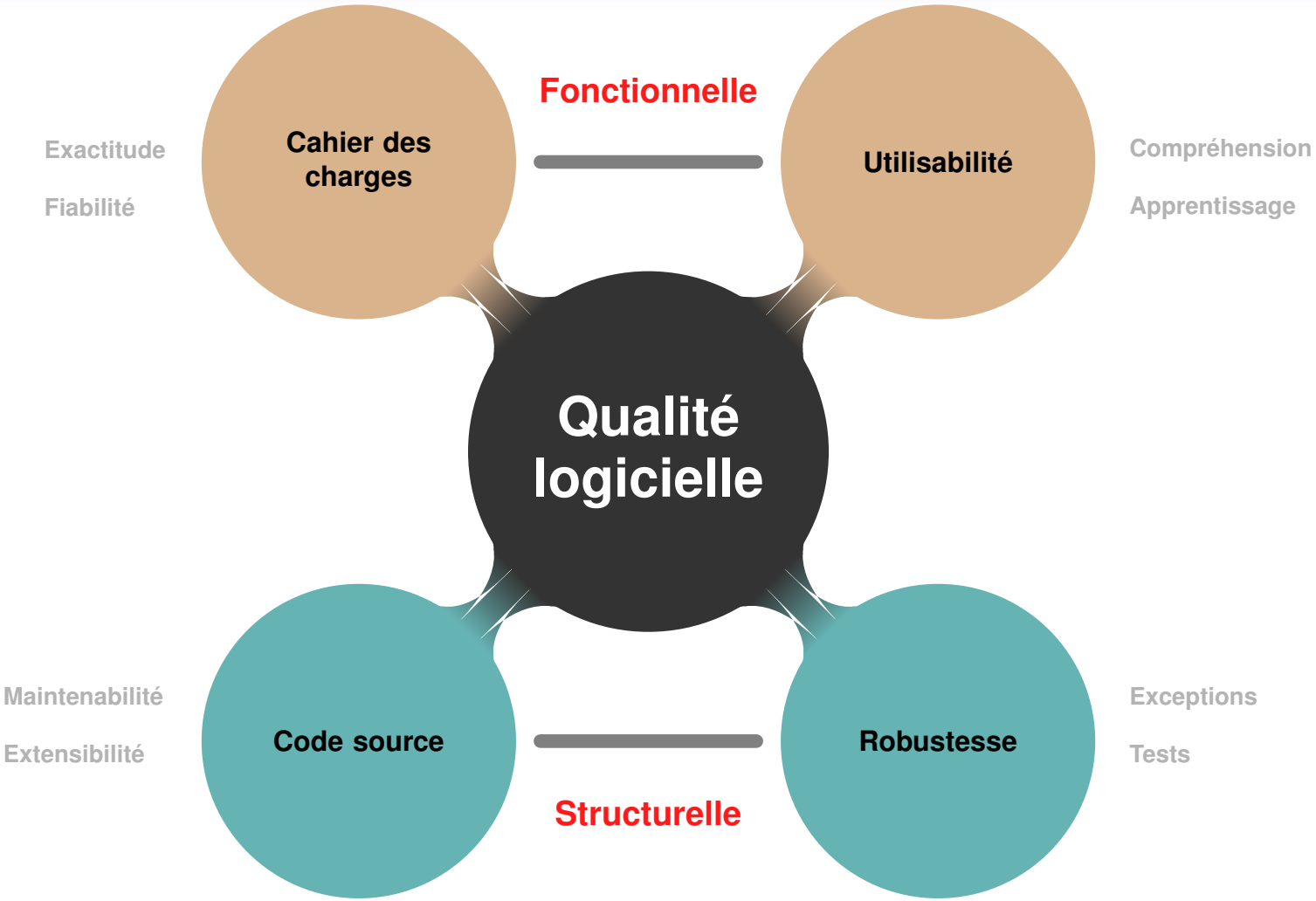
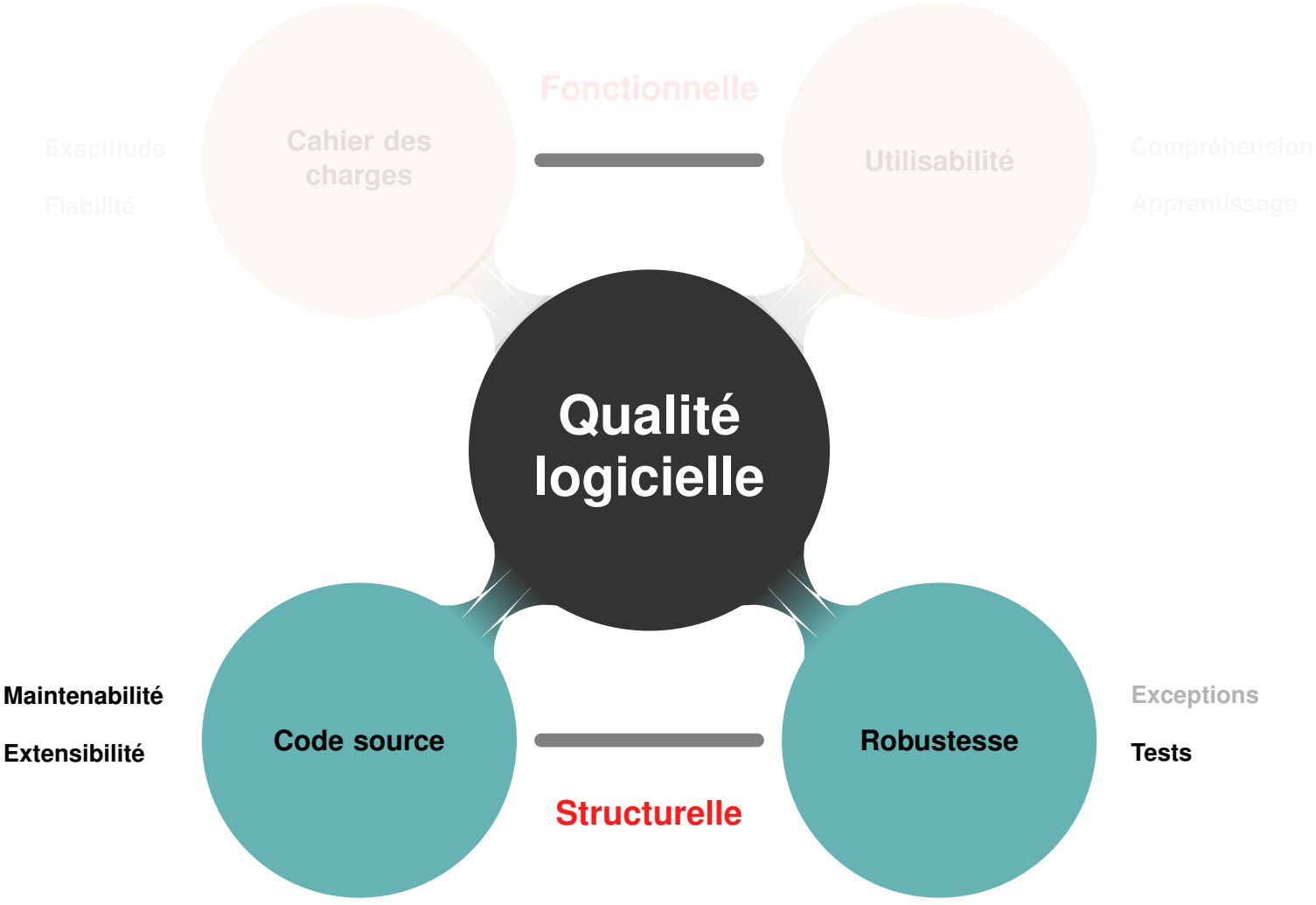
Plan

