Gradle 5

En finir avec les problèmes de gestion de dépendances

Cédric Champeau (@CedricChampeau), Gradle

DEVOXX France

Who am I

```
speaker {
    name 'Cédric Champeau'
    company 'Gradle Inc'
    oss 'Apache Groovy committer',
    successes 'Static type checker',
              'Static compilation',
              'Traits',
              'Markup template engine',
              'DSLs'
        failures Stream.of(bugs),
        twitter '@CedricChampeau',
        github 'melix',
        extraDescription '''Groovy in Action 2 co-author
Misc OSS contribs (Gradle plugins, deck2pdf, jlangdetect, ...)'''
}
```

Gradle in a nutshell

Agnostic Build System

- Java ecosystem
  - Groovy, Kotlin, Scala, …
- Native ecosystem
  - C, C++, Swift, …
- Android
- Misc
  - Go, Asciidoctor, …

Gradle in figures

Gradle in figures

- 5.0M downloads / month

Gradle in figures

- 5.0M downloads / month
- #17 OSS projects worldwide

Gradle in figures

- 5.0M downloads / month
- #17 OSS projects worldwide
- 35 Gradle Engineers

Gradle in figures

- 5.0M downloads / month
- #17 OSS projects worldwide
- 35 Gradle Engineers
- 300K builds/week @LinkedIn

A Java library

```
plugins {
   id 'java-library'
}

dependencies {
   api 'com.acme:foo:1.0'
   implementation 'com.zoo:monkey:1.1'
}
```

A native app

```
plugins {
   id 'cpp-application'
}

dependencies {
    implementation 'org.gradle.cpp-samples:math:1.5'
}
```

Why dependency management?

Source vs published

- Sources
  - (mostly) reliable
  - (often) slow
  - never touched
  - hard to version
  - safe

Source vs published (2)

- Binaries
  - Stable
  - Fast (pre-built)
  - Requires trusted sources
  - Not always metadata

Consuming binaries

- A `lib` directory
- From a Maven repository
  - Maven Central (OSS libraries)
  - Private repositories (closed source, proxies)
- From an Ivy repository
  - Artifactory, …
- From a custom repository
  - JitPack, …

Lib directory

- Straightforward
- No dependency management at all
- Binaries in SCM

Maven/Ivy repository

- GAV coordinates
- transitive dependencies management
- metadata format restricts what you can do

Custom repositories

- Not portable
- Hard to consume transitively

Maven != Maven Central

- Maven: a build tool
- Maven **repository**: a place where you can find binaries

What if there’s no repository?

- Coming soon: **source dependencies**

```
sourceControl {
   vcsMappings {
      withModule("org.test:greeter") {
         from(GitVersionControlSpec) {
            url = "git@github.com:orgtest/greeter.git"
         }
      }
   }
}
```

Managing dependencies

Typical Maven dependency

```xml
<dependencies>
   <groupId>org.apache.commons</groupId>
   <artifactId>commons-lang3</artifactId>
   <version>3.7</version>
   <scope>compile</scope>
</dependencies>
```

Using Gradle

```
dependencies {
   // An API dependency is used in public APIS
   api 'org.apache.commons:commons-lang3:3.7'

   // or...
   // An implementation dependency is used in internals
   implementation 'org.apache.commons:commons-lang3:3.7'
}
```

API vs implementation

- To *build* a library, you need:
  - API+implementation dependencies
- To *compile* against a library, you need:
  - API dependencies
- To *run*, you need:
  - API+implementation+runtime only dependencies

Corollary

All libraries published using Maven do it wrong

Published metadata

- Should be aimed at *consumers*
- It doesn’t matter what you need to compile
- It matters what the *consumers* need
- Published POM should be != producer POM

What Gradle does

- Since 3.4, use the `java-library` plugin
- Maps to `compile` and `runtime` scopes in `pom.xml`
- But it’s not enough…

