par le triplet

+ `com.sqli.training:maven-example:1`

+ Lorsque nous aborderons la problématique des **dépendances**, nous verrons que celles-ci sont elles aussi **identifiées par leur coordonnées**

- ▀ Ex :

+ `org.springframework:spring-beans:3.2-RELEASE`

+ `org.hibernate:hibernate-entitymanager:4.2.1.Final`

- ▀ Ce système de coordonnées est présent dans de nombreux systèmes de gestion de dépendances (linux, ivy, sbt, gradle...)

LE POM (MODÈLE OBJET DE PROJET)

Coordonnées du projet

+ Le **groupId** est

- ▀ En général constitué du nom de l'entreprise précédé de com, org, fr...

+ `com.sqli, fr.sogem`

- ▀ Il peut aussi faire référence au nom de domaine associé au projet

+ `org.springframework`

+ L'**artifactId**

- ▀ Est en général le nom du projet

+ `datavance, hermes ...`

- ▀ Peut-être composé du nom du projet et du sous projet, séparés par un '-'

+ `datavance-front, datavance-back, hermes-web-services...`

LE POM (MODÈLE OBJET DE PROJET)

Coordonnées du projet

+ La version

- Est en général composé d'un numéro à plusieurs chiffres
 - + 1, 3.0, 2.5.5
- Ce numéro peut être étendu pour encoder des informations plus précises
 - + 2.5-BETA
- A chaque projet / organisation de poser les règles de construction du numéro
 - + Ex : Quand incrémente on le numéro majeur, le numéro mineur ?
- Maven **n'impose pas** une nomenclature fixée ; cependant, il est conseillé de respecter certaines règles ...

LE POM (MODÈLE OBJET DE PROJET)

Coordonnées du projet

+ Maven propose et comprend une nomenclature précise pour le numéro de version

- + <major version>.<minor version>.<incremental version>[-<qualifier>]
- Il n'est pas obligatoire de la suivre sur son propre projet
- Mais il ne faut pas oublier que certains plugins se basent sur cette nomenclature pour effectuer certains choix, ou actions
 - + Par exemple, le plugin delivery doit choisir un nouveau numéro de version automatiquement pour le projet, lors d'une release
 - Passer de **1.0.1** à **1.0.2** automatiquement est facile, mais qu'y a-t-il après VER12b, par exemple ?
 - + Ou déterminer si une version d'un projet est plus récente qu'une autre (tri des numéros de version)

LE POM (MODÈLE OBJET DE PROJET)

Coordonnées du projet, version SNAPSHOT

+ La chaîne **-SNAPSHOT**, dans le numéro de version à une signification spéciale pour maven

- SNAPSHOT est un **qualifier**
- Il indique que le projet est en cours de développement et donc potentiellement instable
 - + Aussi, avant toute release / livraison officielle, il convient de passer le numéro de version en numéro « stable », en retirant le **-SNAPSHOT**
 - 1.2-SNAPSHOT -> 1.2, (ou même 1.2-RELEASE, ...)
- Si l'on souhaite quand même déployer (plugin 'release') une version SNAPSHOT, maven va substituer cette chaîne par un timestamp de l'heure de release
 - 1.2-SNAPSHOT -> 1.2-20080207-230803-1
- + En cas de dépendance **-SNAPSHOT**, c'est la version la plus récente qui sera alors utilisée (tri grâce au timestamp)

LE POM (MODÈLE OBJET DE PROJET)

TP

+ Pré-requis :

- Maven installé

+ Objectif :

- Créez un fichier pom minimaliste pour :
 - + Le projet « Foobar »
 - + De la société « sqli »
 - + En version 1
- Lancer la commande 'mvn install'
 - + Vous deviez obtenir un BUILD SUCCESS
 - + Observez la sortie console pour essayer de comprendre ce qui se passe

LE POM (MODÈLE OBJET DE PROJET)

Le Super Pom

+ Nous avons vu que maven adoptait un certain nombre de conventions par défaut pour la majorité des informations projet qui ne sont pas précisées dans notre POM

+ Un certain nombre d'informations sont issues d'un **Super POM**

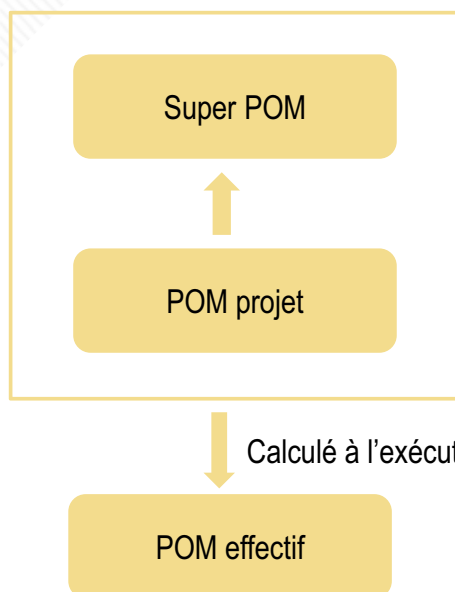
- Ce POM est un POM ancêtre des POM projets
- Tout POM projet hérite de celui-ci
- Des valeurs par défauts y sont renseignées

+ Le Super POM est contenu dans les librairies de maven

- Dans le jar maven-model-builder-xxx
- Sous le nom pom-4.0.0.xml

LE POM (MODÈLE OBJET DE PROJET)

Le Super Pom



Le Super POM sert de base à maven

IL est fusionné avec le pom projet pour produire le « pom effectif »

Le pom effectif est le résultat de la fusion entre le super POM et le POM projet

LE POM (MODÈLE OBJET DE PROJET)

Super Pom et pom effectif

+ La commande :

- mvn help:effective-pom

permet d'afficher le pom issu de ce merge.

+ On y retrouve nos informations spécifiques au projet, ainsi que d'autres, issues du Super POM, avec des valeurs par défaut

LE POM (MODÈLE OBJET DE PROJET)

TP – pom effectif et super pom

+ Pré-requis :

- Maven installé
- Un pom minimaliste

+ Objectif :

- Exécutez la commande mvn help:effective-pom
- Observez le POM résultant

LE POM (MODÈLE OBJET DE PROJET)

Contenu du POM – Conventions et valeurs par défaut du Super POM

+ L'affichage du POM effectif nous a montré que tout POM définit en réalité de nombreuses informations

- Ces informations par défaut sont toutes surchargeables dans notre propre pom projet (il suffit de les ré-écrire), mais...
- Ces informations par défaut forment les **conventions maven** : les adopter permet à toute personne familière de maven de pouvoir prendre en main le projet très facilement
 - + Les choses sont « toujours à la même place » (layout standard)
 - + Builder , packager, compiler revient à toujours exécuter la même commande, quel que soit le projet
 - + Customiser un aspect du build projet revient toujours à la même chose
 - Configurer si besoin le bon plugin
 - Ajouter un nouveau plugin (toujours de la même façon, dans le pom)

LE POM (MODÈLE OBJET DE PROJET)

Contenu du POM – Structure générale

+ Observons les différentes parties du POM effectif de notre POM minimaliste :

```
<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>com.sqli.training</groupId>
  <artifactId>maven-example</artifactId>
  <version>1</version>
  <repositories> (...)
</repositories>
  <pluginRepositories> (...)
</pluginRepositories>
  <build>(...)
</build>
  <reporting>(...)
</reporting>
</project>
```

LE POM (MODÈLE OBJET DE PROJET)

Contenu du POM – Repositories (intro)

+ Repositories et pluginRepositories

- Indique à maven où chercher les dépendances projet ainsi que les plugins maven à exécuter lors du build
- Détaillé au chapitre de gestion des dépendances

+ Reporting

- Indique où doivent-être générés les rapports produits par maven lors de l'activation d'un build particulier
- Détaillé au chapitre Génération de site

LE POM (MODÈLE OBJET DE PROJET)

Contenu du POM – Build

+ Build

- Cette partie contient le cœur de la configuration du processus de transformation (build) du projet en artefact livrable

```
<build>
  <sourceDirectory>(...)solutions\pom-minimaliste\src\main\java</sourceDirectory>
  <scriptSourceDirectory>(...)solutions\pom-minimaliste\src\main\scripts</scriptSourceDirectory>
  <testSourceDirectory>(...)solutions\pom-minimaliste\src\test\java</testSourceDirectory>
  <outputDirectory>(...)solutions\pom-minimaliste\target\classes</outputDirectory>
  <testOutputDirectory>(...)solutions\pom-minimaliste\target\test-classes</testOutputDirectory>
  <resources> (...) </resources>
  <testResources> (...) </testResources>
  <directory>(...)solutions\pom-minimaliste\target</directory>
  <finalName>maven-example-1</finalName>
  <pluginManagement> (...)</pluginManagement>
  <plugins> (...)</plugins>
</build>
```

LE POM (MODÈLE OBJET DE PROJET)

Contenu du POM – Build

+ **sourceDirectory**

- ▀ Où se trouve le code source à compiler

+ **scriptSourceDirectory**

- ▀ Si le projet contient des scripts (sh, bat...), indique où se trouvent ceux-ci

+ **testSourceDirectory**

- ▀ Où se trouve le code source des tests unitaires à compiler

+ **directory**

- ▀ Dans quel répertoire de travail temporaire doivent être déposés les fichiers résultants du build

+ **outputDirectory**

- ▀ Où est déposé le résultat de la compilation des sources

+ **testOutputDirectory**

- ▀ Où est déposé le résultat de la compilation des sources des tests

LE POM (MODÈLE OBJET DE PROJET)

Contenu du POM – Build

+ **resources**

- ▀ Où se trouvent les ressources non compilables du projet (fichier de configuration, images, ...)

+ **testResources**

- ▀ Où se trouvent les ressources non compilables spécifiques aux tests.

+ **finalName**

- ▀ Le résultat d'un build maven est un fichier (dépendant du type de packaging choisit, vu dans un prochain chapitre) dont le nom sera celui précisé par cette balise.

+ Ex dans notre cas : maven-example-1

LE POM (MODÈLE OBJET DE PROJET)

Contenu du POM – Build

+ pluginManagement et plugin

- ▀ Ces sections servent à lister et configurer les plugins maven participants au build du projet. Cette section sera abordé dans le chapitre sur le cycle de vie.

LE POM (MODÈLE OBJET DE PROJET)

Contenu du POM – Propriétés

- + Si l'on observe le POM effectif, on remarque que certaines valeurs (répertoire courant du projet par exemple) sont bien « insérées » dans des éléments prédéfinis par le Super POM
- + Afin de permettre cette « injection » de valeur dans un POM (Super POM ou non !), maven dispose d'une mécanisme de propriétés dynamiques (similaire aux variables d'un langage de templating comme JSP/EL)
- + Ces propriétés sont référencées dans les POM par `${ nom_de_la_propriété }` et sont remplacées dynamiquement lors de la phase de « merge » des POM pour obtenir le POM effectif.

LE POM (MODÈLE OBJET DE PROJET)

Contenu du POM – Propriétés

+ Maven prédéfinit un certain nombre de propriétés par défaut.

Par exemple

- ▶ `project.basedir` : répertoire du projet où se trouve le fichier `pom.xml`
- ▶ `project.build.directory` : répertoire où sont déposés les fichiers de travail du build
- ▶ `project.version` : version du projet
- ▶ ...

+ Exemple d'utilisation : la valeur par défaut de `<finalName>` est définie dans le Super POM de la façon suivante

- ▶ `<finalName>${project.artifactId}-${project.version}</finalName>`

+ Pour la liste complète des propriétés accessibles :

- ▶ <http://books.sonatype.com/mvnref-book/reference/resource-filtering-sect-properties.html>

LE POM (MODÈLE OBJET DE PROJET)

Contenu du POM – Eléments complémentaires

+ Le POM peut contenir d'autres éléments informatifs :

- ▶ Liste des développeurs
- ▶ Liste des contributeurs
- ▶ Référence au gestionnaire de sources
- ▶ MailingList...

+ Ces éléments seront abordés au besoin en fonction des aspects de maven étudiés

- ▶ Vous retrouverez l'ensemble des balises dans la documentation officielle

+ Introduction

+ Le POM

+ Dépendances d'un projet

- Introduction
- Dépendances directes et transitives
- Repositories
- Les scopes
- Résolution des conflits
- Gestion des versions centralisées

+ Relations entre projets

+ Les bonnes pratiques du POM

+ Il est extrêmement rare pour les projets de développement de ne pas dépendre de librairies externes

- Librairie de logging, spring, hibernate ...

+ Historiquement, les dépendances étaient :

- téléchargées à la main
- déposées dans un répertoire de la machine de chaque développeur
- Ajoutées au classpath du projet dans l'éditeur de code
- ...

+ A cette gestion s'ajoutait la difficulté des dépendances des dépendances...

- J'ai besoin de spring, mais spring a besoin de ... qui a besoin de ...

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet

- + **Maven simplifie radicalement ce problème**
- + **On ajoute dans le POM de notre projet la liste des dépendances directes de celui-ci, sous la forme :**

```
<dependencies>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.9</version>
  </dependency>
  ...
</dependencies>
```

- + **On retrouve la nomenclature d'identification utilisée par notre propre projet : groupId, artifactId et version**

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet - repositories

- + **Lors d'un build, maven va automatiquement récupérer les dépendances listées manquantes**

- Si elle n'ont pas déjà été récupérées précédemment

- + **Les dépendances des dépendances sont elles-aussi récupérées**

- On parle de **dépendances transitives**
- Les dépendances étant en général des projet « mavenisés » qui ont pris soin de définir dans leur propre pom la liste de leurs dépendances

- + **Maven interroge un ou plusieurs **repository** pour tenter de retrouver / télécharger les dépendances**

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – repository local

+ Le repository local est d'abord examiné pour déterminer si la dépendance est présente sur la machine ou s'effectue le build

- Un repository local n'est autre qu'un répertoire sur le disque dur
- Par défaut, ce répertoire est situé dans `$HOME/.m2/repository`
- A l'intérieur de ce répertoire, on trouve, organisés dans une arborescence basée sur les groupId/artifactID, les fichiers binaires des dépendances (jar, war, ...)

+ Ex :

- `org.hibernate:hibernate-core:3.0 =>`
- `$HOME/.m2/repository/org/hibernate/hibernate-core/3.0/hibernate-core-3.0.jar`

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – repository distant

+ Si la dépendance n'est pas trouvée en local, maven la télécharge à partir d'un repository distant

- Un repository distant est une arborescence similaire à celle d'un repository local, exposée sur internet à travers un serveur de fichier en http
- Par défaut, le Super POM définit un repository distant central

```
<repositories>
<repository>
  <id>central</id>
  <name>Central Repository</name>
  <url>http://repo.maven.apache.org/maven2</url>
  <layout>default</layout>
  <snapshots>
    <enabled>false</enabled>
  </snapshots>
</repository>
</repositories>
```

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Ajout de repository

+ On peut au besoin ajouter d'autres repositories :

- Tous les projets open-source ne publient pas forcément leur librairies sur le repository central
- Si l'on souhaite par exemple mettre en place un repository propre à l'entreprise, ou mettre un proxy de repository pour diminuer le trafic internet

+ La section <repositories> (évoquée précédemment) du POM permet d'en ajouter au niveau du projet

+ Rappel : maven distingue les versions stables des versions instables (SNAPSHOT)

- On peut indiquer lors de l'ajout d'un repository si celui-ci doit être examiné lors d'une recherche de librairie instable ou non
- Car en pratique, on sépare souvent les repository stables de ceux dédiées aux versions snapshot

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Plugins

+ Les repositories ne sont pas réservés aux dépendances

+ Maven n'est qu'un micro noyau ; l'essentiel des opérations effectuées lors d'un build sont effectuées par des plugins externes...