Qualité fonctionnelle vs. structurelle



Organisation et gestion du code source ⇒ Maintenabilité



Plan

Gradle ?

Un *moteur de production* logiciel

- logiciel libre, 2007, Licence Apache 2.0 ▸ gradle.org
- fonctionnalités similaires à Maven / Ant / Apache Ivy
- compilation / test / intégration continue / déploiement...
- multi-projets : e.g. une application + ses librairies + exemples...
- utilisé pour de très gros projets : Android, NETFLIX...

Écosystème

Supported Languages and Frameworks

Gradle supports Android, Java, Kotlin Multiplatform, Groovy, Scala, Javascript, and C/C++.

Compatible IDEs

All major IDEs support Gradle, including Android Studio, IntelliJ IDEA, Visual Studio Code, Eclipse, and NetBeans.

You can also invoke Gradle via its command-line interface (CLI) in your terminal or through your continuous integration (CI) server.

Gradle

Principales caractéristiques

- ⇒ principe *zéro-conf* : *convention plutôt que configuration*
- ⇒ gestion des dépendances et de leurs versions
- ⇒ accélère la compilation/exécution (cache/démon)
- ⇒ fait de manière à être étendu facilement : plugin

Installation

- Nécessite une JVM 8+
- présent dans tous les dépôts principaux Linux
- ▸ gradle.org/install
- ⇒ *Gradle user home*: `~/.gradle` (dep, tools, cache. . .)