Gradle module metadata

- Aimed at modeling properly *variants* of modules
- Death to classifiers (mostly)
- Model different set of dependencies
- Multi-ecosystem (Java, Native, …)

Gradle metadata format

See sample

Consequence

- `all`/`fat` jars published with correct dependencies
- `guava-jdk5`, `guava-jdk7`, … no longer need to be classifiers
- attributes for matching variants

For native

```
bash-3.2$ ./gradlew check
```

Variant-aware

```
> Task :subvola:gorgoneum:teerer:polytonal:dependencyInsight
project :outissue:carnally
  variant "debugRuntimeElements" [
    c.android.b.a.attributes.BuildTypeAttr   = debug
    c.android.b.g.dep.VariantAttr            = debug (not requested)
    org.gradle.usage                         = java-runtime
    c.android.b.gradle.dep.AndroidTypeAttr   = Aar
  ]
```

Variant-awareness

- Can be used to model complex requirements:
  - "Give me a version which passed QA"
  - "Give me a version optimized for arm64"
  - "Give me stubs for this library"

Rich version constraints

Meaning of versions

- What does it mean to say: "I depend on 1.1"
- Does it mean it doesn’t work using 1.0?
- Implicit statement: "I should work with 1.1+"
- What if it’s not true?

Meaning of versions

- Use `latest.release`?
- Dependency on `1.2-beta-3`: is `beta` important?
- Dependency on snapshots…

Custom dependency reasons

- Explain *why* a dependency is here

```
dependencies {
   implementation('com.google.guava:guava') {
      version { prefer '23' }
      because 'required for immutable collections'
   }
}
```

Custom dependency reasons

- Shown in dependency insight report

```
> gradle dependencyInsight
         --configuration compileClasspath
         --dependency guava

org:foo:com.google.guava:guava:23 (required for immutable collection
   variant "default" [
      Requested attributes not found in the selected variant:
         org.gradle.usage = java-api
   ]
```

Strict versions

- Dependency should be **exactly** this version, or *fail*

```
dependencies {
   api('com.acme:foo') {
      version {
         strictly '1.1'
      }
      because "Only version approved by QA"
   }
}
```

Rejected versions

- Dependency should be **exactly** this version, or *fail*

```
dependencies {
   api('com.acme:foo') {
      version {
         prefer '[1.0, 2.0)'
         reject '1.1'
      }
      because "Version 1.1 has a vulnerability"
   }
}
```

Dependency constraints

Concept

- Influence versions found in the graph, without adding hard dependencies
- "If you use this module, use this version"

<dependencyManagement>

Similar to Maven's `<dependencyManagement>` block but:

- enforced transitively
- published
- consistent behavior

Example 1: dependency version suggestion

```
dependencies {
    constraints {
       api 'com.acme:foo:1.0'
    }

    // no need to put a version number
    api 'com.acme:foo'
}
```

Example 2: influence transitive dependency version

```
dependencies {
    constraints {
       // if 'bar' found transitively, use 1.1
       api 'com.acme:bar:1.1'
    }
    // ...
}
```

Platform vs library

- Platforms define things that "work together"
- Suggests versions, not hard dependencies
- Consumers *depend on* a platform for suggestions

Example: Spring Boot BOM

Constraints as platforms

```
apply plugin: 'java-platform'

dependencies {
   constraints {
       platformApi 'org.springframework.boot:spring-boot:1.5.8-RELEA
       platformApi 'org.springframework.boot:spring-boot-test-autoco
       // ...
   }
}
```

Constraints publication

- Published as constraints in Gradle metadata

```
{
   "variants": [
      {
         "name": "api",
         "dependencyConstraints": [
            { "group": "org.springframework.boot", "module": "spring
            { "group": "org.springframework.boot", "module": "spring
         ],
         "attributes": { "usage": "compile" }
},
...
```