- Les plugins ne sont pas inclus dans la distribution de maven par défaut

+ Ils sont téléchargés au besoin lorsqu'ils sont invoqués, comme les dépendances

- Repository local et repositories distants collaborent de la même façon pour récupérer les plugins manquants
- Les plugins sont donc identifiés comme les dépendances par leur groupId:artifactId:version

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Plugins

+ Maven distingue dans la configuration les repositories distants dédiés aux dépendances de ceux dédiés aux plugins

+ Une section du POM est réservée à la déclaration des repositories des plugins

```
<pluginRepository>
  <id>central</id>
  <name>Central Repository</name>
  <url>http://repo.maven.apache.org/maven2</url>
  <layout>default</layout>
  <snapshots>
    <enabled>false</enabled>
  </snapshots>
  <releases>
    <updatePolicy>never</updatePolicy>
  </releases>
</pluginRepository>
```

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Les scopes

+ Nos dépendances peuvent être variées et concerner différents aspects de notre produit

- Dépendances pour les tests unitaires
- Dépendances pour le runtime (driver jdbc...)

+ Il est possible de préciser des **scopes pour affiner la nature de la dépendance**

- Balise `<scope></scope>` dans une `<dependency>`

+ Maven distingue 5 scopes prédéfinis :

- Compile
- Provided
- Runtime
- Test
- System

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Scope Compile

+ Le scope **compile** est le scope par défaut

- Les dépendances de ce scope sont accessibles à la compilation des sources, des sources de tests, à l'exécution des programmes et des tests, et sont packagées lors du build si le type de packaging le permet (war, par exemple)

+ Le scope **provided**

- Indique que la dépendance est nécessaire à la compilation mais que l'environnement d'exécution du futur programme contient déjà la dépendance. Celle-ci ne sera donc pas packagée avec l'artifact du projet.
 - Par exemple, l'api Servlet de jEE ; nécessaire à la compilation des servlets, mais déjà présente sur le serveur et ne nécessite donc pas d'être livrée avec l'application

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Scope Compile

+ Le scope **runtime**

- La dépendance est nécessaire à l'exécution, mais pas à la compilation (driver jdbc par exemple). La librairie est packagée avec l'application.

+ Le scope **test**

- La dépendance de ce scope n'est disponible qu'à la compilation et l'exécution des tests automatisés. Les librairie n'est pas packagée avec l'application.

+ Le scope **System**

- Est utilisé pour désigner une librairie native spécifique à un environnement (dll windows par exemple), qui doit être présente sur la machine de build. Maven ne cherche pas la dépendance dans les repositories. Il faut indiquer le chemin local vers la librairie.

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Scopes et dépendances transitives

+ Le choix du scope influe sur la transitivité.

+ Rappel : Une dépendance transitive est une dépendance d'une dépendance

- ▀ Si votre projet dépend de A, et que A dépend de B, alors B est une dépendance transitive de votre projet.

+ Le tableau suivant indique l'impact en matière de transitivité

- ▀ Au croisement des scopes est indiqué le scope pris en compte
- ▀ Une case vide indique que la dépendance transitive est ignorée

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Scopes et dépendances transitives

Le scope choisi par mon projet pour la dépendance A	Scope final des dépendances transitives en fonction du scope choisi par A			
	compile	provided	runtime	test
compile	compile	-	runtime	-
provided	provided	-	provided	-
runtime	runtime	-	runtime	-
test	test	-	test	-

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Résolution des conflits

+ Des conflits de versions peuvent survenir

- ▀ Si l'on spécifie une dépendance directe vers A en version X mais qu'une dépendance transitive vers A nécessite une version Y

+ Ces conflits de versions vont générer un ajout multiple d'une même dépendance, en plusieurs versions différentes

- ▀ Cela peut ne pas avoir d'effet (immédiat...) sur le projet
- ▀ Mais tôt ou tard, des problèmes de chargement de classes surviendront

+ A vérifier avec

- ▀ mvn dependency:tree -Dverbose

+ Pour résoudre le problème, plusieurs possibilités

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Résolution des conflits - Exclusion

+ Exclure une dépendance transitive

- ▀ Il faut indiquer dans le pom, dans la dépendance à l'origine de la transitive fautive, la balise < exclusions>

```
<dependency>
  <groupId>org.springframework.idap</groupId>
  <artifactId>spring-idap-core</artifactId>
  <version>2.3</version>
  <exclusions>
    <exclusion>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Résolution des conflits - Exclusion

- + Une dépendance transitive exclue ne sera plus tirée par le projet
- + On exclue en général les dépendances transitives en version plus ancienne que celle que l'on utilise sur le projet

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Résolution des conflits – Dependency management

- + Un autre moyen beaucoup plus puissant est d'utiliser la balise `<dependencyManagement>`
 - ▀ Cette balise, à rajouter dans le pom du projet, permet de préciser quelle version doit être prise en compte pour une dépendance donnée
 - + Cela fixe le numéro de version pour les dépendances transitive ET directes !

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Résolution des conflits – Dependency management

```
<?xml version="1.0" encoding="UTF-8"?>
<project>

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.sqli.training</groupId>
  <artifactId>maven-example</artifactId>
  <version>1</version>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aop</artifactId>
        <version>3.5</version>
      </dependency>
    </dependencies>
  </dependencyManagement>

</project>
```

LE POM (MODÈLE OBJET DE PROJET)

Dépendances d'un projet – Résolution des conflits – Dependency management

- + Il est ensuite inutile pour les dépendances directes de préciser le numéro de version fixé dans le dependencyManagement**
- + Et les dépendances transitives sont automatiquement récupérées dans la version managée !**
- + Conclusion : utilisez le dependencyManagement sur vos projets !**

LE POM (MODÈLE OBJET DE PROJET)

TP – Ajout de dépendances et résolution des conflits

+ Pré-requis :

- Maven installé
- Un pom avec un packaging de type war (à créer au besoin)

+ Objectif :

- Ajoutez les dépendances suivantes à votre projet :
 - + commons-beanutils:commons-beanutils:jar:1.9.1
 - + struts:struts:jar:1.2.8
- Effectuer une construction du projet et observez la multiplication des versions
 - + mvn dependency:tree -Dverbose
- Corrigez le problème de conflits de version en utilisant ce qui a été vu dans le cours

LE POM (MODÈLE OBJET DE PROJET)

Sommaire

- + Introduction
- + Le POM
- + Syntaxe de POM
- + Dépendances d'un projet
- + **Relations entre projets**
 - Projets multi-modules
 - Héritage de projet
- + Les bonnes pratiques

LE POM (MODÈLE OBJET DE PROJET)

Relations entre projets

+ Nous n'avons pour l'instant parlé que de projet simple « monolithique »

- ▀ Le projet / librairie est autonome et n'a qu'un usage unique ; il n'est pas utile de le découper en plusieurs parties distinctes pouvant être utilisées séparément
- ▀ Le projet ne s'inscrit pas dans un cadre plus vaste (DSI), imposant des contraintes sur les librairies, plugins...

+ Maven propose des mécaniques pour répondre aux deux aspects évoqués ci-dessus :

- ▀ Packager un projet en plusieurs artefacts (plusieurs jar, un jar et un war ...)
- ▀ Suivre des règles venant « de plus haut »

LE POM (MODÈLE OBJET DE PROJET)

Relations entre projets – projet multi modules

+ Pour maven, 1 POM = 1 Artefact généré

- ▀ C'est une critique faite par les détracteurs de maven
- ▀ Mais qui en pratique est saine et évite les build ampoulés

+ Si l'on souhaite packager un projet en plusieurs artefacts, il faut découper le projet en plusieurs modules

+ Un module est essentiellement un sous projet / sous répertoire

- ▀ Par exemple, pour le projet ProjetXYZ
 - \ProjetXYZ
 - \Module1
 - \Module2

+ Chaque module doit contenir un fichier pom.xml

LE POM (MODÈLE OBJET DE PROJET)

Relations entre projets – projet multi modules

+ Chaque module est donc considéré comme un projet à part entière

- On peut alors exécuter des commandes maven dans chacun des modules indépendamment
- Chaque module doit avoir un artifactId différent : le groupId est en général le même puisqu'il s'agit du même projet
- On peut bien sûr déclarer des dépendances entre modules de la façon standard (<dependency>)

+ C'est d'ailleurs obligatoire, les modules ne pouvant se 'voir' automatiquement entre eux, même au sein d'un même projet

+ Il est cependant fastidieux de devoir construire chaque module indépendamment, s'il s'agit du même projet

LE POM (MODÈLE OBJET DE PROJET)

Relations entre projets – projet multi modules

+ Pour résoudre ce problème, le pom du dossier racine du projet multi-module peut être configuré de façon à jouer le rôle de pom agrégateur

- Son packaging est choisi à 'pom' (nous reviendrons sur ce packaging)
- Il faut ajouter la balise <modules> et lister les modules que l'on souhaite construire en même temps que le pom agrégateur avec la sous balise <module>
- Le nom du module correspond au nom du répertoire du sous-module, et non son artifactId
 - + Une bonne convention est cependant d'adopter artifactId = nom du répertoire du module
 - + Module non listé = module non construit

LE POM (MODÈLE OBJET DE PROJET)

Relations entre projets – projet multi modules

```
<?xml version="1.0" encoding="UTF-8"?>
<project>

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.sqli.training</groupId>
  <artifactId>maven-multimodule-root</artifactId>
  <version>1</version>
  <packaging>pom</packaging>
  ...
  <modules>
    <module>model</module>
    <module>service</module>
  </modules>

</project>
```

LE POM (MODÈLE OBJET DE PROJET)

Relations entre projets – projet multi modules

+ Il est tout à fait possible de constituer des modules / sous-modules de façon récursive

- ▀ Certains projets complexes possèdent plusieurs niveaux de modules / sous-module avec des pom agrégateurs à différents étages

LE POM (MODÈLE OBJET DE PROJET)

Relations entre projets – héritage

+ La création de plusieurs pom pour les différents modules d'un projet génère en général beaucoup de copier-coller

- ▀ Des dépendances communes à tous les modules...
- ▀ Des configurations de plugins identiques...

+ Maven offre un mécanisme simple pour régler ce problème : l'héritage de pom

+ Souvenez-vous du pom effectif : celui-ci est la résultante de la fusion du Super POM et du POM projet

- ▀ Il s'agit en réalité d'un héritage implicite !

LE POM (MODÈLE OBJET DE PROJET)

Relations entre projets – héritage

+ Pour réaliser un héritage entre POM, il faut utiliser la balise <parent> dans le POM enfant

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.sqli.training</groupId>
    <artifactId>maven-multimodule-root</artifactId>
    <version>2.2.0-SNAPSHOT</version>
  </parent>

  <artifactId>maven-multimodule-webapp</artifactId>

</project>
```

LE POM (MODÈLE OBJET DE PROJET)

Relations entre projets – héritage

+ Le POM parent doit par défaut se trouver dans le répertoire parent du POM enfant

- ▀ Il est possible de changer cet emplacement avec la balise `<relativePath>`
- ▀ Il faut préciser le numéro de version du parent

+ Il n'est alors plus nécessaire de préciser le numéro de version ou le groupId dans le POM enfant

- ▀ Ils sont hérités du parent.

LE POM (MODÈLE OBJET DE PROJET)

Relations entre projets – héritage

+ Le POM effectif du POM enfant est alors le résultat de la fusion du Super POM, du POM parent et du POM enfant

+ L'utilisation de l'héritage est très utile pour factoriser les éléments communs

- ▀ Dépendances communes
- ▀ DependencyManagement commun dans le parent permettant de contrôler l'ensemble des version des modules enfants et de résoudre les conflits
- ▀ Plugins et configurations des plugins

+ Le POM parent joue très souvent aussi le rôle de POM agrégateur global

- ▀ **Attention à ne pas confondre les deux notions !**

LE POM (MODÈLE OBJET DE PROJET)

Relations entre projets – héritage – POM d'entreprise

+ Certaines organisations / DSI ont élaboré un POM d'entreprise, duquel tout projet doit hériter

- ▀ votre propre POM parent doit hériter de ce POM d'entreprise

+ Cette pratique vise à standardiser un certain nombre d'éléments au sein d'une organisation

- ▀ Bibliothèques, frameworks autorisés, en quelle version...
- ▀ Profils de builds standardisés (vu dans un prochain chapitre)
- ▀ Simplification de l'intégration des projets dans l'usine logicielle
 - + Configuration de plugins spécifiques
- ▀ ...

LE POM (MODÈLE OBJET DE PROJET)

TP – Pom parent, pom aggregateur

+ Pré-requis :

- ▀ Maven installé

+ Objectif :

- ▀ Créer un projet multi module
 - + Choisissez votre groupId, votre artifactId et votre version
 - + Créer un pom parent qui soit aussi un pom aggregateur
 - + Créez deux modules enfants qui doivent être construits tous les deux lors de la construction du parent
 - Le type de module sera de type jar
 - + Déclarez des dépendances pour les deux modules enfants
 - junit:junit:4.11 (scope test)
 - org.slf4j:slf4j-api (scope compile)
 - ch.qos.logback:logback-classic (scope runtime)
 - + Examinez les dépendances des différents modules avec la commande vue précédemment

LE POM (MODÈLE OBJET DE PROJET)

Sommaire

- + Introduction
- + Le POM
- + Syntaxe de POM
- + Dépendances d'un projet
- + Relations entre projets
- + **Les bonnes pratiques du POM**

LE POM (MODÈLE OBJET DE PROJET)

Bonnes pratiques

- + **Gérer les versions avec le mécanisme de dependency management**
 - Pas de numéro de version directement dans une dependency !
 - Pensez à vérifier les dépendances transitives et à vous assurer régulièrement au fur et à mesure de l'ajout de librairies sur les projets qu'aucun conflit de version n'apparaît
- + **Grouper les dépendances par responsabilités et identifier avec des commentaires ces responsabilités**
 - Librairies de logging
 - Librairies de persistance
 - Librairies destinées aux tests...

- + **Conservez une arborescence de répertoire respectant la hiérarchie de l'héritage**

- Il est possible de ne pas le faire mais cela amène de la confusion

- + **ArtifactId d'un module = nom du répertoire**

- + **ArtifactId du POM parent = « nom du projet »-parent**

- + **Ne changez pas les valeurs par défaut des emplacements de répertoire srcDirectory, testDirectory...**

- Convention over configuration !

- + Même si l'on peut changer la configuration....



CYCLE DE VIE DU BUILD

Sommaire

- + Introduction**
- + Cycle de vie par type de packaging**
- + Focus sur les Goals principaux**

CYCLE DE VIE DU BUILD

Sommaire

- + Introduction**
 - ▀ Principe du cycle de vie
 - ▀ Plugins, goals et alias
 - ▀ Cycle de vie Clean
 - ▀ Cycle de vie Default
 - ▀ Cycle de vie Site
- + Cycle de vie par type de packaging**
- + Focus sur les Goals principaux**

+ Nous avons jusqu'à présent parlé des projets, de leurs dépendances, de leurs organisations en modules et de leur relations inter-modules

- C'est le côté « statique » du modèle

+ Intéressons nous maintenant aux aspects dynamiques

- Que se passe t'il lors d'un build ?
- Que signifie « build » d'ailleurs ?
- Quelle partie de Maven effectue les tâches du « build » ?
- Comment adapter un build ?