Plan

Création d'un projet

Possible de créer un projet géré par Gradle via l'IDE, mais la ligne de commande reste, pour l'instant, le meilleur choix. ⇒ `gradle init`:

```
fab@boid ~/ztmp/demo
$ gradle init

Select type of project to generate:
  1: basic
  2: application
  3: library
  4: Gradle plugin
Enter selection (default: basic) [1..4] 2

Select implementation language:
  1: C++
  2: Groovy
  3: Java
  4: Kotlin
  5: Scala
  6: Swift
Enter selection (default: Java) [1..6]

Split functionality across multiple subprojects?:
  1: no - only one application project
  2: yes - application and library projects
Enter selection (default: no - only one application project) [1..2]

Select build script DSL:
  1: Groovy
  2: Kotlin
Enter selection (default: Groovy) [1..2]

Generate build using new APIs and behavior (some features may change in the next minor release)? (default: no) [yes, no] yes

Select test framework:
  1: JUnit 4
  2: TestNG
  3: Spock
  4: JUnit Jupiter
Enter selection (default: JUnit Jupiter) [1..4]

Project name (default: demo):

Source package (default: demo):

> Task :init
Get more help with your project: https://docs.gradle.org/7.5.1/samples/sample_building_java_applications.html
```

Groovy Kotlin

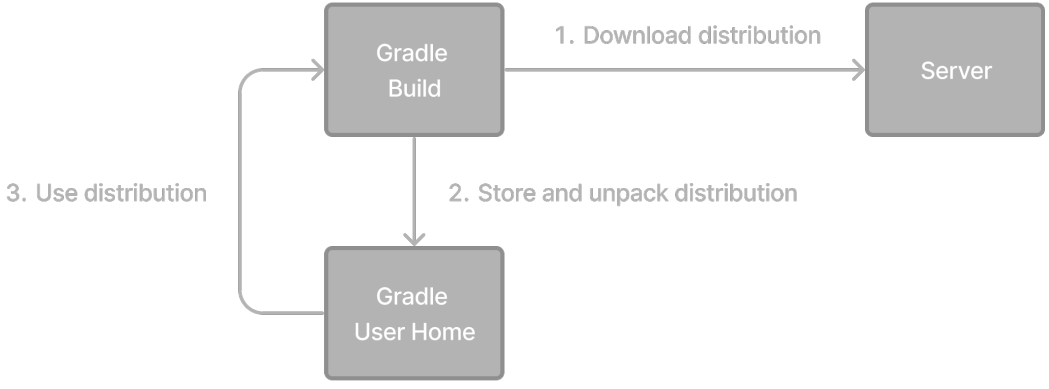
```
├── gradle (1)
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew (2)
├── gradlew.bat (2)
├── settings.gradle (3)
└── app
    ├── build.gradle (4)
    └── src
        ├── main
        │   └── java (5)
        │       └── demo
        │           └── App.java
        └── test
            └── java (6)
                └── demo
                    └── AppTest.java
```

1. Generated folder for wrapper files
2. Gradle wrapper start scripts
3. Settings file to define build name and subprojects
4. Build script of `app` project
5. Default Java source folder
6. Default Java test source folder

Gradle Wrapper : une version par projet

Uutilise la version de Gradle spécifiée par le projet pour les build

▸ wrapper



- Le programme `gradlew` se trouve à la racine
- La bonne version sera téléchargée sur l'hôte si besoin
- ⇒ à la racine : `./gradlew <task>`
- ⇒ dans un sous-projet : `../gradlew <task>`

settings.gradle

Groovy Kotlin

settings.gradle

```
rootProject.name = 'demo'
include('app')
```

- **`rootProject.name`** : nom global du build/projet (pas nécessairement le nom du dossier contenant)
- **`include('app')`** : nom des sous-projets inclus, un seul ici : `app`

app/build.gradle

Groovy | Kotlin

app/build.gradle

```
plugins {
    id 'application' (1)
}

repositories {
    mavenCentral() (2)
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.8.2' (3)

    implementation 'com.google.guava:guava:31.0.1-jre' (4)
}

application {
    mainClass = 'demo.App' (5)
}

tasks.named('test') {
    useJUnitPlatform() (6)
}
```

(1) Apply the application plugin to add support for building a CLI application in Java.

(2) Use Maven Central for resolving dependencies.

(3) Use JUnit Jupiter for testing.

(4) This dependency is used by the application.

(5) Define the main class for the application.

app/src/main/java/demo/App.java

```java
/*
 * This Java source file was generated by the Gradle 'init' task.
 */
package demo;

public class App {
    public String getGreeting() {
        return "Hello World!";
    }

    public static void main(String[] args) {
        System.out.println(new App().getGreeting());
    }
}
```

app/src/test/java/demo/AppTest.java

```java
/*
 * This Java source file was generated by the Gradle 'init' task.
 */
package demo;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class AppTest {
    @Test void appHasAGreeting() {
        App classUnderTest = new App();
        assertNotNull(classUnderTest.getGreeting(), "app should have a greeting");
    }
}
```

`./gradlew run` et `./gradlew build`

```
fab@boid ~/ztmp/demo
$ ./gradlew run

> Task :app:run
Hello World!

BUILD SUCCESSFUL in 588ms
2 actionable tasks: 2 executed
fab@boid ~/ztmp/demo
$ ./gradlew build

BUILD SUCCESSFUL in 1s
7 actionable tasks: 6 executed, 1 up-to-date
```