- Published as `<dependencyManagement`

DEVOXX France

Capabilities

Not all conflicts are version conflicts

- `awesome-lib` depends on `commons-logging`
- `react-lib` depends on `jcl-over-slf4j`

Problem: you shouldn’t have both on classpath

Not all conflicts are version conflicts

- `google-collections` was superceded by `guava`
- `groovy-all` provides the same capability as `groovy`

Future-proof

- If anybody introduces a conflict, we *will* discover it:

```
Cannot choose between
    cglib:cglib-nodep:3.2.5 and cglib:cglib:3.2.5
    because they provide the same capability: cglib:cglib:3.2.5
```

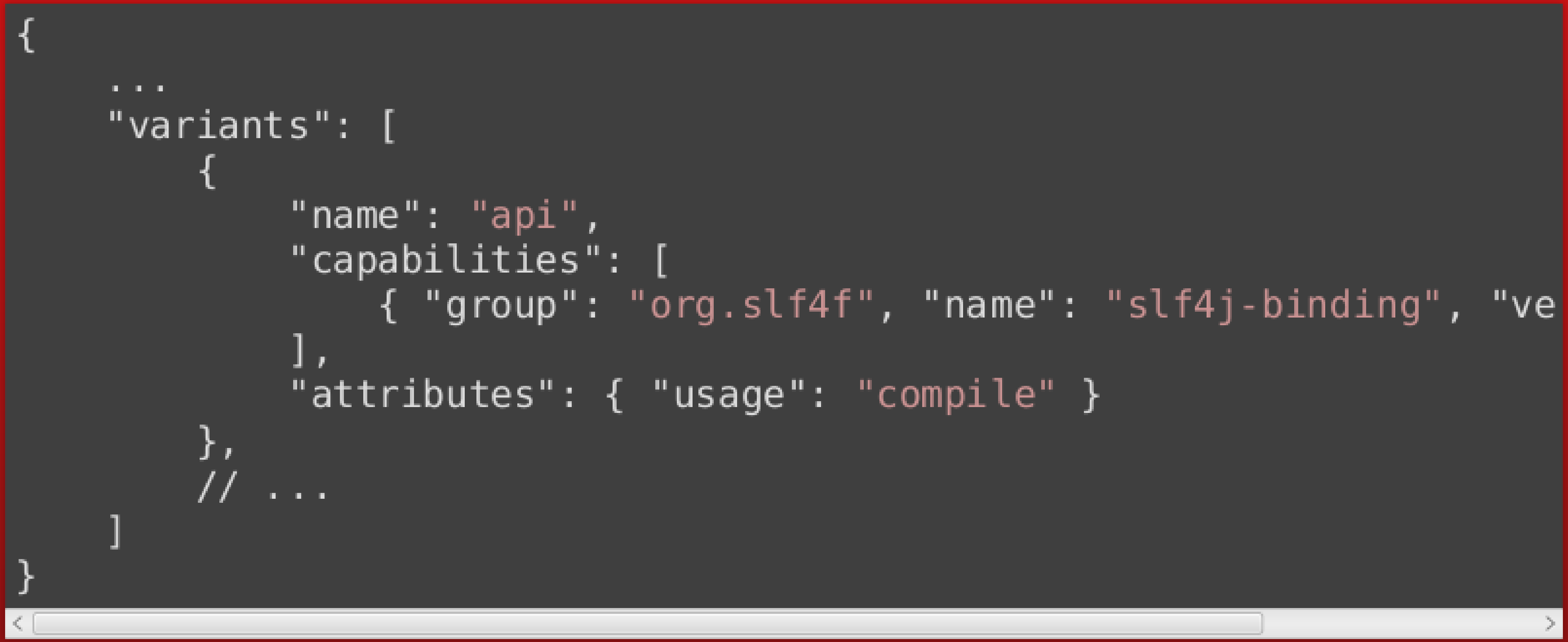
How to declare capabilities?