+ Un build est appelé lifecycle

- Maven définit 3 lifecycles standards
 - + Clean
 - + Default (appellé aussi par abus de langage « build »)
 - + Site

+ Un lifecycle est un ensemble de phases

+ A une phase est rattaché un ensemble de goals

+ Les goals sont réalisés par les plugins maven

+ Dans la plupart des cas, déclencher « un build » équivaut à invoquer un lifecycle complet ou faire appel à une phase précise d'un lifecycle

- mvn install (invocation de la phase install du cycle de vie par défaut)
- mvn clean (invocation du cycle de vie clean)
- mvn clean install (invocation du cycle de vie clean puis de la phase install du cycle de vie par défaut)

+ Déclencher un cycle de vie va déclencher l'ensemble des phases de ce cycle

+ A chaque phase peut correspondre une ou plusieurs « action »

+ Les actions sont réalisées par les goals des plugins maven

- Goal === Mojo en jargon maven

+ Le rattachement des goals aux phases des lifecycles s'effectue dans les pom, via les balises <plugin> et <execution>

- Nous en verrons une illustration plus loin
- Attention ! Techniquement, on ne scripte pas un build comme on le ferait avec Ant, on rattache l'exécution de fonction unitaires à des phases précises d'une plan de construction pré-défini

+ Bien que le déclenchement de plugin soit dans la plupart des cas géré à travers les cycles de vie, on peut aussi invoquer directement le goal d'un plugin

+ Appeler un goal d'un plugin revient en quelque sorte à appeler une fonction d'un objet en programmation :

▀ Ex java : `accountService.loadAllAccounts()`

+ Problématique du plugin

- ▀ En programmation, j'ai l'instance d'objet à ma disposition et j'appelle ses fonctions
- ▀ En maven, à quoi correspond l'instance de plugin ?

+ Pour accéder à « l'instance » de plugin et appeler une « fonction », il faut en théorie utiliser la convention de nommage suivante :

- ▀ `plugin-groupId : plugin-artifactId : plugin-version : goal`
- ▀ Ex : `org.apache.maven.plugins:maven-clean-plugin:2.1:clean`
 - + Maven « instancie » la version précise du plugin spécifiée et appelle le goal précisé (ici, `clean`)

CYCLE DE VIE DU BUILD

Plugins, goals et alias

+ En pratique, cette syntaxe groupId:artifactId est trop verbeuse !

+ Maven propose plusieurs mécaniques pour simplifier cette invocation

- Gestion d'alias
- Gestion automatique des versions

CYCLE DE VIE DU BUILD

Plugins, goals et alias

+ Alias : maven tente de déduire un nom de plugin en appliquant une règle de nommage automatique à partir de l'alias utilisé

- Selon le pattern suivant
 - + Pour les plugins 'officiels' maven : maven-`${alias}`-plugin
 - + Pour les plugins non officiels : `${alias}`-maven-plugin
 - + Ainsi :
 - 'clean' correspond potentiellement à org.apache.maven.plugins:maven-clean-plugin
- Si aucun plugin ne correspond, les autres stratégies sont appliquées

+ Alias : On définit soit même un alias dans le pom, par plugin, via goalPrefix

```
<plugin>
  <artifactId>maven-clean-plugin</artifactId>
  <version>2.3</version>
  <configuration>
    <goalPrefix>myPrefix</goalPrefix>
  </configuration>
</plugin>
```

+ Alias : On peut aussi définir son alias dans un fichier **maven-metadata.xml**, qui est déposé sur le repository distant

- ▀ Un fichier par groupId, déposé dans le répertoire du dit groupId

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
  <plugins>
    <plugin>
      <name>Maven ACR Plugin</name>
      <prefix>acr</prefix>
      <artifactId>maven-acr-plugin</artifactId>
    </plugin>
    <plugin>
      <name>Maven Ant Plugin</name>
      <prefix>ant</prefix>
      <artifactId>maven-ant-plugin</artifactId>
    </plugin>
  </plugins>
</metadata>
```

CYCLE DE VIE DU BUILD

Plugins, goals et alias

+ Version : On peut l'omettre. En l'absence de numéro de version, maven utilise

- ▀ la version définie dans la section `pluginManagement` du pom effectif
- ▀ Ou la version la plus récente du plugin disponible
 - + Vérification dans les repository distants, puis dans le repository local

CYCLE DE VIE DU BUILD

Clean lifecycle

+ Revenons maintenant sur les cycles de vie

+ Le cycle de vie **clean** est le plus simple

- ▀ `mvn clean`
- ▀ Il vide le répertoire de travail généré par maven lors de l'invocation des autres lifecycles

+ Il est constitué de 3 phases, et invoque les goals suivants

- ▀ Pre-clean
 - + N'est liée à aucun goal par défaut
- ▀ Clean
 - + Plugin 'maven-clean-plugin' (alias : clean), goal clean
- ▀ Post-clean
 - + N'est liée à aucun goal par défaut

CYCLE DE VIE DU BUILD

Clean lifecycle

- + **Pre-clean et Post-clean** sont des phases du cycle de vie qui existent pour vous permettre de rattacher des goals spécifiques à votre build
- + La phase **Clean** est liée au plugin **maven-clean-plugin**, et plus précisément à son goal **clean**
- + Ainsi, les commandes suivantes vont, par défaut, toutes aboutir au même résultat concret
 - `mvn clean (lifecycle clean)` => invoquera le plugin clean lors de la phase liée
 - `mvn maven-clean-plugin:clean` (appel direct au plugin sans passer par un lifecycle / une phase)
 - `mvn clean:clean` (appel direct au plugin en utilisant son alias)

CYCLE DE VIE DU BUILD

Clean lifecycle

- + Si l'on observe le pom effectif de notre projet, on trouve le **rattachement explicite du goal clean du plugin maven-clean-plugin à la phase clean**

```
<plugins>
<plugin>
  <artifactId>maven-clean-plugin</artifactId>
  <version>2.4.1</version>
  <executions>
    <execution>
      <id>default-clean</id>
      <phase>clean</phase>
      <goals>
        <goal>clean</goal>
      </goals>
    </execution>
  </executions>
</plugin>
</plugins>
```

+ Vous pouvez, selon ce principe, rattacher autant d'exécution de plugin et de goals aux phases définies par les lifecycles

- ▀ Ceci est bien sûr valable pour tous les lifecycles

+ Attention à la syntaxe d'invocation permissive :

- ▀ Ne pas confondre lifecycle, plugin, alias de plugin...
 - + On confond souvent 'mvn clean' et 'mvn clean:clean' qui sont pourtant des choses très différentes (même si le résultat – suppression du répertoire de travail /target – est le même) !

+ Le cycle par défaut est lui plus complexe

+ Il est constitué des phases suivantes :

- ▀ validate
 - ▀ initialize
 - ▀ generate-sources
 - ▀ process-sources
 - ▀ generate-resources
 - ▀ process-resources
 - ▀ compile
 - ▀ process-classes
 - ▀ generate-test-sources
 - ▀ process-test-sources
 - ▀ generate-test-resources
- (suite) →

CYCLE DE VIE DU BUILD

Default lifecycle

- ▀ process-test-resources
- ▀ test-compile
- ▀ process-test-classes
- ▀ test
- ▀ prepare-package
- ▀ package
- ▀ pre-integration-test
- ▀ integration-test
- ▀ post-integration-test
- ▀ verify
- ▀ install
- ▀ deploy

CYCLE DE VIE DU BUILD

Default lifecycle

- + Contrairement au cycle de vie clean, aucun binding n'est pré-défini pour les phases du default lifecycle**
- + C'est le choix d'un **type de packaging** pour notre projet qui liera certains plugins aux phases présentées précédemment**
 - ▀ Le type de packaging est abordé au chapitre suivant

+ Enfin, le lifecycle **site est dédié à la génération de documentation**

- ▀ Site internet « résumé » du projet, javadoc, métriques et rapport d'analyses statiques et dynamiques du code source...

+ Il est constitué de 4 phases et des goals suivants

- ▀ pre-site
- ▀ site
 - + plugin:goal = site:site
- ▀ post-site
- ▀ site-deploy
 - + plugin:goal = site:deploy

+ Introduction

+ Cycle de vie par type de packaging

- ▀ Jar
- ▀ Pom
- ▀ War
- ▀ Ear
- ▀ Autres packaging

+ Les Goals

- + Le choix du type de packaging va déterminer les plugins qui seront rattachés aux différentes phases du default lifecycle
- + Nous avons rapidement abordé le principe de « packaging » d'un projet maven avec le pom parent
 - ▀ Packaging : pom
 - ▀ Balise <packaging>
- + On distingue principalement les types de packaging suivants
 - ▀ Jar
 - ▀ Pom
 - ▀ War

- + La packaging Jar est le packaging par défaut lorsqu'il n'est pas précisé
- + Ce packaging affecte les goals des plugins suivants aux phases :

Phase	Plugin alias : goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

+ Il faut se référer à la documentation des plugins pour avoir une description précise de leur utilité

+ En quelques mots :

- ▀ resources:resources
 - + Copie des ressources du projet
- ▀ compiler:compile
 - + Compilation des .java en .class
- ▀ resources:testResources
 - + Copie des ressources de tests
- ▀ compiler:testCompile
 - + Compilation des .java de test en .class
- ▀ surefire:test
 - + Execution des tests unitaires

- ▀ jar:jar
 - + Création du jar de l'application : .class et ressources
- ▀ install:install
 - + Installation du jar de l'application dans le repository maven local
- ▀ deploy:deploy
 - + Installation du jar de l'application dans un repository maven distant

+ Le packaging le plus simple : se contente de publier le pom.xml comme artifact produit !

+ Ce packaging affecte les goals des plugins suivants aux phases :

Phase	Plugin alias : goal
package	site:attach-descriptor
install	install:install
deploy	deploy:deploy

+ Le packaging war est le packaging dédié aux web applications jee

+ Ce packaging affecte les goals des plugins suivants aux phases :

Phase	Plugin alias : goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	war:war
install	install:install
deploy	deploy:deploy

+ On remarque de nombreuses similitudes avec le packaging jar

+ Différence notable : la façon dont va être packagée l'application

- ▀ Au lieu de produire un jar, le build va produire un war
- ▀ Le war contiendra le code compilé, les sources des pages html, jsp, css, js... ainsi que les dépendances en scope compile et system (et les transitives du même scope, cf. tableau sur la gestion des scopes et des transitives)
- ▀ Se référer à la documentation du maven-war-plugin pour une description détaillée

+ Il existe et il est possible de définir de nouveaux packaging

- ▀ Ex : ear, ejb, flex, nbm...
- ▀ Le principe reste le même ; la consultation des plugins liés est nécessaire à l'appropriation d'un lifecycle

+ Définir son propre packaging est hors du scope de la formation

- + Introduction
- + Cycle de vie par type de packaging
- + **Focus sur les Goals principaux**
 - Rappel : phases et goals
 - Traiter les ressources
 - Compilation
 - Traiter les ressources des tests
 - Compilation des tests
 - Tester
 - Installer l'artefact
 - Déploiement

- + **Nous avons vu que les goals représentent les actions du build et sont réalisés par les plugins maven**
- + **Ces goals sont rattaché à des phases des cycles de vie maven**
- + **Pour le cycle de vie par défaut, les goals vont varier en fonction du packaging projet choisi**
- + **Certains goals / plugins sont invoqués par plusieurs packaging différents**
 - La réutilisabilité / granularité des plugins maven est importante
- + **Nous allons maintenant étudier certains de ces plugins, incontournables dans l'usage de maven**
 - Nous allons les présenter dans l'ordre de leur intervention lors d'un build lifecycle par défaut

+ Le traitement des ressources projet correspond à la phase process-resources

+ Plugin :

- org.apache.maven.plugins: maven-resources-plugin

+ Le traitement des ressources implique

- Copie des fichiers « /src/main/resources » dans « target/classes »
 - + Rappel : il s'agit des répertoires pré-définis dans le Super POM
- Lors de la copie, filtrage du contenu des fichiers et remplacement des tokens délimités par \${...} si le filtering est activé
 - + Nous reviendrons sur ce mécanisme dans un chapitre dédié

+ Attention à l'encoding des ressources !

- UTF-8, ISO... le plugin resources doit être configuré pour utiliser le bon encoding
 - + Là aussi, nous y reviendrons

+ La compilation va compiler le code source java en .class

+ Plugin

- org.apache.maven.plugins: maven-compiler-plugin
- Fait appel à la commande 'javac'

+ Attention, il considère que le code source correspond à java 1.3 et que la JVM cible est une JVM 1.1

- Il faut changer ces valeurs en configurant le plugin pour lui indiquer les bonnes versions
 - + La configuration des plugins sera ré-abordées plus en détail dans un chapitre ultérieur

```
<plugin>
<artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.5</source>
    <target>1.5</target>
  </configuration>
</plugin>
```

+ **Le traitement des ressources de tests et la compilation des sources de tests sont effectuées à l'identique**

- /src/main/resources devient src/test/resources
- /src/main/java devient src/test/java
- target/classes devient target/test-classes

+ **Maven a intégré très tôt le principe de tests automatisés en continu**

+ **A chaque build, les tests compilés dans src/test/resources sont joués**

- Bien sûr, il faut avoir écrit des tests...

+ **En cas d'échec d'un test, le build est interrompu**

+ **Sémantiquement, on distingue plusieurs catégories de test. On peut en retenir deux :**

- Test unitaires de composant (boite blanche) : TU
- Tests d'intégration (boite noire) : TI

+ **Maven nous permet de distinguer les deux catégories en liant les TU et TI à des phases différentes et à des plugins différents**

+ Tests unitaires

- Plugin : org.apache.maven.plugins:maven-surefire-plugin
- Lié à la phase `test`

+ Tests d'intégration

- Plugin : org.apache.maven.plugins:maven-failsafe-plugin
- Lié à la phase `integration-test`

+ Se référer à chaque plugin pour le détail de la configuration de celui-ci et son comportement par défaut

- Possibilité d'utiliser différents noyaux de tests automatisés (junit, testng,...)
- Configuration fine des nomenclatures de nommages des classes de tests (unitaire, intégration...)

+ La phase d'installation suit la phase de packaging

- Rappel : la phase de packaging sert à produire un livrable unique à partir des sources du projet et dépend du type de packaging choisit (war, jar ...)

+ Cette phase invoque le plugin `maven-install-plugin` qui va copier dans le repository local de la machine l'artefact produit

- En suivant le layout de répertoire `groupId / artifactId / version`

+ Installer l'artefact dans le repository local est indispensable pour « partager » un module du projet avec les autres modules

- Ex : la compilation, la construction d'un war, l'exécution de tests automatisés d'un module A d'un projet dépendant d'un module B du même projet ne pourra pas se faire si le module B n'est pas installé en local

+ La phase de déploiement est équivalente à la phase d'installation, mais sur le repository distant

- ▀ Copie de l'artefact produit par la phase package

+ Le plugin chargé du déploiement distant doit bien sûr être configuré pour pouvoir effectuer le déploiement

- ▀ Quelle est l'adresse du repository distant ?