`./gradlew build`

- ⇒ app/build/distributions/app.tar
- ⇒ app/build/distributions/app.zip

Plan

`build.gradle` ⇒ bloc `dependencies`

Groovy | Kotlin

app/build.gradle

```
plugins {
    id 'application' (1)
}

repositories {
    mavenCentral() (2)
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.8.2' (3)

    implementation 'com.google.guava:guava:31.0.1-jre' (4)
}

application {
    mainClass = 'demo.App' (5)
}

tasks.named('test') {
    useJUnitPlatform() (6)
}
```

(1) Apply the application plugin to add support for building a CLI application in Java.

(2) Use Maven Central for resolving dependencies.

(3) Use JUnit Jupiter for testing.

(4) This dependency is used by the application.

(5) Define the main class for the application.

Gestion des dépendances

Principes et intérêts

- Généralement un projet nécessite d'autres librairies : fonctionnalités avancées / test / intégration continue . . .
- Gradle $\Rightarrow$ automatisation des tâches de déclaration / résolution / utilisation

Fonctionnement

- déclaration : spécifie la version de la librairie
- résolution : récupère la librairie depuis un dépôt ou localement
- utilisation : spécifie le niveau d'usage : compilation / test / API / runtime. . .

Exemple de déclaration des dépendances

build.gradle

```
repositories {
    // Use Maven Central for resolving dependencies.
    mavenCentral()
}

dependencies {
    // Use JUnit test framework.
    testImplementation("junit:junit:4.13.2")

    // This dependency is used by the application.
    implementation("com.google.guava:guava:32.1.2-jre")
}
```

Some key concepts in Gradle dependency management include:

Repositories - The source of dependencies → `mavenCentral()`

Maven Central is a collection of jar files, plugins, and libraries provided by the Maven community and backed by Sonatype. It is the de-facto public artifact store for Java and is used by many build systems.

une dépendance : `group` / `name` / `version`

```
dependencies {
    // Use JUnit test framework.
    testImplementation("junit:junit:4.13.2")

    // This dependency is used by the application.
    implementation("com.google.guava:guava:32.1.2-jre")
}
```

Spécification d'une dépendance

	Description	com.google.guava:guava:32.1.2-jre	junit:junit:4.13.2
Group	identifier of an organization	`com.google.guava`	`junit`
Name	dependency identifier	`guava`	`junit`
Version	version # to import	`32.1.2-jre`	`4.13.2`