- Capabilities are *versioned*
- Each component provides an *implicit capability* corresponding to its GAV
- Additional capabilities declares on outgoing variants

```
configurations.api
   .outgoing
   .capability('org.slf4f:slf4j-binding:1.0')
```

Capabilities are published

- Gradle metadata only!



```
{
    ...
    "variants": [
        {
            "name": "api",
            "capabilities": [
                { "group": "org.slf4f", "name": "slf4j-binding", "ve
            ],
            "attributes": { "usage": "compile" }
        },
        // ...
    ]
}
```

#DevoxxFR #gradle @CedricChampeau

DEVOXX™ France

Dependency locking

Idea: make dynamic dependencies acceptable

- Ranges bad for reproducibility: `[1.0, )`
- May break build without notice
- Doesn’t enforce a tested version

Dependency locking

- Remember *resolved* version numbers
- *Lock* them in a lock file
- Use the lock file when resolving
- Lock file is pushed to VCS
- Fail if a dependency was upgraded

Usage

- Activate locking

```
dependencyLocking {
   lockAllConfigurations()
}
```

- Generate locks

```
./gradlew dependencies --write-locks
```

Example lock file

compileClasspath.lockfile

```
# This is a Gradle generated file for dependency locking.
# Manual edits can break the build and are not advised.
# This file is expected to be part of source control.
android.arch.core:common:1.0.0
android.arch.lifecycle:common:1.0.3
android.arch.lifecycle:runtime:1.0.3
com.android.support:animated-vector-drawable:27.0.2
com.android.support:appcompat-v7:27.0.2
com.android.support:support-annotations:27.0.2
com.android.support:support-compat:27.0.2
com.android.support:support-core-ui:27.0.2
com.android.support:support-core-utils:27.0.2
...
```

Alignment

Module sets

- Some modules are meant to be used together
  - e.g: `groovy-2.4.15` with `groovy-json-2.4.15`
- if one is upgraded, the other has to be upgraded too

Technique

- Add constraints on all other modules

e.g: `groovy` has a constraint on `groovy-json`:

```
dependencies {
   constraints {
       api 'org.codehaus.groovy:groovy-json:2.4.15'
       api 'org.codehaus.groovy:groovy-xml:2.4.15'
       // ...
   }
}
```

Metadata is live

Lifecycle doesn’t end at publishing

- Modules are published at date `d`
- Bugs are discovered at `d+1`
- Reaches maturity at `d+70`
- Vulnerabilities are discovered at `d+147`
- Should we allow using vulnerable dependencies?