```
<distributionManagement>
  <repository>
    <id>SQLIMavenRepository</id>
    <url>dav:http://10.0.0.1/repository/</url>
  </repository>
</distributionManagement>
```

+ Le serveur doit bien sûr accepter le transfert de fichier

- ▀ Webdav
- ▀ Scp...

+ En fonction du mode de transfert choisi, la configuration dans le pom doit-être adaptée :

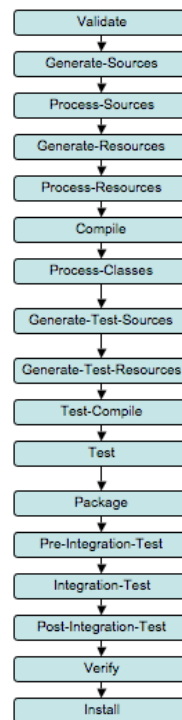
- ▀ Ajouter les librairies nécessaires (webdav, scp...)
- ▀ Configurer les credentials d'accès

+ Se référer aux ressources disponibles sur internet, en fonction du choix fait par votre organisation

- ▀ C'est une partie très variable et fluctuante d'une organisation à l'autre
- ▀ Ex ssh : <http://maven.apache.org/plugins/maven-deploy-plugin/examples/deploy-ssh-external.html>

CYCLE DE VIE DU BUILD

Les phases : illustration



LE POM (MODÈLE OBJET DE PROJET)

TP – Types de packaging

+ Pré-requis :

- Maven installé

+ Objectif :

- Ecrivez une classe java simple et un test unitaire simple dans chaque module enfant
- Ajoutez un module enfant de type war à votre projet multi-module développé dans le TP précédent
 - + Ce projet doit déclarer des dépendances vers les deux autres modules enfants
 - + Effectuez un build du projet global et observez ce qui est construit dans les modules de chaque type



PROFILS DE BUILD

Sommaire

- + Principes et utilisation
- + Activation de profils
- + Aspects avancés

- + **Principes et utilisation**
- + Activation de profils
- + Aspects avancés

- + **Il est souvent nécessaire d'adapter un build en fonction de l'environnement cible pour lequel on construit son application**
 - ▀ Dev, recette, production, client x, client y...
- + **Maven propose une mécanique standardisée pour « adapter » le build projet et le décliner en plusieurs variantes : les profils**
- + **Un profil porte un nom (id) et est une sorte de « mini pom » dans le pom**
 - ▀ On peut y définir / redéfinir de nombreux éléments du pom

+ Ex de profil :

```
<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
    ">

  <build>...</build>
  <profiles>
    <profile>
      <id>production</id>
      <build>
        <plugins>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
              <debug>>false</debug>
              <optimize>>true</optimize>
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>
```

+ Une définition de profil peut contenir les éléments suivants :

```
<profile>
  <id></id>
  <activation></activation>
  <build>
    <defaultGoal>...</defaultGoal>
    <finalName>...</finalName>
    <resources>...</resources>
    <testResources>...</testResources>
    <plugins>...</plugins>
  </build>
  <reporting>...</reporting>
  <modules>...</modules>
  <dependencies>...</dependencies>
  <dependencyManagement>...</dependencyManagement>
  <distributionManagement>...</distributionManagement>
  <repositories>...</repositories>
  <pluginRepositories>...</pluginRepositories>
  <properties>...</properties>
</profile>
```

+ Si l'on observe les éléments que l'on peut faire figurer dans le profil, on voit que

- ▀ On peut modifier les plugins invoqués, en invoquer d'autres supplémentaire
- ▀ Ajouter des dépendances, changer les versions des dépendances
- ▀ Filtrer des ressources différentes
- ▀ Activer des sous-modules spécifiques
- ▀ ... etc

+ On pourrait presque définir l'ensemble du pom dans un profil...

+ Mais il faut tout de même se limiter au maximum les éléments spécifiques des profils

- ▀ L'absence de profil revenant à un profil par défaut qui est celui du pom lui-même.
- ▀ Les profils ne devraient contenir que des deltas mineurs

+ On peut créer autant de profils que l'on souhaite

- ▀ Chaque profil devant avoir un id unique

+ Utilisation classique

- ▀ Profil « dev », « recette », « production », adaptant le build à la plateforme ciblée lors du build

+ Pour être utilisé, un profil doit être activé

- + Principes et utilisation
- + **Activation de profils**
 - Activation explicite
 - Activation par propriétés
- + Aspects avancés

- + **Pour activer un profil, on peut le faire explicitement lors de l'invocation de maven en utilisant le nom du profil avec le flag `-P`**
 - `mvn clean install -PacceptanceTesting`
- + **Le profil dont l'id correspond sera activé et le pom effectif prendra en compte ses spécificités**
- + **On peut préciser plusieurs profils à la fois en séparant les ids par des virgules**
 - `mvn clean install -Pdev,postgresDatabase,noLdap`

PROFILS DE BUILD

Activation par propriété

+ On peut aussi définir dans la déclaration du profil des conditions d'activation automatique

- ▀ Balise <activation>

+ Plusieurs conditions d'activation automatique sont gérés

- ▀ <os>
 - + L'activation se base sur l'os de la machine de build
- ▀ <jdk>
 - + L'activation se base sur la version du jdk utilisée pour le build
- ▀ <file>
 - + Présence / absence d'un certain fichier
- ▀ <property>
 - + Présence d'une propriété d'environnement au lancement du build, spécifiée avec -D

PROFILS DE BUILD

Activation par propriété, exemples

```
<profile>
  <id>dev</id>
  <activation>
    <jdk>1.5</jdk>
    <os>
      <name>Windows XP</name>
      <family>Windows</family>
      <arch>x86</arch>
      <version>5.1.2600</version>
    </os>
    <property>
      <name>customProperty</name>
      <value>BLUE</value>
    </property>
    <file>
      <exists>file2.properties</exists>
      <missing>file1.properties</missing>
    </file>
  </activation>
</profile>
```

PROFILS DE BUILD

Activation par propriété

+ La combinaison de plusieurs critères est évaluée avec un ET logique

- ▀ Dans l'exemple précédent, tous les critères doivent être remplis pour activer le profil

+ Il est possible de détecter l'absence de présence d'une propriété système en la préfixant d'un !

```
<profile>
  <id>dev</id>
  <activation>
    <property>
      <name>!customProperty</name>
    </property>
  </activation>
</profile>
```

PROFILS DE BUILD

Sommaire

+ Principes et utilisation

+ Activation de profils

+ Aspects avancés

- ▀ activeByDefault
- ▀ Connaitre la liste des profils actifs

+ Les profils sont hérités !

- ▀ Déclaration dans le pom parent, utilisation dans un build du pom enfant

+ Il est possible d'identifier un profil « par défaut » dans la configuration d'activation

- ▀ `<activeByDefault>true</activeByDefault>`
- ▀ Il sera alors actif si et seulement si aucun autre profil **du même fichier pom.xml** n'est actif
 - + Attention à l'héritage, des profils `activeByDefault` peuvent exister à plusieurs niveaux !

+ Il peut être utile de connaître la liste des profils actifs

- ▀ `mvn help:active-profiles`

PROPRIETES ET FILTRAGE
DES RESSOURCES



+ Introduction

+ Les différentes propriétés accessibles

+ Introduction

+ Les différentes propriétés accessibles

+ Nous avons vu que lors de la phase process-resources les fichiers situés dans src/main/resources étaient copiés, et éventuellement filtrés des tokens délimités par \${...}

+ Afin que le filtrage s'effectue, il faut l'activer dans le pom du projet

▀ Il n'est pas activé par défaut pour éviter les effets de bords non voulus

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

+ Il est possible d'ajouter des répertoires de ressources au build en plus de celui par défaut

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
    </resource>
    <resource>
      <directory>src/main/xml</directory>
    </resource>
    <resource>
      <directory>src/main/images</directory>
    </resource>
  </resources>
</build>
```


+ Il est possible d'inclure / exclure certains types de fichiers par leur extension

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <includes>
        <include>*.properties</include>
      </includes>
      <excludes>
        <include>*.bat</include>
        <include>*.sh</include>
      </excludes>
    </resource>
  </resources>
</build>
```

+ Introduction

+ Les différentes propriétés accessibles

- ▀ Propriétés Projet
- ▀ Propriétés Settings
- ▀ Variables d'environnement
- ▀ Propriétés Système
- ▀ Propriétés utilisateur

+ Les Propriétés Projet

- ▶ Commencent par `project.*`
- ▶ Permettent de référencer n'importe quelle valeur du POM
 - + `project.groupId`, `project.build.finalName` etc... (respecter la hiérarchie des éléments du pom)
 - + L'ensemble des propriétés accessibles sont disponibles dans la documentation de maven
 - Par exemple pour maven 3
 - <http://maven.apache.org/ref/3.0.3/maven-model/maven.html>

+ Les Propriétés Settings

- ▶ Commencent par `settings.*`
- ▶ Permettent de référencer n'importe quelle valeur du fichier de settings maven
 - + Nous allons étudier le fichier de settings dans une prochaine section
 - + Sur le même principe que le pom, les éléments de ce fichier sont accessibles en suivant l'imbrication des balises
 - + L'ensemble des propriétés accessibles sont disponibles dans la documentation de maven
 - Par exemple pour maven 3
 - <http://maven.apache.org/ref/3.0.3/maven-settings/settings.html>

+ Les Propriétés d'environnement

- ▀ Commencent par env.*
- ▀ Permettent de référencer n'importe quelle variable d'environnement définie par la machine exécutant le build
 - + env.HOME
 - + env.MVN_HOME
 - + ...

+ Les Propriétés système

- ▀ Sont les propriétés exposées par la classe java.lang.System
- ▀ Toute variable retournée par System.getProperty() est accessible
- ▀ Quelques exemples
 - + java.version
 - + user.name
 - + ...

+ Les Propriétés utilisateur

- ▀ Sont des propriétés définies dans le pom via la balise `<properties>`

- + Les profils peuvent aussi déclarer des propriétés

- ▀ Chaque sous-balise de `<properties>` devient une propriété accessible

- + Balise `<foo>` => à référencer avec `${foo}`

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

LE POM (MODÈLE OBJET DE PROJET)

TP – Profils de builds et filtrage des propriétés 1 / 3

+ Pré-requis :

- ▀ Maven installé

+ Objectif :

- ▀ Créez un projet maven simple de type jar
 - + Dans ce projet, créer une classe java simple qui lit le contenu d'un fichier texte présent dans le classpath et l'affiche dans la console
 - Cf slide tp 3/3 pour la classe java
 - + Dans ce fichier texte, **placer une propriété qui devra être filtrée lors du build**
 - + Déclarer deux **profils**, **chacun déclarant une valeur différente pour la propriété**

+ Objectif (suite) :

- Exécutez votre programme en activant les profils et constatez le bon fonctionnement
 - Pour executer une classe java munie d'un main en maven :
 - `mvn exec:java -Dexec.mainClass=packageDeMaClasse.NomDeMaClasse`
- Testez aussi l'activation de profil automatique
 - + `activeByDefault`
 - + Si une propriété système est spécifiée
 - + ...

```
import java.io.*;

public class App {

    public static void main(String[] args) {
        try {
            System.out.println("File content is :");

            BufferedReader reader = new BufferedReader(new InputStreamReader(
                App.class.getResourceAsStream("/fileToRead"), "UTF-8")
            );
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



OPTIONS D'EXECUTION

Sommaire

- + Options de ligne de commande
- + Options avancées

- + Options de ligne de commande
- + Options avancées

+ Quelques options activable en ligne de commande peuvent être très utiles

- ▀ 'mvndebug' au lieu de 'mvn' : permet de lancer maven en mode 'remote debugging' java. L'exécution se bloque et attend qu'un debugger se connecte (comportement par défaut de mvndebug). Il suffit ensuite de connecter son ide sur la remote jvm pour déboguer l'application lancée via maven.
- ▀ -h : liste l'ensemble des propriétés maven activable en ligne de commande
- ▀ -D : permet de définir une propriété système à la volée
 - + Exemple : mvn -DmyProp=titi
- ▀ -f : permet de lancer un build en pointant sur un fichier pom.xml particulier
- ▀ -gs : permet de pointer sur un fichier de settings.xml spécifique

OPTIONS D'EXECUTION

Options de ligne de commande

- -l : spécifie un fichier de log ou écrire la sortie console du build
- -D : permet de définir une propriété système à la volée
 - + Exemple : mvn -DmyProp=titi
- -f : permet de lancer un build en pointant sur un fichier pom.xml particulier
- -o : (utile en cas de coupure réseau) lance maven en mode déconnecté ; maven n'interrogera aucun repository distant.
- -q : sortie console minimaliste
- -X : sortie console en mode debug ; très utile pour diagnostiquer divers problèmes liés au build

OPTIONS D'EXECUTION

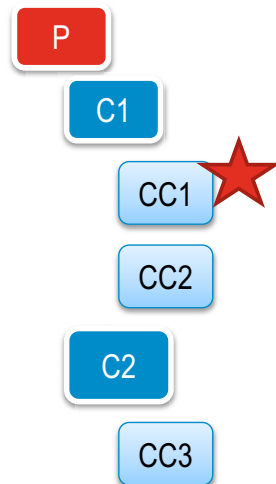
Sommaire

- + Options de ligne de commande
- + **Options avancées**
 - Reprise de build
 - Build des dépendances liées
- + Maven help

OPTIONS D'EXECUTION

Options avancées – Reprise de build

- + Si on lance un build du projet suivant, à partir du pom P (pom aggregateur) et que le build échoue à la construction de CC1, que faire une fois l'erreur fixée ?



OPTIONS D'EXECUTION

Options avancées – Reprise de build

- + Relancer le build complet à partir de P est une possibilité, mais cela nécessite de reconstruire des artifacts inutilement (ceux ayant été construits avec succès)

- + Il est possible de relancer le build à partir de P mais en précisant explicitement que l'on souhaite reprendre ce build complet à partir de CC1

- + Il faut utiliser l'option de ligne de commande `-rf` [cheminDuModule dans la hiérarchie du build]

- + Exemple :

```
cd /P
```

```
mvn clean install
```

Le build échoue à /P/C1/CC1

On corrige l'erreur (test échoué, erreur dans le pom...)

```
mvn clean install -rf /P/C1/CC1
```

OPTIONS D'EXECUTION

Options avancées – Build des projet dépendants

+ Dans un projet multi-modules, il arrive parfois que l'on modifie un module parmi d'autres et que l'on souhaite s'assurer que les modules qui dépendent de celui-ci fonctionnent toujours

- ▀ Vous avez bien sûr écrits des tests unitaires d'intégration dans ce but... ☺

+ Pour ça, il est possible de builder l'ensemble du projet à partir du pom agrégateur

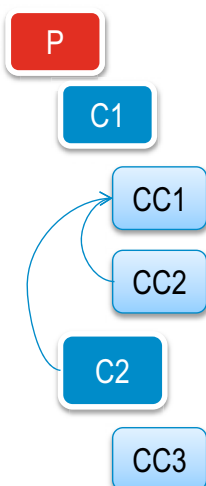
- ▀ Mais c'est là encore une perte potentielle de temps (reconstruction de modules non impactés)

OPTIONS D'EXECUTION

Options avancées – Build des projet dépendants

+ Il est aussi possible de lancer un build du module modifié et de demander à maven de builder aussi les modules qui dépendent de celui-ci automatiquement

- ▀ Option `-amd` (*also make dependants*) couplée à `-pl` (*projects list*)



> cd P
> mvn clean install -amd -pl C1/CC1

Le build construira CC1, puis CC2 et C2

→ Dépend de



CONFIGURATION AVANCÉE

Sommaire

+ Settings

- ▀ Utilité
- ▀ Cryptage des mots de passes

+ Dépendances

- ▀ Dépendances optionnelles
- ▀ Plage de versions

+ Configuration des plugins

- ▀ Configurer finement les plugins
- ▀ Ajouter des dépendances à un plugin
- ▀ Configuration spécifique à une exécution
- ▀ Activation de plugin dans les profils

+ Un fichier de configuration, **settings.xml** permet de définir des éléments de configuration globaux

- Pom : configuration au niveau projet
- Settings : configuration générale s'appliquant à tous les builds

+ Ce fichier sert à définir une configuration spécifique à la machine de build exécutant maven

- Emplacements des repository locaux
- Propriétés utilisateurs nouvelles ou surchargées pour cet environnement précis
- ...

+ Le fichier de settings n'est donc en général pas destiné à être distribué / partagé

+ Ce fichier existe à plusieurs endroits

- \$MVN_HOME/conf/settings.xml (fichier de settings au niveau maven) : appelé global settings
 - + Est chargé pour tout build, quel que soit l'utilisateur l'ayant initié
- \${user.home}/.m2/settings.xml (fichier de settings au niveau utilisateur): appelé user settings
 - + Chaque utilisateur de la machine de build peut posséder son propre settings.