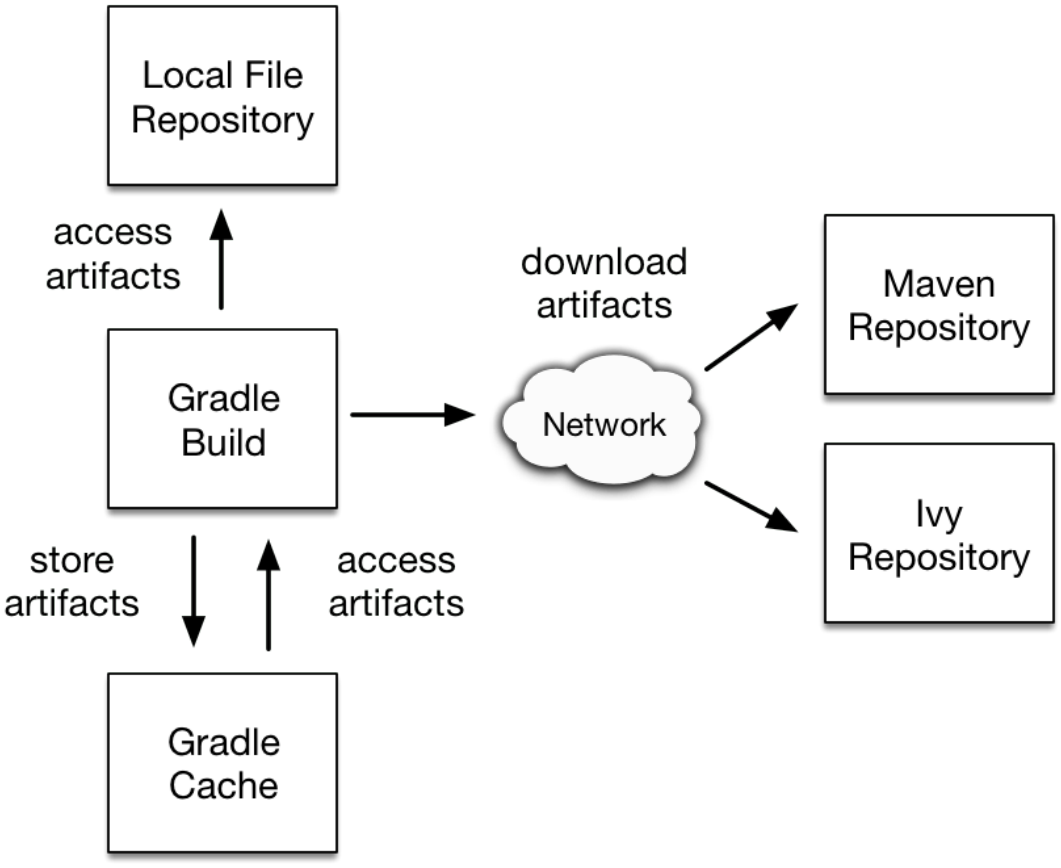
Visualisation des dépendances

Dépendances du projet et leurs dépendances

```
$ ./gradlew :app:dependencies

> Task :app:dependencies

------------------------------------------------------------
Project ':app'
------------------------------------------------------------

...

compileClasspath - Compile classpath for source set 'main'.
\--- com.google.guava:guava:32.1.2-jre
     +--- com.google.guava:failureaccess:1.0.1
     +--- com.google.guava:listenablefuture:9999.0-empty-to-avoid-conflict-w
     +--- com.google.code.findbugs:jsr305:3.0.2
     +--- org.checkerframework:checker-qual:3.12.0
     +--- com.google.errorprone:error_prone_annotations:2.11.0
     \--- com.google.j2objc:j2objc-annotations:1.3

...
```

The output clearly depicts that `com.google.guava:guava:32.1.2-jre` has a dependency on `com.google.guava:failureaccess:1.0.1`.

Résolution


Local File Repository
access artifacts
Gradle Build
store artifacts
access artifacts
Gradle Cache
download artifacts
Network
Maven Repository
Ivy Repository

Paramétrage des dépôts

Example 1. Adding central Maven repository

Groovy | Kotlin

build.gradle

```
repositories {
    mavenCentral()
}
```

Paramétrage des dépôts

Ajouter d'autres serveurs de dépôt, e.g. propres à l'entreprise

Example 3. Declaring multiple repositories

Groovy | Kotlin

build.gradle

```
repositories {
    mavenCentral()
    maven {
        url "https://repo.spring.io/release"
    }
    maven {
        url "https://repository.jboss.org/maven2"
    }
}
```

Paramétrage des dépôts

Dépendances dans des dossiers locaux :

Example 4. Flat repository resolver

Groovy | Kotlin

build.gradle

```
repositories {
    flatDir {
        dirs 'lib'
    }
    flatDir {
        dirs 'lib1', 'lib2'
    }
}
```

Plan

Exécution des tests

Exécuter les tests nécessite

- qu'ils soient codés dans `app/src/test/java` (défaut)
- d'exécuter la tâche `test` ou `check` (qui fera plus tard plus de chose)
- c'est tout !

Résultats sous forme de rapport dans `app/build/reports/tests/test/index.html`

Test Summary

1	0	0	0.014s
tests	failures	ignored	duration

100% successful

Packages Classes

Package	Tests	Failures	Ignored	Duration	Success rate
demo.app	1	0	0	0.014s	100%

Configuration (un peu plus) avancée

Exemple de configuration

build.gradle

```
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.7.1'
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}

tasks.named('test', Test) {
    useJUnitPlatform()

    maxHeapSize = '1G'

    testLogging {
        events "passed"
    }
}
```

Configuration (un peu plus) avancée

Options possibles pour la tâche `test`

You can control how the test process is launched via several properties on the `Test` task, including the following:

`maxParallelForks` **— default: 1**

You can run your tests in parallel by setting this property to a value greater than 1. This may make your test suites complete faster, particularly if you run them on a multi-core CPU. When using parallel test execution, make sure your tests are properly isolated from one another. Tests that interact with the filesystem are particularly prone to conflict, causing intermittent test failures.

Your tests can distinguish between parallel test processes by using the value of the `org.gradle.test.worker` property, which is unique for each process. You can use this for anything you want, but it's particularly useful for filenames and other resource identifiers to prevent the kind of conflict we just mentioned.

`forkEvery` **— default: 0 (no maximum)**

This property specifies the maximum number of test classes that Gradle should run on a test process before its disposed of and a fresh one created. This is mainly used as a way to manage leaky tests or frameworks that have static state that can't be cleared or reset between tests.

Warning: a low value (other than 0) can severely hurt the performance of the tests

`ignoreFailures` **— default: false**

If this property is `true`, Gradle will continue with the project's build once the tests have completed, even if some of them have failed. Note that, by default, the `Test` task always executes every test that it detects, irrespective of this setting.

`failFast` **— (since Gradle 4.6) default: false**

Set this to `true` if you want the build to fail and finish as soon as one of your tests fails. This can save a lot of time when you have a long-running test suite and is particularly useful when running the build on continuous integration servers. When a build fails before all tests have run, the test reports only include the results of the tests that have completed, successfully or not.

