En finir avec les problèmes de gestion de dépendance

Speakers : Cédric Champeau (Gradle)
Format : Conférence
Date : 19 avril 2018
Slides : https://melix.github.io/devoxxfr-gradle-5-dependency-mgmt/

Gradle est un système de build agnostique. Il peut builder différents langages : les langages de la JVM, mais également du natif (C, C++ et Swift).
Gradle est l'outil de build pour Android.
Quelques chiffres :

- 35 ingénieurs travaillent sur Gradle
- 5 millions de download / mois
- Linkedin réalise 300 000 builds Gradle par semaine

Gradle se focalise sur l'objectif du build. Chaque binaire a ses spécificités : libairie Java vs applicatif Java vs appli native en C++.

Cédric rappelle les différentes techniques de partage du code :

- Sources
  - Nécessite de tout recompiler (lent)
- Binaires
  - Partage des librairies
  - Stable et signé

Problématique : où mettre ses binaires ?

- Anciennement : dossier lib/
- Maven Central + private repositories
- Ivy repository
- Custom repository : repository virtuel avec JitPack

Lib

- Simple
- On a tout
- Ça marche bien

Maven/Ivy repository

- Nécessite de donner des coordonnées aux composants logiciels : groupId/artefactId/version (GAV Maven)
- Nécessite de la gestion des dépendances transitives dans les méta-données (pom.xml ou ivy.xml). Les méta-données donnent des informations qui ont été utilisées lors du build. Très importantes, ce sont également les principales sources d'ennuis.

Attention à ne pas confondre Maven (outil de build) et Maven Central (repo public).

Depuis Gradle, on peut utiliser du Maven Central. Et depuis Maven, on peut utiliser Jitpack.

Que faire si la librairie n'est pas disponible dans Maven Central ?

- Cout d'entrée pour publier sur Maven Central (je vous recommande ce billet)
- Dans Gradle, on va pouvoir pointer sur un repo Git pour construire le binaire. Très pratique dans le monde C / C++

Gestion des dépendances

- Remarque importance sur le scope compile de Maven. Pas de différence entre ce qui sépare l'API de l'implémentation.
- Gradle sait faire la part des choses : permet d'exposer les dépendances utilisées dans les API publiques (ex : commons-lang3). Possibilité de déclarer les dépendances utilisées en interne par l'implémentation.

Pour builder une librairie : API + Implémentation
Pour compiler du code applicatif utilisant une librairie : API de la librairie
Pour l'exécution du code : API + Implémentation + Runtime only dependencies

Problème : toutes les librairies publiés sur Maven sont faussent.

Tout ce qui doit être publié est à destination du consommateur. Ce dernier n'a pas besoin de savoir comment la librairie a été compilée.

Depuis la version 5 de Gradle, il y'a un **nouveau format de métadonnées**.
Objectif : modéliser la complexité du consommateur et du producteur
Méta-données au **format JSON** car dédiées aux outils de builds.
Exemple : sample-module.json

Le mot clé **variant** est utilisé. Déjà présent dans Gradle : permet d'adresser par exemple différentes versions d'une même app mobile pour des devices différents.
Chaque variant dispose de ses propres dépendances.

Les utilisateurs de Guava connaissent les classifiers de guava : guava-jdk5, guava-jdk7 …. Or, il est possible d'avoir 2 versions de Guava dans le classpath car le système de gestion de dépendances n'a pas l'information.

Dans le monde natif, il n'y a pas de dépendance car trop d'hétérogénéité : variantes en fonction du CPU, du mode debug …

Rich Version Constraint

.Dans Gradle, on peut désormais expliquer le choix d'une dépendance
Exemple : version 23 car requis pour les collections immutables

Parfois, les développeurs savent que l'application ne fonctionnera pas avec une autre version que celle spécifiée. Utilisation du mot clé **strictly** pour ne pas ramener de version différente.