+ Contenu du fichier de settings :

```
<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/>
  <interactiveMode/>
  <usePluginRegistry/>
  <offline/>
  <pluginGroups/>
  <servers/>
  <mirrors/>
  <proxies/>
  <profiles/>
  <activeProfiles/>
</settings>
```

+ Quelques propriétés importantes en détails

- ▀ Nous n'allons pas toutes les passer en revue

+ **localRepository: chemin vers le repository local.**

- ▀ Valeur par défaut : `${user.home}/.m2/repository`
- ▀ Utile de le changer pour indiquer une partition non système par exemple

+ **offline: mode offline par défaut pour tous les builds**

- ▀ Utile pour un serveur d'intégration continue par exemple ne devant jamais se connecter à des repository extérieurs

+ Proxies

- ▀ Liste les serveur proxy à utiliser pour accéder à internet

```
<proxies>
  <proxy>
    <id>myproxy</id>
    <active>true</active>
    <protocol>http</protocol>
    <host>proxy.somewhere.com</host>
    <port>8080</port>
    <username>proxyuser</username>
    <password>somepassword</password>
    <nonProxyHosts>*.google.com|biblio.org</nonProxyHosts>
  </proxy>
</proxies>
```

+ id: ID unique du serveur dans le fichier de settings

+ active

- ▀ Active / désactive un proxy

+ protocol, host, port

- ▀ Localisation du proxy sur le réseau, séparé en 3 propriétés
- ▀ Dans l'exemple précédent, adresse du proxy :
http://proxy.somewhere.com:8080

+ username, password

- ▀ En cas d'authentification requise auprès du proxy

+ nonProxyHosts

- ▀ Permet de lister les adresses ne devant pas être atteinte en passant par un proxy
 - + Ex : localhost, 127.0.0.1...

+ La balise <servers> sert à définir les informations de connexion à d'autres machines qui ne doivent pas figurer dans le pom projet

- ▀ Plugin delivery par exemple

```
<servers>
  <server>
    <id>server001</id>
    <username>my_login</username>
    <password>my_password</password>
    <privateKey>${user.home}/.ssh/id_dsa</privateKey>
    <passphrase>some_passphrase</passphrase>
    <filePermissions>664</filePermissions>
    <directoryPermissions>775</directoryPermissions>
    <configuration></configuration>
  </server>
</servers>
```

+ id: ID unique du serveur

- ▀ correspond à l'élément d'identification du référentiel / miroir

+ utilisateur, mot de passe:

- ▀ identifiant et le mot de passe requis pour l'authentification sur ce serveur.

+ privateKey, passphrase:

- ▀ Comme les deux éléments précédents, cette paire spécifie un chemin d'accès à un fichier de clé privée (par défaut \$ {user.home}/.ssh/id_dsa
- ▀ Attention : si privateKey + passphrase, ne pas préciser de password !

+ filePermissions, directoryPermissions:

- ▀ Quand un fichier ou un répertoire du référentiel est créé sur le déploiement, utiliser les permission unix précisées (permissions à la nomenclature *NIX - nombre à trois chiffres comme 664, ou 775).

CONFIGURATION AVANCÉE

Settings : cryptage des mots de passes

+ Afin d'éviter de préciser les mots de passe en clair, un outil de cryptage a été intégré à maven depuis la version 2.1.0+

- Il faut générer un mot de passe maître sur la machine de build
- Encrypter ensuite le mot de passe du serveur / proxy
- Référencer le mot de passe crypté dans le fichier de settings au lieu du mot de passe en clair
- Nous allons présenter la procédure plus en détails ci-après ; vous pouvez consulter la documentation officielle pour une présentation exhaustive :

+ <https://maven.apache.org/guides/mini/guide-encryption.html>

CONFIGURATION AVANCÉE

Settings : cryptage des mots de passes

+ Procédure de mise en œuvre

- Génération d'un master password sur la machine de build
 - + `mvn --encrypt-master-password <motDePasseMaitreEnClair>`
 - + Créer le fichier
 - `${user.home}/.m2/settings-security.xml`
 - + Et y référencer le mot de passe crypté de la façon suivante

```
<settingsSecurity>
  <master>motDePasseMaitreCrypté</master>
</settingsSecurity>
```
- Encryptage du mot de passe du server
 - + `mvn --encrypt-password <passwordServer>`
 - + Puis mise à jour de la balise password avec la version encryptée
 - `<password>{COQLCE6DU6GtcS5P=}</password>`

+ Profils

Une variante des profils de build dont nous avons parlé dans une section précédente peut aussi être utilisée dans le fichier de settings

+ Attention, ne pas mettre des profils liés à des projets dans le fichier de settings ; ces profils doivent être réservés à un usage global de maven

Il s'agit du même concept, mais dans une version simplifiée

```
<settings>
  <profiles>
    <profile>
      <id>..</id>
      <activation>...</ activation >
      <repositories>...</repositories>
      <pluginRepositories>...</pluginRepositories>
      <properties>...</properties>
    </profile>
```

+ Activation automatique de profils dans le settings

L'élément <activeprofiles> permet d'activer par défaut certains profils du fichier de settings

```
<settings>
  <activeProfiles>
    <activeProfile>env-test</activeProfile>
  </activeProfiles>
</settings>
```

CONFIGURATION AVANCÉE

Settings : pour plus d'informations

+ Consultez l'aide en ligne de maven pour le détails des autres propriétés non abordées

+ <https://maven.apache.org/settings.html>

LE POM (MODÈLE OBJET DE PROJET)

TP – Fichier de settings

+ Pré-requis :

- ▀ Maven installé

+ Objectif :

- ▀ Créer s'il n'existe pas le fichier de settings sur votre poste
- ▀ Changez l'emplacement du repository local de maven pour pointer sur un autre emplacement que \$HOME/.m2/repository
 - + Une pratique utile à faire systématiquement sur chaque poste pour éviter de remplir le disque système de centaines de mégas de jars...
- ▀ Ajouter une définition de proxy (fictive) dans le fichier prenant un login et un mot de passe d'utilisateur (fictif)
 - + Cryptez le mot de passe

+ Settings

- Cryptage des mots de passes

+ Dépendances

- Plage de versions

+ Configuration des plugins

- Configurer finement les plugins
- Ajouter des dépendances à un plugin
- Configuration spécifique à une exécution
- Activation de plugin dans les profils

+ Nous avons vu précédemment que l'on devait déclarer les numéros de version de nos dépendances

+ Nous avons aussi vu que parfois, certains artefacts pouvaient récupérer des dépendances transitives en plusieurs versions différentes

- Cette situation problématique est résolue avec le dependencyManagement

+ Maven offre une dernière mécanique concernant les numéros de versions : l'utilisation de plages de version

- C'est une pratique cependant peu répandue...
- Et potentiellement un peu risquée...

+ La plage de version nous permet de préciser que l'on souhaite une version d'une dépendance comprise entre un minima et un maxima, inclusif ou exclusif

- Inclusif : [...,...]
- Exclusif : (...,...)
- On peut mixer les deux : [...,...) ou (..., ...]
- Ex : [3.8,4.0) => je veux n'importe quelle version disponible à partir de la 3.8 et inférieur à la 4.0.

+ Maven choisit le numéro de version de façon appropriée en respectant ce qui est édicté dans le pom

+ Settings

- Cryptage des mots de passes

+ Dépendances

- Plage de versions

+ Configuration des plugins

- Configurer finement les plugins
- Ajouter des dépendances à un plugin

- + **Nous avons vu comment invoquer des plugins maven à travers les goals des cycles de vie**
- + **Les plugins liés au cycles de vies standards s'appuient sur les conventions maven et les propriétés prédéfinies du pom**
 - `project.artifactId`, etc...
- + **Il est cependant très souvent nécessaire de devoir « adapter » le comportement des plugins aux spécificités de notre projet**
 - Par exemple, le plugin compiler : quelle version de java utilisée comme cible ? 1.5 ? 1.6 ? Et quelle encoding utiliser pour copier / filtrer les ressources du projet ?

- + **Les plugins se configurent à travers la balise `<configuration>`**

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.6</source>
        <target>1.6</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

- Chaque plugin expose un certain nombre d'éléments de configuration adaptable de cette façon
- Attention, la section `build > plugins` ne sert qu'à configurer les plugins utilisables, pas à les invoquer !

+ Il est aussi possible de déclarer des dépendances spécifiques dédiées à l'exécution d'un plugin

- ▀ Driver jdbc par exemple, ou implémentation spécifiques de pool de connexion, de cache....

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>8.1.14.v20131031</version>
  <dependencies>
    <!-- Used by the datasource declared in jetty-env-->
    <dependency>
      <groupId>commons-dbcp</groupId>
      <artifactId>commons-dbcp</artifactId>
      <version>1.4</version>
    </dependency>
  </dependencies>
</plugin>
```

+ Pré-requis :

- ▀ Maven installé
- ▀ Avoir fait le tp sur les profils et le filtrage des ressources

+ Objectif :

- ▀ Réglez les problèmes d'encoding éventuellement rencontrés avec votre fichier texte en configurant l'encoding utilisé par le plugin maven des ressources



GENERATION DE SITE / RAPPORTS

Sommaire

- + Introduction
- + Construire le site d'un projet
- + Personnalisation de l'apparence
- + Documentation intégrée au site
- + Déploiement du site
- + Trucs et astuces

+ Maven peut générer pour vous un site web html complet reprenant de nombreuses informations sur le projet

- Fiche d'identité
 - + Développeurs, nom, versions....
- Résumé sur les dépendances du projet
- Rapports et métriques diverses

+ Ce site web est utile pour communiquer de façon structurée sur votre projet

- Permet de générer une documentation projet à un instant T qui peut être archivée pour une consultation future
- C'est un bon tableau de bord

+ Construire le site du projet est simple : il faut invoquer le lifecycle `site`

- `mvn site`

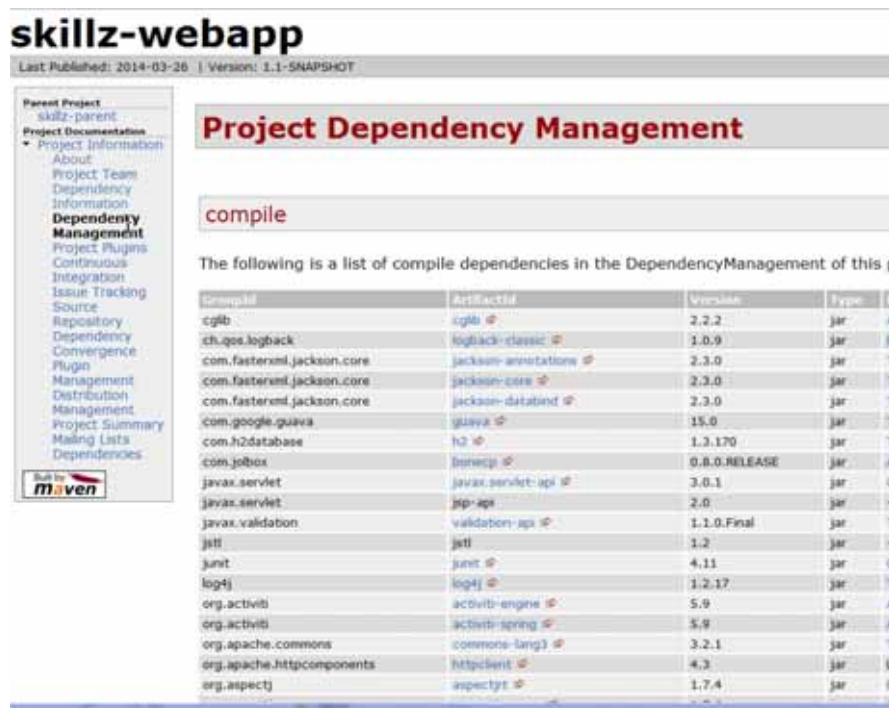
+ Ce lifecycle va générer une arborescence documentaire en html complète dans le répertoire `/target/site`

+ Vous pouvez la consulter de façon purement statique ou en passant par un serveur jetty local

- Ouvrir le fichier `index.html` dans `/target/site` de votre module
- Executer `mvn site:run` et accéder à l'url `http://127.0.0.1`

GENERATION DE SITE / RAPPORTS

Quelques exemples



GroupId	ArtifactId	Version	Type	Scope
cglib	cglib	2.2.2	jar	A
ch.qos.logback	logback-classic	1.0.9	jar	E
com.fasterxml.jackson.core	jackson-annotations	2.3.0	jar	T
com.fasterxml.jackson.core	jackson-core	2.3.0	jar	T
com.fasterxml.jackson.core	jackson-databind	2.3.0	jar	T
com.google.guava	guava	15.0	jar	T
com.h2database	h2	1.3.170	jar	T
com.jolbox	bonecp	0.8.0.RELEASE	jar	A
javax.servlet	javax.servlet-api	3.0.1	jar	C
javax.servlet	jsp-api	2.0	jar	C
javax.validation	validation-api	1.1.0.Final	jar	T
jstl	jstl	1.2	jar	C
junit	junit	4.11	jar	C
log4j	log4j	1.2.17	jar	T
org.activiti	activiti-engine	5.9	jar	A
org.activiti	activiti-spring	5.9	jar	A
org.apache.commons	commons-lang3	3.2.1	jar	T
org.apache.httpcomponents	httpclient	4.3	jar	L
org.aspectj	aspectjrt	1.7.4	jar	E

© SQLI GROUP – 2012

217

GENERATION DE SITE / RAPPORTS

Personnalisation de l'apparence – le fichier site.xml

+ L'apparence par défaut du site est conçue pour être très générique

+ Pour adapter certains aspects du site généré, il est nécessaire de passer par des fichiers de configurations

+ Il faut :

- créer un répertoire `/src/site` destiné à recevoir tous les éléments de customisation du site générés
- Y créer un fichier `site.xml` minimaliste

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/DECORATION/1.4.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/DECORATION/1.4.0
    http://maven.apache.org/xsd/decoration-1.4.0.xsd">
  </project>
```

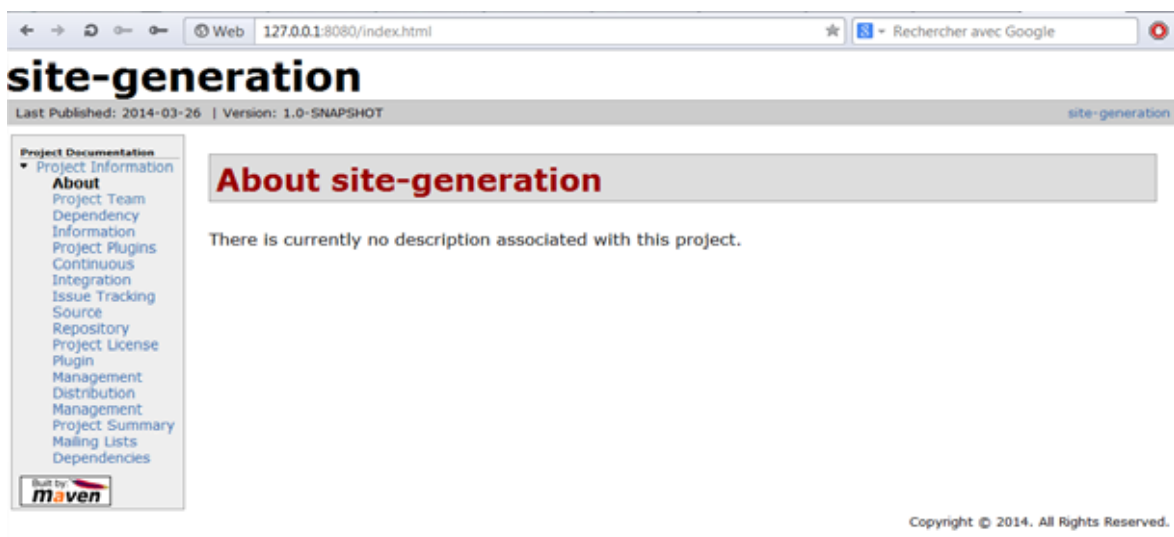
© SQLI GROUP – 2012

218

+ On peut ensuite customiser les éléments suivants :