You can also enable this behavior by using the `--fail-fast` command line option, or disable it respectively with `--no-fail-fast`.

`testLogging` **— default: *not set***

This property represents a set of options that control which test events are logged and at what level. You can also configure other logging behavior via this property. See `TestLoggingContainer` for more detail.

`dryRun` **— default: false**

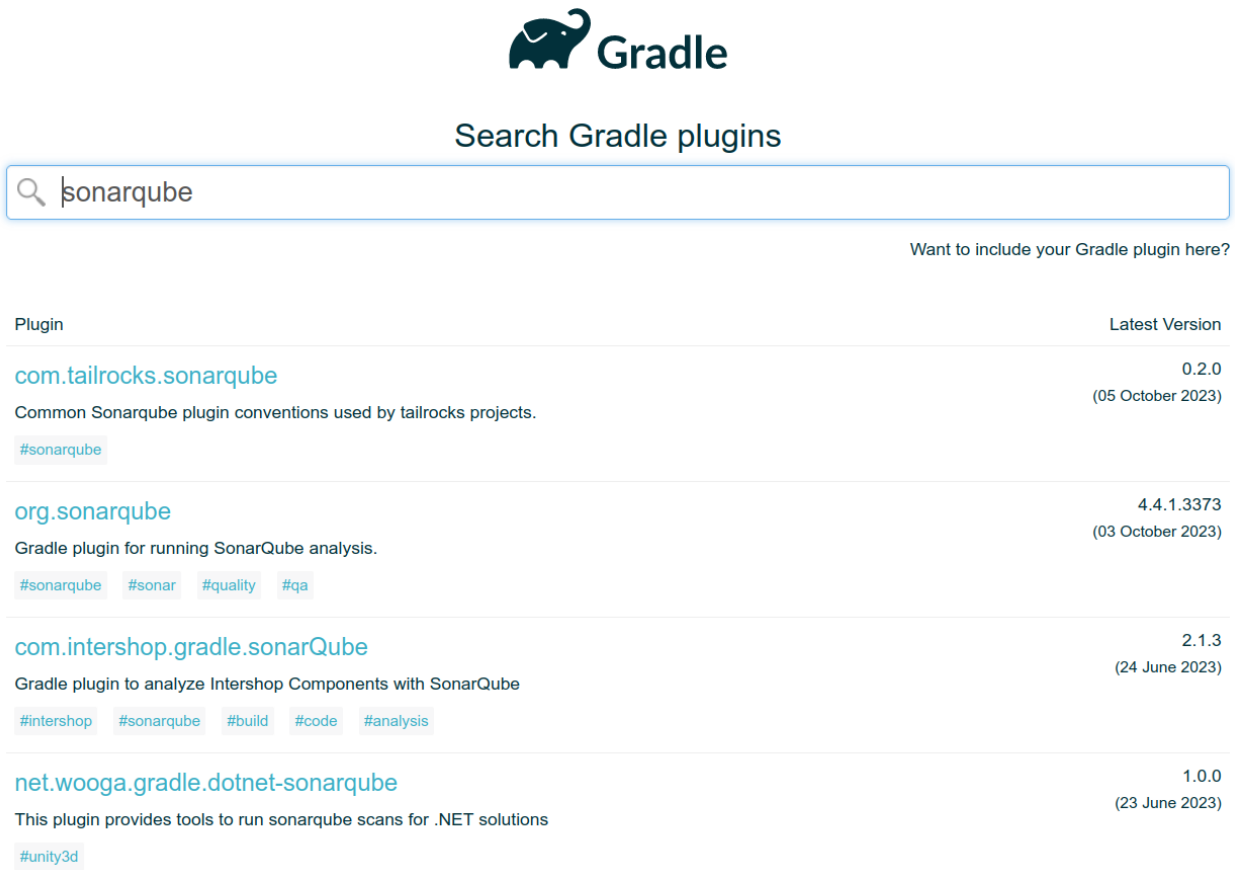
If this property is `true`, Gradle will simulate the execution of the tests without actually running them. This will still generate reports, allowing for inspection of what tests were selected. This can be used to verify that your test filtering configuration is correct without actually running the tests.

Plan

Gradle plugins

Pour les tâches plus spécifiques (e.g. CI/CD), Gradle fonctionne sur un principe de plugins.