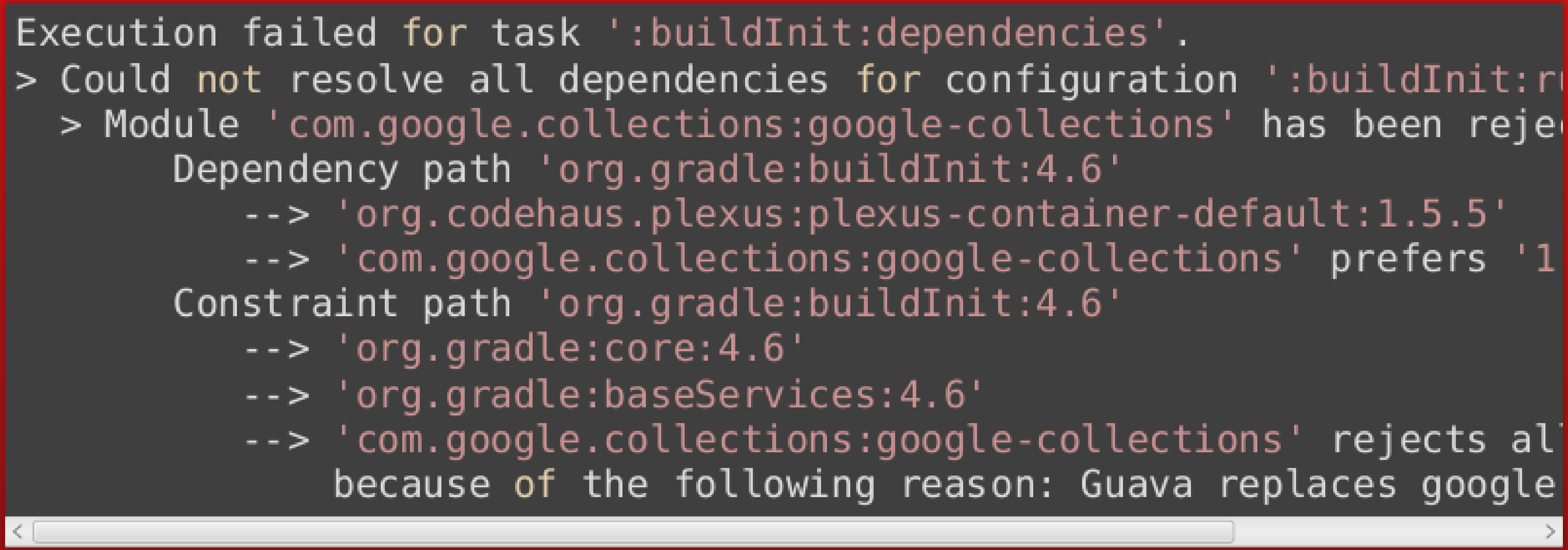
Blacklisting

Fail if we resolve to a blacklisted version

```
dependencies {
   constraints {
       implementation('org.foo:awesome-lib') {
           version {
               prefer '1.2'
               reject '1.1'
           }
           because 'Version 1.1 is buggy'
       }
   }
}
```

Error messages

- Error message will give more context



```
Execution failed for task ':buildInit:dependencies'.
> Could not resolve all dependencies for configuration ':buildInit:r
  > Module 'com.google.collections:google-collections' has been reje
       Dependency path 'org.gradle:buildInit:4.6'
          --> 'org.codehaus.plexus:plexus-container-default:1.5.5'
          --> 'com.google.collections:google-collections' prefers '1
       Constraint path 'org.gradle:buildInit:4.6'
          --> 'org.gradle:core:4.6'
          --> 'org.gradle:baseServices:4.6'
          --> 'com.google.collections:google-collections' rejects al
              because of the following reason: Guava replaces google
```

Deprecated modules

- Use case: "Library X is deprecated, please use Y instead"
- Similar to blacklisting
- Warn instead of fail

Component metadata rules

Fixing bad metadata

- Libraries are often published with *bad* metadata
  - strong dependencies instead of optional
  - wrong scope
  - incorrect version
  - excludes that shouldn't be there
  - …

Component metadata rules

- Modifies metadata of a component (consumer only)
- Allows adding/removing dependencies/constraints/capabilities

Component metadata rules: example 1

- Downgrading a dependency

```
withModule(module) {
   allVariants {
      withDependencyConstraints {
         filter { it.group == "org.apache.ivy" }.forEach {
            version { prefer("2.2.0") }
            because("Gradle depends on ivy implementation details whi
         }
      }
   }
}
```

Component metadata rules: example 2

- Remove a dependency

```
withModule("org.eclipse.jgit:org.eclipse.jgit") {
   allVariants {
      withDependencies {
         removeAll { it.group == "com.googlecode.javaewah" }
      }
   }
}
```

Component metadata rules: example 3

- Add a capability

```
withModule('org.ow2.asm:asm') { module ->
   allVariants {
      withCapabilities {
         addCapability("asm", "asm", module.id.version)
      }
   }
}
```

Conclusion

Be part of the new world!

Conclusion

- Slides: https://melix.github.io/devoxxfr-gradle-5-dependency-mgmt
- Discuss: @CedricChampeau

Thanks!