- ▀ Bannière
- ▀ Positionnement des informations contextuelles (version, date...)
- ▀ Contenu du menu de navigation
- ▀ Contenus statiques

- ▀ Avant customisation via site.xml



GENERATION DE SITE / RAPPORTS

Personnalisation de l'apparence

➤ Après customisation



GENERATION DE SITE / RAPPORTS

Personnalisation de l'apparence – Exemple de fichier

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/DECORATION/1.4.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/DECORATION/1.4.0 http://maven.apache.org/xsd/decoration-1.4.0.xsd"
  name="Projet SQLI Skillz">

  <bannerLeft>
    <name>Formation maven</name>
    <src>images/methodes-sqli-ecommerce-590x303.png</src>
    <href>http://www.sqli.com</href>
  </bannerLeft>

  <bannerRight>
    <src>images/232704_980.png</src>
  </bannerRight>

  <publishDate position="bottom" />
  <version position="bottom"/>

  <body>
    <menu name="Formation maven">
      <item name="Presentation" href="index.html"/>
    </menu>

    <menu ref="reports"/>
    <menu ref="modules" />
  </body>
</project>
```

+ Les éléments du fichier site.xml sont

- <project>
 - + Attribut « name » : pour changer le titre du site
- <bannerLeft> <bannerRight>
 - + Permet de changer les logos de la bannière supérieure du site (droite et gauche)
 - + Attributs :
 - alt, border, width, height, title, name, src : équivalent html
 - Href : adresse de l'image à partir de src/site/resources
- <publishDate> <version>
 - + Attribut « position » : pour positionner date de publication et version
 - left, right, navigation-top, navigation-bottom, bottom

+ On peut inclure du contenu statique à la génération du site

- Rédiger les contenus statiques au format apt, fml, xdoc, html ou xhtml
 - + Ces formats sont hors du scope de la formation
- Déposer ces fichiers dans l'arborescence /src/site selon le layout suivant

```
site/
+- apt/
| +- index.apt
| +- about.apt
| +- embedding.apt
|
+- fml/
| +- faq.fml
|
+- resources/
| +- images/
| | +- banner-left.png
| | +- banner-right.png
| |
| +- architecture.html
| +- jira-roadmap-export-2007-03-26.html
|
+- xdoc/
| +- xml-example.xml
|
+- site.xml
```



+ Le menu de gauche peut-être complètement adapté

```
<body>
  <menu name="Formation maven">
    <item name="Presentation" href="index.html"/>
  </menu>

  <menu ref="reports"/>
  <menu ref="modules" />
</body>
```

▀ <menu> : une entrée de menu

- + <item> : un lien dans le menu pointant sur une ressource statique du site
- + Attribut ref : indique à maven de générer un menu contenant des éléments pré-définis standards :
 - parent (lien vers le site du projet parent), reports (rapports), modules (lien vers les sites des modules enfants)



- + **Maven peut produire plusieurs rapports que vous pouvez ajouter au site Web généré pour afficher l'état actuel du projet.**
- + **Ces rapports sont générés à travers des goals de plugins**
- + **Pour inclure ces rapports, il faut ajouter la liste des plugins de rapports dans la section <reporting> du pom**
 - ▀ Chaque plugin étant configurable pour affiner le rapport généré
 - ▀ Certains plugins peuvent générer plusieurs mini-rapports

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-project-info-reports-plugin</artifactId>
      <reportSets>
        <reportSet>
          <reports>
            <report>dependencies</report>
            <report>scm</report>
          </reports>
        </reportSet>
      </reportSets>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <reportSets>
        <reportSet>
          <reports>
            <report>checkstyle</report>
          </reports>
        </reportSet>
      </reportSets>
    </plugin>
  </plugins>
</reporting>
```

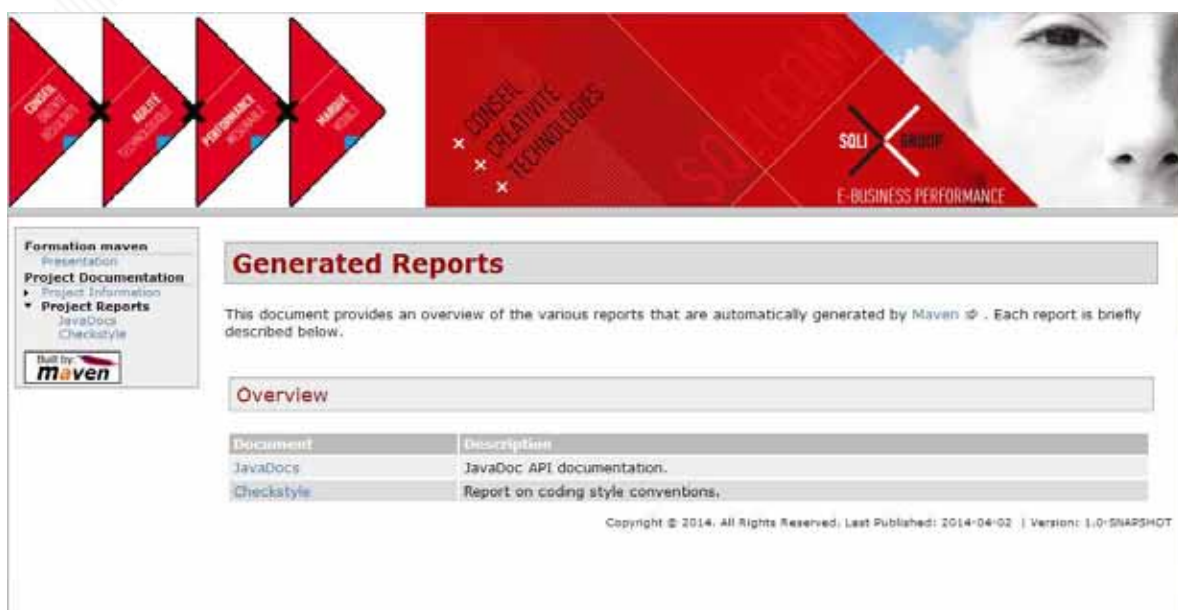
+ De nombreux plugins de rapports sont disponibles

+ Les principaux :

- ▀ checkstyle
 - + Rapport checkstyle (qualité de code)
- ▀ javadoc
 - + Javadoc intégrée au site
- ▀ pmd
 - + Rapport PMD (détection d'erreur potentielles dans le code)
- ▀ surefire-report
 - + Résultats des tests unitaires

+ Se référer à la documentation de référence

- ▀ <http://maven.apache.org/plugins/>, section « reporting »



The screenshot shows the Maven website's 'Generated Reports' page. The header is red with the SQLI GROUP logo and the text 'CONSEIL INNOVATION TECHNOLOGIES' and 'E-BUSINESS PERFORMANCE'. The left sidebar contains a navigation menu with links like 'Formation maven', 'Project Documentation', 'Project Information', 'Project Reports', 'JavaDocs', and 'Checkstyle'. The main content area is titled 'Generated Reports' and contains an overview table of reports.

Document	Description
JavaDocs	JavaDoc API documentation.
Checkstyle	Report on coding style conventions.

Copyright © 2014. All Rights Reserved. Last Published: 2014-04-02 | Version: 1.0-SNAPSHOT

+ Pré-requis :

- ▀ Maven installé

+ Objectif :

- ▀ Créez un projet de type war
- ▀ Ajouter une dépendance

```
<dependency>
```

```
  <groupId>javax</groupId>
```

```
  <artifactId>javaee-web-api</artifactId>
```

```
  <version>6.0</version>
```

```
  <scope>provided</scope>
```

```
</dependency>
```

- ▀ Générez le site du projet, ajoutez des rapports et essayez de customiser son apparence à votre gré.

CREATION DE PLUGINS



- + Introduction
- + Ecrire un plugin
- + Paramètres
- + Plugin et cycle de vie

- + **Nous l'avons vu plusieurs fois, un plugin maven possède comme tout artefact maven un groupId, un artifactId et un numéro de version**
- + **Ecrire un plugin maven commence donc par la création d'un module maven avec un packaging de type 'maven-plugin'**
 - Depuis la version 3 de maven, la création de plugin est simplifiée par l'utilisation d'annotation Java 5 ; c'est cette approche que nous allons utiliser dans la formation
- + **Ce plugin se résumera à une ou plusieurs classes java munie(s) d'une méthode *execute*, point d'entrée de l'exécution du plugin**
 - Une classe java == un goal de notre plugin == un 'mojo'
 - Un plugin maven est souvent constitué de plusieurs mojos.

+ Il faut respecter les conventions de nommage pour le groupId

▀ <yourplugin>-maven-plugin

+ Maven expose à travers l'api de développement de plugins de nombreuses informations contextuelles (pom, dependances...) ; cette api étant extrêmement riche, nous ne ferons qu'aborder les bases du développement de plugin sans aller trop loin

+ Introduction

+ Ecrire un plugin

+ Paramètres

+ Plugin et cycle de vie

+ Une fois le module maven créé, il faut y ajouter les dépendances utile à l'écriture du code java du plugin

▀ Dépendances de compilation

- + org.apache.maven:maven-plugin-api:x.xx (api mojo)
- + org.apache.maven.plugin-tools:maven-plugin-annotations:xxx (support des annotations)

+ Et configurer l'exécution d'un plugin nécessaire au build des mojos

- + Nécessaire pour interpréter les annotations java 5
- + Crée un fichier descripteur de plugin (v2 de maven)

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-plugin-plugin</artifactId>
      <version>3.2</version>
      <configuration>
        <skipErrorNoDescriptorsFound>true</skipErrorNoDescriptorsFound>
      </configuration>
      <executions>
        <execution>
          <id>mojo-descriptor</id>
          <goals>
            <goal>descriptor</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

- + Une fois le module créé (de type maven-plugin), les dépendances et la configuration du build effectuée dans le pom, on peut écrire le code java des mojos**

```
import org.apache.maven.plugin.MojoExecutionException;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugins.annotations.Mojo;

@Mojo(name = "sayhi")
public class MinimalMojo extends AbstractMojo {

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

- + Ce plugin tout simple affichera un message prédéfini dans la console lors de l'exécution**

- ▀ A invoquer avec**

- + mvn myGroupId:myArtifactId:myVersion:sayhi**

- + Introduction**
- + Ecrire un plugin**
- + Paramètres**
- + Plugin et cycle de vie**

CREATION DE PLUGINS

Ecrire un plugin – Passage de paramètres

+ Le passage de paramètres au plugin est très simple

- Il faut créer une variable dans la classe java
- Annoter cette variable avec `@Parameter`
 - Property indique le nom de la propriété qui va servir à passer l'argument au mojo
 - defaultValue : valeur par défaut si la propriété n'est pas précisée

```
@Mojo(name = "say")
public class ParametrizedMojo extends AbstractMojo {

    @Parameter(property = "say.message", defaultValue = "Hello World!")
    private String message;

    public void execute() throws MojoExecutionException {
        getLog().info(message);
    }
}
```

- A invoquer avec

+ `mvn myGroupId:myArtifactId:myVersion:say -Dsay.message=Goodbye`

CREATION DE PLUGINS

Ecrire un plugin – Passage de paramètres

+ Le passage de paramètre peut se faire aussi dans la configuration du plugin

- Balise `<configuration>`
- On ne précise pas le nom du mojo lorsque l'on spécifie le nom du paramètre
 - `say.message => <message>`

```
<plugin>
<groupId>sample.plugin</groupId>
<artifactId>hello-maven-plugin</artifactId>
<version>1.0-SNAPSHOT</version>
<configuration>
    <message> Goodbye</message>
</configuration>
</plugin>
```

+ On peut utiliser différents types en java pour les paramètres

- ▀ String
- ▀ Boolean, Integer, Double,
- ▀ Date
 - + Pattern : 'yyyy-MM-dd HH:mm:ss.S a' ou 'yyyy-MM-dd HH:mm:ssa'
- ▀ File
 - + Ex : c:\tmp
- ▀ URL
 - + Ex : http://www.google.com

+ Certains types plus complexes sont supportés

- ▀ Arrays
 - + Ex :

```
<myArray>
  <param>value1</param>
  <param>value2</param>
</myArray>
```
- ▀ Collections
 - + Même principe que Array
- ▀ Maps


```
<myMap>
  <key1>value1</key1>
  <key2>value2</key2>
</myMap>
```

CREATION DE PLUGINS

Ecrire un plugin – Passage de paramètres

▀ Java.util.Properties

```
< myProperties>
  <property>
    <name>propertyName1</name>
    <value>propertyValue1</value>
  </property>
  <property>
    <name>propertyName2</name>
    <value>propertyValue2</value>
  </property>
</myProperties>
```

CREATION DE PLUGINS

Sommaire

- + Introduction
- + Ecrire un plugin
- + Paramètres
- + **Plugin et cycle de vie**

- + Nous avons illustré l'invocation manuelle de notre plugin
- + En pratique, notre plugin sera plutôt invoqué lors du build d'un projet
- + Il faut donc, comme pour n'importe quel autre plugin maven, le binder à une phase et configurer son exécution
 - ▀ Indiquer l'utilisation du plugin dans la partie <build>
 - ▀ Préciser la phase
 - ▀ Préciser le goal à invoquer
 - ▀ Configurer le passage des paramètres
- + Note : on peut préciser une phase par défaut dans le plugin lui-même
 - ▀ Ex: @Mojo(defaultPhase = LifecyclePhase.VALIDATE, name = "say")