- gradle plugins hub



Plugin pour dépendances complexes
Exemple pour JavaFX

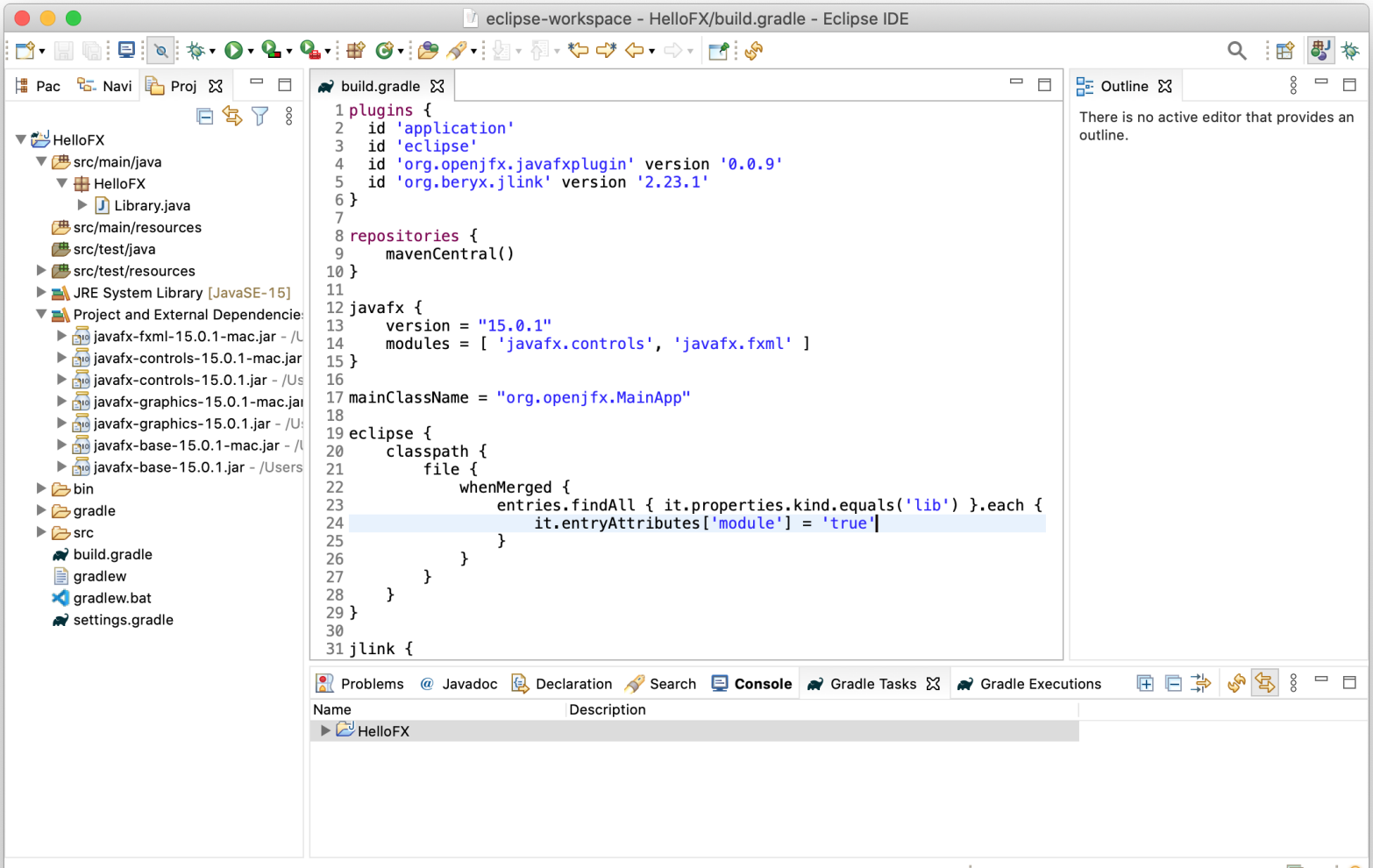
Parfois, les dépendances sur des projets complexes (e.g. modularisés) peuvent être configurées grâce à des plugins dédiés.
Exemple avec ▸ org.openjfx.javafxplugin qui rajoute un bloc `javafx`

build.gradle (JavaFX_samples) ×

```
1 plugins {
2     id 'application'
3     id 'org.openjfx.javafxplugin' version '0.1.0'
4 }
5
6 repositories {
7     mavenCentral()
8 }
9
10 javafx {
11     version = '17'
12     modules = [ 'javafx.controls', 'javafx.fxml' ]
13 }
14
15 application {
16     mainClass = 'helloworld.HelloWorldFX'
17 }
```

Les IDE nécessitent souvent des ajustements...