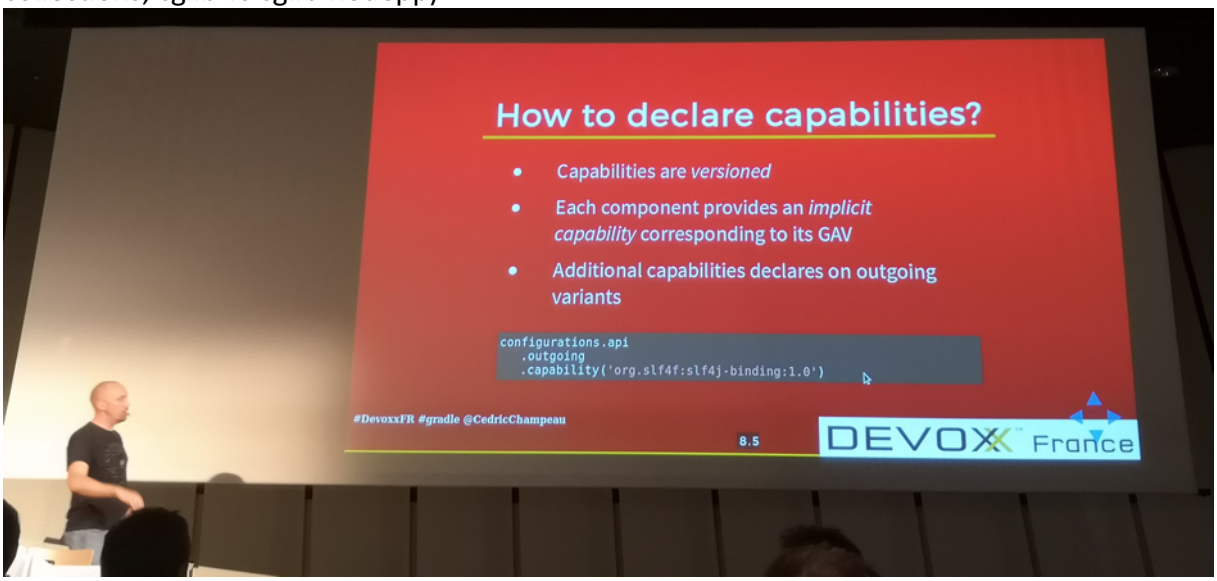
Dependency Constraints

Nouveau concept : Depdendency Constraints
Permet d'ajouter des informations d'une dépendance qu'on n'utilise pas.
Existe en Maven avec le <dependencyManagement> : mais ces informations peuvent être surchargées par les utilisateurs

Concept de platforme
Spring Boot est une forme de plateforme. Les développeurs de Spring Boot ont validé les versions des frameworks qui sont compatibles les unes avec les autres.
Exprimé sous forme de BOM Spring Boot.
Dans Gradle, il sera possible de modéliser ces contraintes. Portabilité imparfaite vers les BOM.

Capabilities

Gradle permet d'exprimer le fait que 2 dépendances sont similaires (même capacité) et ne devraient pas exister en même temps dans le classpah (ex : logback vs log4j, guava vs google collections, cglib vs cglib-nodepp)

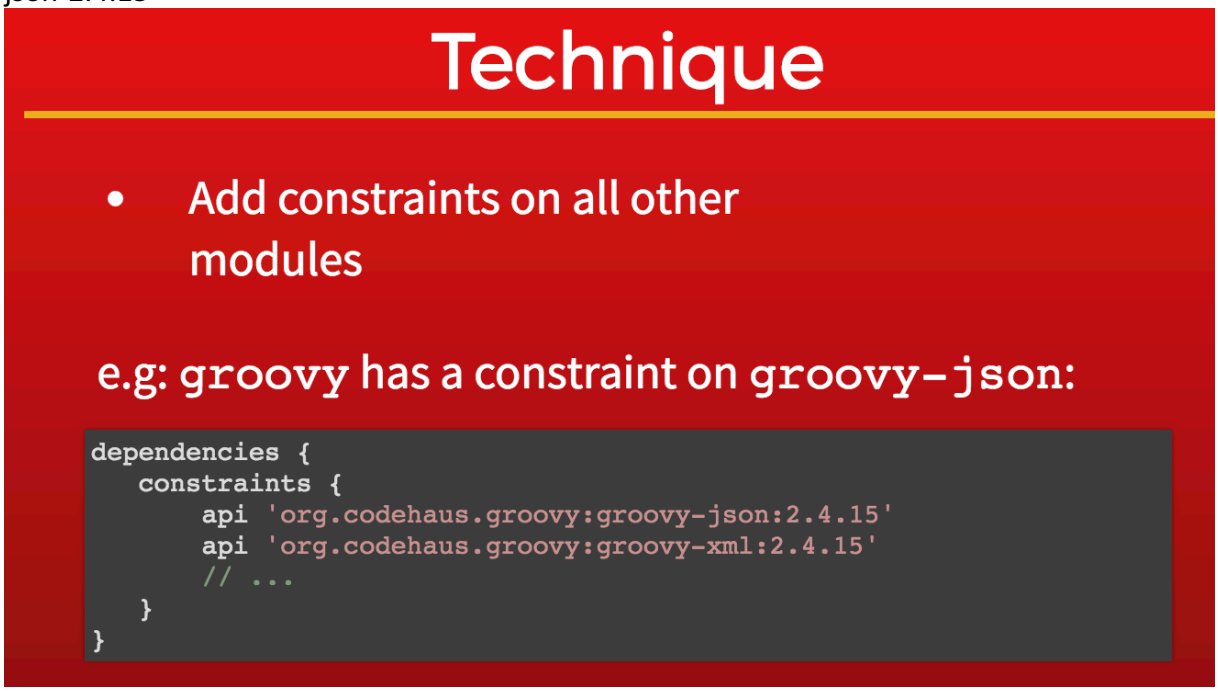


Dynamic dependencies

L'utilisation des ranges pose problème. Les développeurs npm en savent quelque chose.
Pénible pour la reproductibilité du build.
Solution : génération d'un fichier de lock.

Alignement

Certaines versions des librairies doivent être alignées. Exemple : groovy-2.4.15 et groovy-json-2.4.15



Metadata is live

Le cycle de vie d'un module ne se termine pas au moment où elle est publiée. Lorsqu'une vulnérabilité est découverte après la publication, il faut en prendre compte.

Comment réparer les métadonnées ?
Gradle part du principe que les métadonnées sont fausses.
Avant de les utiliser, Gradle va les transformer et les corriger avant de les consommer. Cas typique du exclude qui est transitif et qui n'explique pas pourquoi il est là.

Le plugin dependencyManagement de Spring exploite cette fonctionnalité en corrigeant les métadonnées.