```
<plugin>
  <groupId>training.maven</groupId>
  <artifactId>example-maven-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>
  <executions>
    <execution>
      <id>first-execution</id>
      <phase>generate-resources</phase>
      <goals>
        <goal>say</goal>
      </goals>
      <configuration>
        <message>The Eagle has Landed!</message>
      </configuration>
    </execution>
  </executions>
</plugin>
```


+ Référez vous à la documentation maven pour une description complète des apis et possibilités en matière de développement de plugins maven.

- ▀ Il est possible de faire des choses très complexes dans les plugins maven, l'api exposée n'était cependant pas triviale

+ Pré-requis :

- ▀ Maven installé
- ▀ Disposer d'un éditeur de code java peut-être utile pour ce tp

+ Objectif :

- ▀ Créez un plugin maven qui :
 - + prend en paramètre une chaîne précisant un numéro d'artifactId
 - + Vérifie que la chaîne est conforme au pattern xx.yy.zz
 - + Affiche un message dans la console en cas de non conformité
- ▀ Invoquez le plugin dans l'un des autres projets créés précédemment
 - + Choisissez une phase d'invocation qui vous semble convenir



INTRODUCTION

Maven : Support par les IDE

+ Maven est supporté par les principaux IDE



Et bien sur, l'IDE le plus utilisé (à tort ou à raison)



Eclipse

MAIS ... le support de Maven par Eclipse n'est pas un long fleuve tranquille ...

La conception d'Eclipse rend l'intégration très compliquée, contrairement à NetBeans et IntelliJ pour lesquels l'intégration est aisée et directe

INTRODUCTION

Maven et Eclipse

+ M2E : Maven to Eclipse Integration Plugin

C'est un plugin **Eclipse** pour **Maven**

A ne pas confondre avec le plugin **Maven** pour **Eclipse**



 Sonatype

Historiquement développé par Sonatype, la société derrière Maven



Développement géré par la fondation open source Eclipse depuis 2011

Directement intégré depuis la version
Indigo (06/2011)

Encore en plein développement

2011 (3.7)

2012 (4.2)

2013 (4.3)



INTRODUCTION

M2E : des débuts difficiles, et des limitations

+ La problématique

Eclipse a un processus de build incrémental sophistiqué et une couche d'abstraction élaborée devant le file system

Eclipse et Maven entrent en compétition pour piloter le build du projet : où les autres IDE ont simplement adopté Maven comme builder, il faut dans Eclipse faire cohabiter les deux outils de la façon la plus harmonieuse possible. Pas simple et source de nombreux bugs majeurs, comportements mystérieux et instables (« m2e dance » aka « m2e voodoo »...)

Le plugin M2E a fortement évolué depuis ses première versions.

- Soucis de compatibilité ascendante entre versions de M2E (mineur mais frustrant)
- Les premières versions avaient de nombreuses limitations, progressivement levées

Aujourd'hui encore (03/2014) M2E ne reconnait et ne sais intégrer harmonieusement, et nativement, que les principaux plugins Maven.

L'utilisation de certains plugins Maven, dans un build Maven piloté par Eclipse, reste non supportée à ce jour par le seul plugin M2E, ou même impossible sans développement.

INTRODUCTION

M2E : extension du support des plugins via les connecteurs M2E

+ La solution... et ses contraintes / limitations



Attention : le nombre de combinaisons possibles entre les versions d'Eclipse et du plugin M2E embarqué (et des versions éventuelles de connecteurs complémentaires et/ou de plugins complémentaires) rendent impossible d'adresser tous les cas.

Les informations ci-après sont valables pour une version Kepler SR2 d'Eclipse embarquant un plugin M2E 1.4

En tout état de cause, si vous voulez utiliser l'intégration Maven dans Eclipse, vous devez disposer d'une version récente (on vous aura prévenu).

Il est aussi recommandé d'éviter les build trop exotiques, sauf à disposer de solides compétences en développement sur le framework sous jacent à Eclipse.

INTRODUCTION

M2E : extension du support des plugins via les connecteurs M2E

+ Principe de fonctionnement

M2E va détecter les goals Maven impliqués dans le build Maven par analyse du pom effectif

Si il détecte des plugins qu'il ne sait pas gérer, il va émettre des warnings qui seront matérialisés dans l'éditeur graphique du pom.xml et listés dans la vue des erreurs.

On peut résoudre les warning via un quick-fix dans certains cas. Dans d'autres, il faut procéder à une phase manuelle de configuration du projet et d'Eclipse. Dans d'autres, il n'existe purement et simplement pas de solution out-of-the-box.

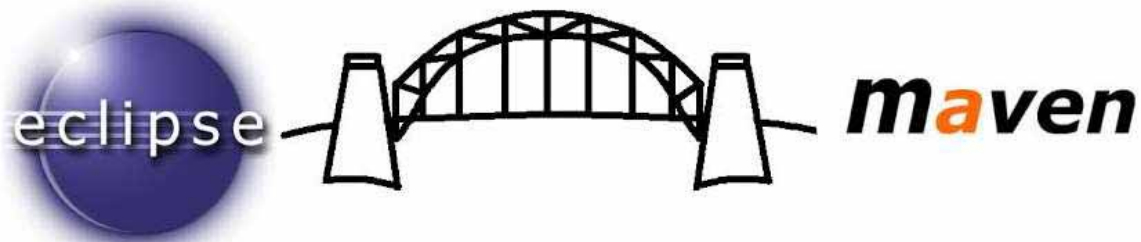
*Cette configuration consiste à déclarer des **connecteurs M2E***

Un connecteur M2e est un plugin Eclipse, spécifique à un plugin Maven, et qui est en charge d'informer Eclipse (via M2E) des actions impactantes pour le bon déroulement du build effectués par le plugin Maven qu'il interface. Ex : génération de code source ou de ressource, modification de byte-code (rappel : Eclipse ne voit pas les modification sur le file system effectuées hors de son scope)

INTRODUCTION

M2E : extension du support des plugins via les connecteurs M2E

+ Connecteur M2E ... c'est quoi encore ce truc ?



Un connecteur M2E est donc un bridge entre un plugin Maven et le plugin M2E, qui est lui-même un bridge entre le builder interne Eclipse et Maven

INTRODUCTION

M2E : extension du support des plugins via les connecteurs M2E

+ M2E-WTP : ensemble de connecteurs M2E pour WTP

Rappel : WTP, Web Tools Project, est un ensemble de plugins apportant des fonctionnalités pour le développement web dans Eclipse. Il est embarqué dans la distribution « Eclipse IDE for Java EE Developers »



The Maven Integration for WTP project, also known as m2e-wtp, provides a tight integration between Maven Integration for Eclipse (m2e) and the Eclipse Web Tools Project (WTP). m2e-wtp provides a set of m2e connectors used for the configuration of Java EE projects in WTP, brings unique Maven features to your Eclipse IDE and helps you convert your Eclipse projects to Maven

<http://projects.eclipse.org/projects/technology.m2e.m2e-wtp>



INTRODUCTION

Eclipse, Maven, M2E, M2E connectors. Qu'en penser ?

+ Conclusion (temporaire)

Le principe de connecteur M2E permet potentiellement de répondre à tous les problèmes

MAIS

- *La documentation du projet M2E est plus que succincte*
- *Peu de connecteurs sont disponibles dans des versions officielles et testées*
- *Le développement ne semble pas très actif (coût d'entrée très important)*
- *Tout nouveau plugin Maven pour être intégré nécessite le développement d'un connecteur spécifique > tout simplement irréaliste*
- *Bref, c'est tout sauf industrialisé à ce jour (03/2014) > REFLECHISSEZ A DEUX FOIS !*



INTRODUCTION

Eclipse et Maven : best practices

+ Conclusion

Eclipse répond parfaitement si vos besoins sont simples, l'utilisation de Maven basique (le multi module est bien géré) et que vous ne mettez pas en œuvre de plugins élaborés

En alternative à WTP + M2E-WTP, envisagez l'utilisation de plugin Tomcat pour Maven (ou autre équivalent)

Si vos besoins sont complexes, utilisez la ligne de commande, ou optez pour un autre IDE que Eclipse

Si l'utilisation d'Eclipse + Maven est une contrainte, ayez conscience de la nécessité de mener une phase de R&D non négligeable pour la mise au point de votre outillage

Exemple de projet Web sans WTP

+ Utilisation des plugins Maven Tomcat et Jetty

L'idée est que c'est Maven qui démarre le serveur tomcat ou jetty

On peut ainsi exécuter l'application mavenisée sans autre outillage que Maven (et les plugins spécifiques)

Très pratique si on ne veut pas se frotter à l'intégration Eclipse WTP/Maven

BONUS

+ Import d'un projet Multi-Module depuis un SCM

Créer un projet multi-module et le commiter sous SVN ou autre ne présente aucune difficulté avec Eclipse

Par contre, faire un checkout du projet multi-module depuis un autre poste est une autre paire de manche...

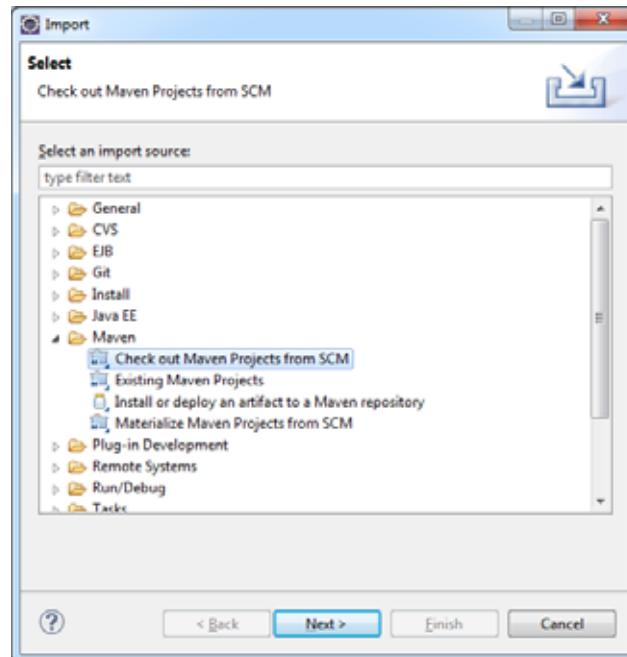
En effet, le checkout classique par la perspective « SVN explorer » ou autre ne permet pas de retrouver un projet multi-module opérationnel

Nombreux sont ceux qui ont le problème, on trouve sur le web divers workaround avec plus ou moins de contraintes.

Il existe pourtant un mécanisme intégré, peu connu, qui comble cette lacune

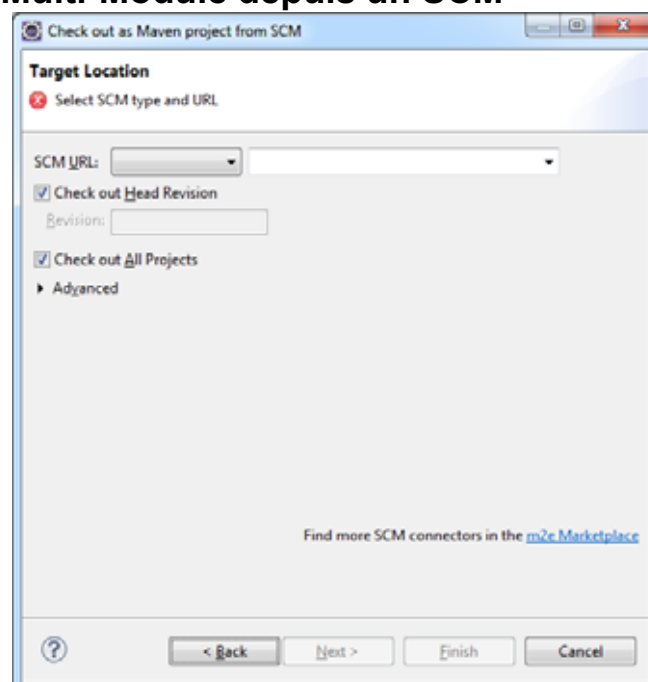
+ Import d'un projet Multi-Module depuis un SCM

File / Import



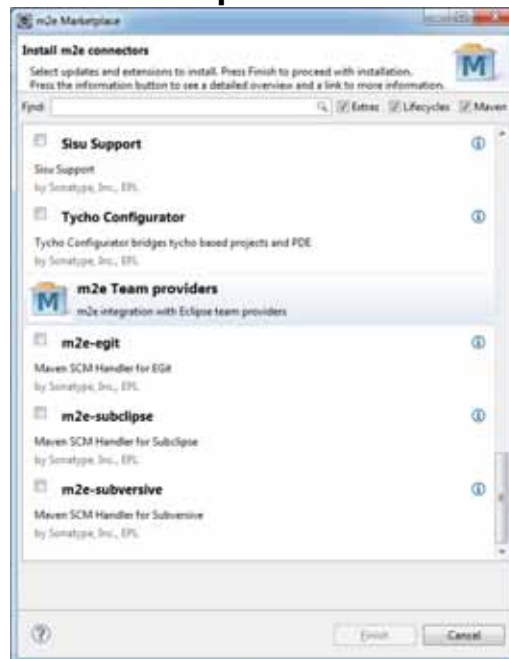
+ Import d'un projet Multi-Module depuis un SCM

*Downloader et installer le
connecteur SCM (lien en
bas à droite)*



+ Import d'un projet Multi-Module depuis un SCM

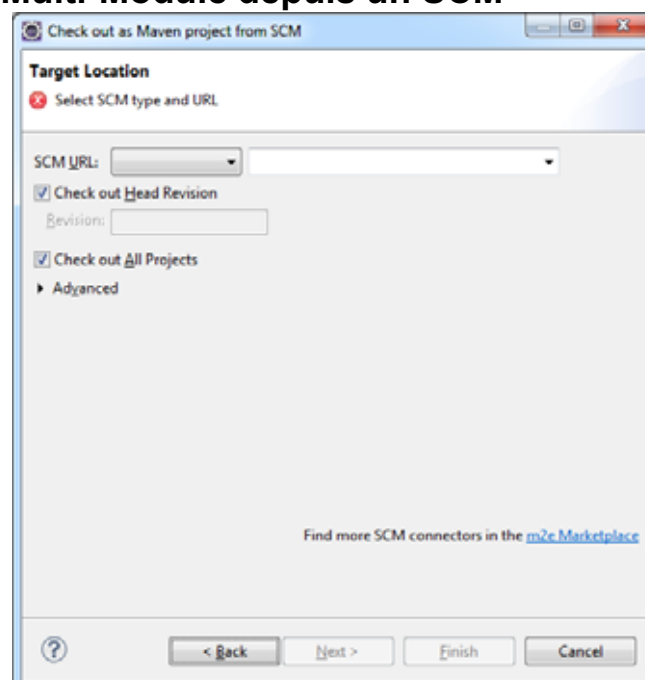
Scroller et sélectionner le
« M2E Team Provider »
correspondant à votre cas



+ Import d'un projet Multi-Module depuis un SCM

Downloader et installer le
connecteur SCM (lien en
bas à droite)

Un redémarrage d'Eclipse
sera nécessaire (proposé
automatiquement)

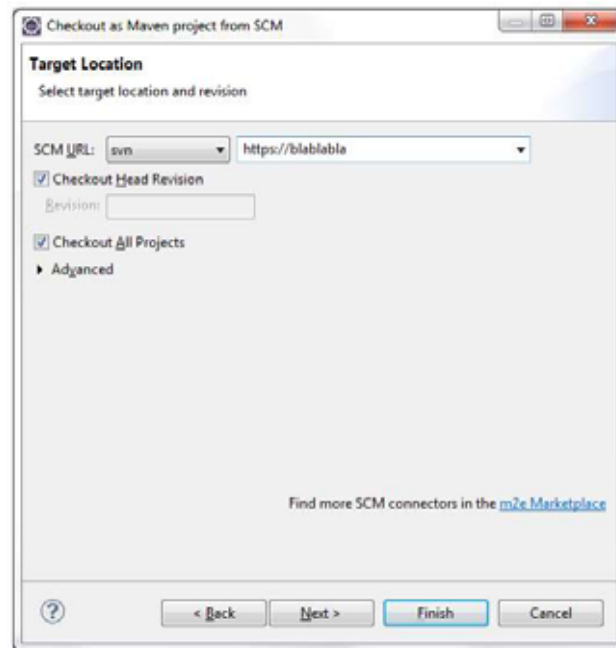


+ Import d'un projet Multi-Module depuis un SCM

Après redémarrage d'Eclipse, on repasse par la même procédure (File / Import / Check Out Maven Projet from SCM).

Cette fois, on peut préciser l'URL SCM (spécifique selon le SCM) et cliquer sur Finish

L'import prendra un peu de temps (attention l'écran peut sembler figé)



+ Automatiser le processus de build

- ▀ Une commande unique permet de compiler une ou plusieurs applications.

+ Automatiser les tests unitaires

- ▀ La phase de test est automatiquement exécutée.
- ▀ Si une erreur est rencontrée alors une alerte est levée

+ Automatiser le déploiement

- ▀ Une fois l'application compilée, testée et packagée elle est déployée automatiquement sur un serveur d'application
- ▀ L'environnement cible est le serveur d'intégration

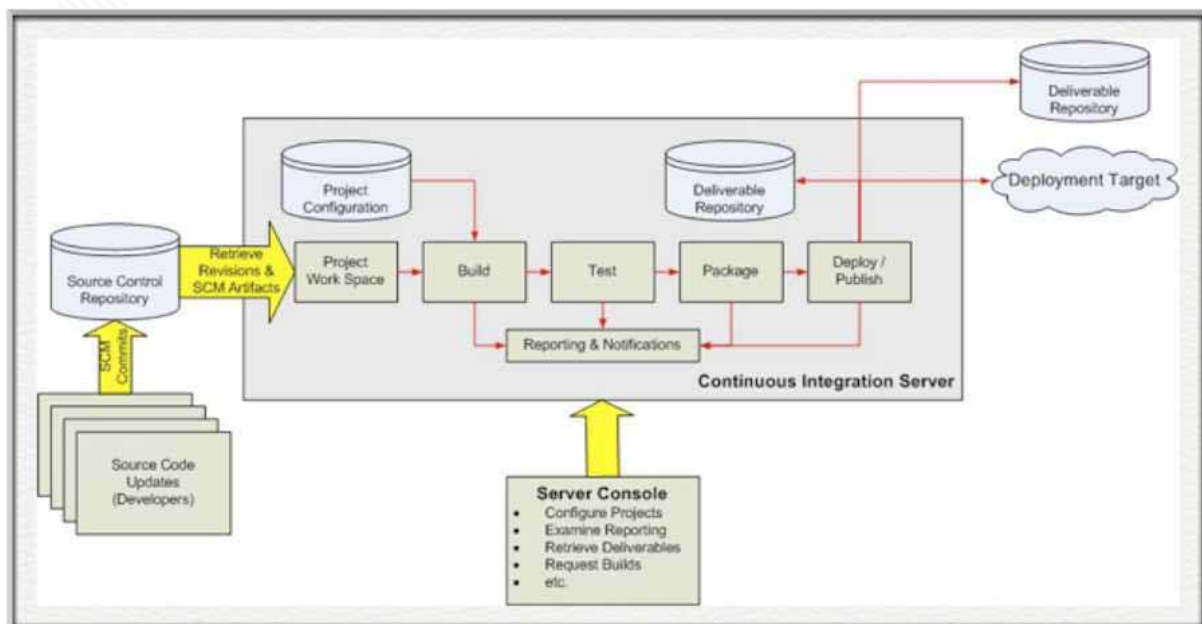
+ Automatiser la génération du portail qualité

- ▀ Le rapport de qualimétrie est centralisé

+ Automatiser la diffusion des informations relative au projet

+ La règle d'or: aucune intervention manuelle lors du processus d'intégration continue.

INTÉGRATION CONTINUE (IC) Workflow



INTÉGRATION CONTINUE (IC) Avantages

- + Les applications sont systématiquement déployées avec les versions les plus récentes du code.**
- + Les erreurs sont détectés très tôt**
 - ▀ Rapports sur les erreurs de compilations.
 - ▀ Alertes sur les résultats des tests unitaires.
- + Les équipes projets ont un retour immédiat et constant sur les conséquences de leur développement.**

+ **Commit réguliers**

- ▀ Les développeurs doivent commiter très régulièrement.
- ▀ Les développeur doivent s'assurer que ce qui est commité est compilable.

+ **Mise en place initiale de la plateforme d'IC longue**

- ▀ Impose une étude d'impacts sur les processus de développement et de livraison.

+ **Hudson**

- ▀ <http://hudson-ci.org/>

+ **Jenkins (fork de Hudson)**

- ▀ <http://jenkins-ci.org/>

+ **CruiseControl**

- ▀ <http://cruisecontrol.sourceforge.net/>

+ **Atlassian Bamboo**

- ▀ <https://www.atlassian.com/software/bamboo>

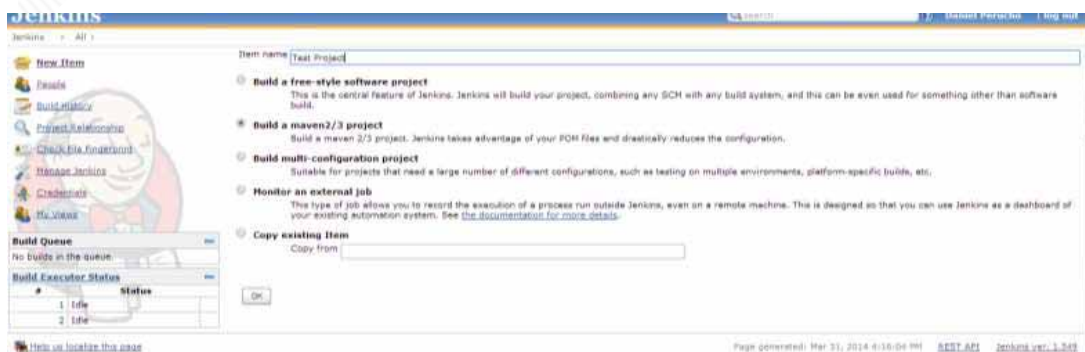
+ **TeamCity**

- ▀ <http://www.jetbrains.com/teamcity/>

- + La définition d'une tâche Maven dans Jenkins est extrêmement simple
- + Jenkins s'appuie sur les conventions Maven pour la gestion des builds

MAVEN ET JENKINS

Configuration d'une tâche (1/3)



- + La configuration initiale consiste à
 - ▀ Définir le nom de la tâche
 - ▀ Le type de build à effectuer

MAVEN ET JENKINS

Configuration d'une tâche (2/3)

configuration

☐ None

☒ Subversion

Modules

Repository URL:

Local module directory (optional):

Repository depth:

Ignore externals: ☒

Check-out Strategy:

Repository browser:

Advanced

Build Triggers

☒ Build whenever a SNAPSHOT dependency is built

☐ Build after other projects are built

☐ Trigger builds remotely (e.g., from scripts)

☐ Build periodically

☐ Poll SCM

Pre Steps

Add prebuild step

Build

Root POM:

Goals and options:

Advanced

Post Steps

Save Apply

MAVEN ET JENKINS

Configuration d'une tâche (3/3)

+ Pour finaliser la configuration

- Définir l'URL pointant vers la ressource au sein du gestionnaire de source
- Définir le nom du fichier POM (par défaut « pom.xml »)
- Spécifier le goal et options pour lancer le build Maven

+ Le système est prêt pour lancer les builds.

+ Plusieurs options pour le lancement de builds

- Les choix sont cumulables

Build Triggers

- ☒ Build whenever a SNAPSHOT dependency is built
- ☐ Build after other projects are built
- ☐ Trigger builds remotely (e.g., from scripts)
- ☐ Build periodically
- ☐ Poll SCM

+ Déclenche automatiquement un build par les dépendances

- ☒ Build whenever a SNAPSHOT dependency is built

+ Soit une application A possédant des dépendances sur les modules M1 et M2

- Si un build est lancé sur le module M1 et/ou M2 alors Jenkins déclenche automatiquement un build de l'application A

+ Conditions à respecter

- Les modules M1 et/ou M2 doivent être gérés par Jenkins
 - + Jenkins introspecte les dépendances de l'application A
- Les versions de M1 et M2 doivent être configurées en tant que SNAPSHOT

+ Déclenche un build après avoir construit un ou plusieurs autres projets

☒ Build after other projects are built

Project names

Batch-Console-UAT-Deploy

Multiple projects can be specified like 'abc, def'

+ Jenkins offre des API de services consommables par d'autres applications

☒ Trigger builds remotely (e.g., from scripts)

Authentication Token

Use the following URL to trigger build remotely: `JENKINS_URL/job/Batch-Console-Sonar/build?token=TOKEN_NAME` or `/buildWithParameters?token=TOKEN_NAME`
Optionally append `&cause=Cause+Text` to provide text that will be included in the recorded build cause.

- Configuration d'un token pour identifier le build

+ Services appelés via les hooks subversion

- Déclenchement automatique d'un build suite à un commit SVN

+ Véritable mise en place de l'intégration continue

- L'application est construite dès qu'une modification est apportée

+ Le build est déclenché à intervalles réguliers

☒ Build periodically

Schedule

H 0 3 * *

- Expression de type crontab à configurer

+ Jenkins effectue un scan du référentiel de sources à intervalles réguliers

☒ Poll SCM

Schedule

H 0 3 * *

- Jenkins lance une compilation dès qu'une ressource à été modifié depuis le dernier build.

+ Mode de déclenchement à éviter car impacte les performances du référentiel de sources

MAVEN ET JENKINS

Console Maven

```
#!/
[INFO] Building tar: /var/e-commerce/jenkins/jobs/Batch-Console-UAT-Deploy/workspace/distribution-batch-console/target/distribution-batch-console-2.3.0-SNAPSHOT-batch-console.tar.gz
[WARNING] Failed to get class for org.codehaus.mojo.tomcat.DeployMojo
[INFO]
[INFO] --- tomcat-maven-plugin:1.1:deploy (default-elt) @ distribution-batch-console ---
[INFO] Skipping non-war project
[INFO]
[INFO] Reactor Summary:
[INFO]
[INFO] Project - Batch Console Application ..... SUCCESS [4.218s]
[INFO] Batch Console Application ..... SUCCESS [0.362s]
[INFO] Module - Batch Console Manager ..... SUCCESS [0.364s]
[INFO] Module - Batch Console Core ..... SUCCESS [10.791s]
[INFO] Module - Batch Console Configuration ..... SUCCESS [0.148s]
[INFO] Module - Batch Console Web Application ..... SUCCESS [0:01.589s]
[INFO] Distribution - Batch-Console Application ..... SUCCESS [9.235s]
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 5:45.259s
[INFO] Finished at: Mon Mar 31 10:45:11 CEST 2014
[INFO] Final Memory: 346/308M
[INFO]
[INFO] Archiving /var/e-commerce/jenkins/jobs/Batch-Console-UAT-Deploy/workspace/distribution-batch-console/pom.xml to com.allians.emagin.vwebapp-batch-console/distribution-batch-console/2.3.0-SNAPSHOT/distribution-batch-console-2.3.0-SNAPSHOT.pom
[INFO] Archiving /var/e-commerce/jenkins/jobs/Batch-Console-UAT-Deploy/workspace/distribution-batch-console/target/distribution-batch-console-2.3.0-SNAPSHOT-batch-console.tar.gz to com.allians.emagin.vwebapp-batch-console/distribution-batch-console/2.3.0-20140331.084504-3/distribution-batch-console-2.3.0-20140331.084504-3-batch-console.tar.gz
[INFO] Archiving /var/e-commerce/jenkins/jobs/Batch-Console-UAT-Deploy/workspace/pom.xml to com.allians.emagin.vwebapp-batch-console/2.3.0-SNAPSHOT/vwebapp-batch-console-2.3.0-SNAPSHOT.pom
[INFO] Archiving /var/e-commerce/jenkins/jobs/Batch-Console-UAT-Deploy/workspace/batch-console/batch-console-config/pom.xml to com.allians.emagin.vwebapp-batch-console/batch-console-config/2.3.0-SNAPSHOT/batch-console-config-2.3.0-SNAPSHOT.pom
[INFO] Archiving /var/e-commerce/jenkins/jobs/Batch-Console-UAT-Deploy/workspace/batch-console/batch-console-config/target/batch-console-config-2.3.0-SNAPSHOT.jar to com.allians.emagin.vwebapp-batch-console/batch-console-config/2.3.0-20140331.083959-5/batch-console-config-2.3.0-20140331.083959-5.jar
[INFO] Archiving /var/e-commerce/jenkins/jobs/Batch-Console-UAT-Deploy/workspace/batch-console/pom.xml to com.allians.emagin.vwebapp-batch-console/batch-console/2.3.0-SNAPSHOT/batch-console-2.3.0-SNAPSHOT.pom
```

+ Jenkins affiche en temps réel les traces d'exécution des commandes Maven.

MAVEN ET JENKINS

Télécharger les livrables

+ Jenkins affiche le lien de téléchargement du livrable

- Parce qu'il s'agit d'un build Maven, Jenkins sait où chercher l'archive.

MAVEN ET JENKINS

Détail des modules

Modules

S	W	Name	Last Success	Last Failure	Last Duration
		Batch Console Application	5 days 23 hr - #1	N/A	51 ms
		Distribution - Batch Console Application	5 days 23 hr - #1	N/A	6.4 sec
		Module - Batch Console Configuration	5 days 23 hr - #1	N/A	0.6 sec
		Module - Batch Console Core	5 days 23 hr - #1	N/A	6.4 sec
		Module - Batch Console Manager	5 days 23 hr - #1	N/A	11 sec
		Module - Batch Console Web Application	5 days 23 hr - #1	N/A	6.2 sec
		Project - Batch Console Application	5 days 23 hr - #1	N/A	3.6 sec

Icons: L

Legend BSS for all BSS for failures BSS for just latest builds

+ Jenkins effectue une introspection du projet

- Les informations sont ventilées par modules
- Il est possible de lancer le build d'un module indépendamment des autres

MAVEN ET JENKINS

Suivi des changements

+ Changements effectués depuis le dernier build

```
#13
[SUMMARY] user: Campaign Management
[trunk/webapp-batch-console/batch-console-webapp/src/main/resources/META-INF/spring/batch/jobs/CAMP_MAIN_PROSPECT_Export_B2C_CSV.xml]
[trunk/webapp-batch-console/batch-console-webapp/src/main/resources/META-INF/spring/batch/jobs/CampManagementTransferJob.xml]
Revision 43481 by ACC_AmitS:
ECONDEV-10687-Campaign Management
[trunk/webapp-batch-console/batch-console-webapp/src/main/resources/META-INF/spring/batch/common/common-jobs-context.xml]
Revision 43480 by ACC_AmitS:
ECONDEV-10687-Campaign Management
[trunk/webapp-batch-console/batch-console-config/src/main/it/resources/dev.properties]
[trunk/webapp-batch-console/batch-console-config/src/main/it/resources/int.properties]
[trunk/webapp-batch-console/batch-console-config/src/main/it/resources/local.properties]
[trunk/webapp-batch-console/batch-console-config/src/main/it/resources/no-test.properties]
[trunk/webapp-batch-console/batch-console-config/src/main/it/resources/prod.properties]
[trunk/webapp-batch-console/batch-console-config/src/main/it/resources/test.properties]
[trunk/webapp-batch-console/batch-console-config/src/main/it/resources/uat.properties]
[trunk/webapp-batch-console/batch-console-config/src/main/resources/batch-console.properties]
[trunk/webapp-batch-console/batch-console-core/src/main/java/com/ecommerce/service/reporting/activity/B2CPlanMaker.java]
[trunk/webapp-batch-console/batch-console-core/src/main/java/com/ecommerce/export/ProspectJobRunner.java]
[trunk/webapp-batch-console/batch-console-core/src/main/java/com/ecommerce/export/ClientAccount/Util/CutProspectFromItem.java]
[trunk/webapp-batch-console/batch-console-core/src/main/java/com/ecommerce/export/ClientAccount/Writer/CutProspectItemWriter.java]
[trunk/webapp-batch-console/batch-console-core/src/main/java/com/ecommerce/export/transfer/CampManagementTransferReaderImpl.java]
Revision 43475 by HQ_DanielP:
Update POM version for R20140326_HF
[trunk/webapp-batch-console/batch-console-config/pom.xml]
[trunk/webapp-batch-console/batch-console-core/pom.xml]
[trunk/webapp-batch-console/batch-console-manager/pom.xml]
[trunk/webapp-batch-console/batch-console-webapp/pom.xml]
[trunk/webapp-batch-console/batch-console/pom.xml]
[trunk/webapp-batch-console/distribution-batch-console/pom.xml]
[trunk/webapp-batch-console/pom.xml]
```

Pour toute l'application

Changes

#13 (Mar 26, 2014 3:32:35 PM)

43475. Update POM version for R20140326_HF — HQ_DanielP / detail
42932. POM projects version updates for R20140310_MR — HQ_DanielP / detail

Par module