Voir ici pour JavaFX : ▸ JavaFX and IDEs



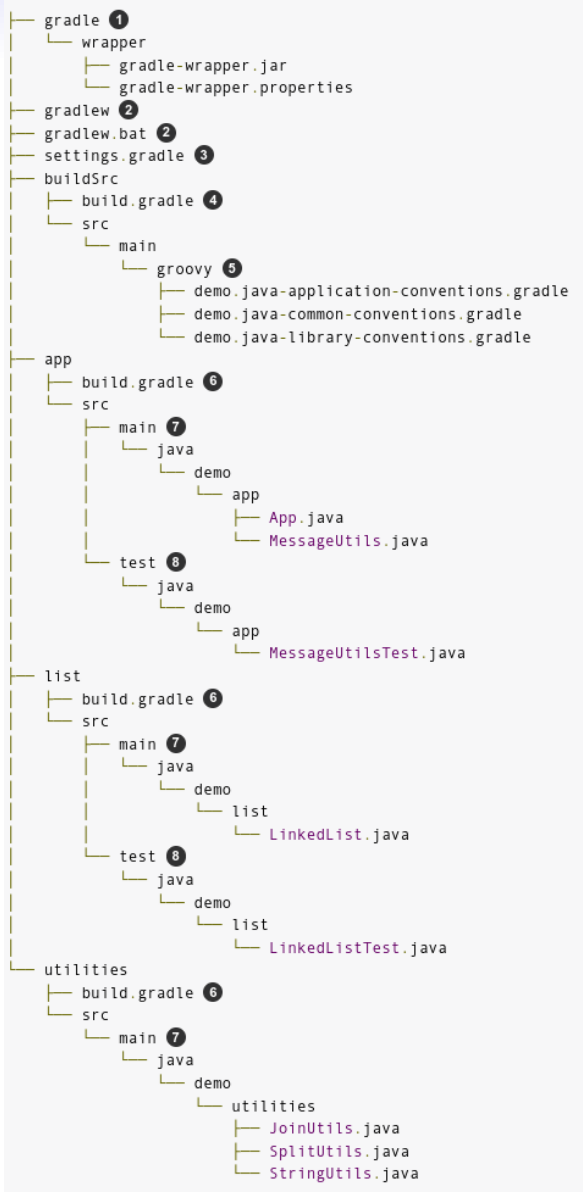
Plan

Avec plusieurs sous-projets

▸ exemple

```
$ gradle init

Select type of project to generate:
  1: basic
  2: application
  3: library
  4: Gradle plugin
Enter selection (default: basic) [1..4] 2

Split functionality across multiple subprojects?:
   1: no - only one application project
   2: yes - application and library projects
Enter selection (default: no - only one application project) [1..2] 2

Select implementation language:
  1: C++
  2: Groovy
  3: Java
  4: Kotlin
  5: Scala
  6: Swift
Enter selection (default: Java) [1..6] 3

Select build script DSL:
  1: Groovy
  2: Kotlin
Enter selection (default: Groovy) [1..2] 1

Select test framework:
  1: JUnit 4
  2: TestNG
  3: Spock
  4: JUnit Jupiter
Enter selection (default: JUnit 4) [1..4]

Project name (default: demo):
Source package (default: demo):


BUILD SUCCESSFUL
```

Organisation des répertoires



1. Generated folder for wrapper files
2. Gradle wrapper start scripts
3. Settings file to define build name and subprojects
4. Build script of *buildSrc* to configure dependencies of the build logic
5. Source folder for *convention plugins* written in Groovy or Kotlin DSL
6. Build script of the three subprojects - `app`, `list` and `utilities`
7. Java source folders in each of the subprojects
8. Java test source folders in the subprojects

settings.gradle

À la racine du projet, il permet de définir le nom du projet principal et les noms des sous-projets contenus

Kotlin Groovy

settings.gradle

```
rootProject.name = 'demo'
include('app', 'list', 'utilities')
```

- `rootProject.name` assigns a name to the build, which overrides the default behavior of naming the build after the directory it's in. It's recommended to set a fixed name as the folder might change if the project is shared - e.g. as root of a Git repository.
- `include("app", "list", "utilities")` defines that the build consists of three subprojects in the corresponding folders. More subprojects can be added by extending the list or adding more `include(…)` statements.

Partager la logique des build dans `buildSrc`

Un nouveau répertoire est créé ⇒ `buildSrc`

- Il permet de factoriser les parties communes aux sous-projets
- ⇒ dépendances
- ⇒ les paramétrages des plugins
- ⇒ les dépôts
- ...

`commons-conventions` partagées entre build

Kotlin | Groovy

buildSrc/src/main/groovy/demo.java-common-conventions.gradle

```
plugins {
    id 'java' (1)
}

repositories {
    mavenCentral() (2)
}

dependencies {
    constraints {
        implementation 'org.apache.commons:commons-text:1.10.0' (3)
    }

    testImplementation 'org.junit.jupiter:junit-jupiter:5.9.3' (4)

    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}

tasks.named('test') {
    useJUnitPlatform() (5)
}
```

The `java-common-conventions` defines some configuration that should be shared by all our Java project — independent of whether they represent a library or the actual application. First, we apply the Java Plugin (1) to have all functionality for building Java projects available. Then, we declare a repository — `mavenCentral()` — as source for external dependencies (2), define dependency constraints (3) as well as standard dependencies that are shared by all subprojects and set JUnit 5 as testing framework (4…). Other shared settings, like compiler flags or JVM version compatibilities, could also be set here.

Conventions spécifiques pour `library` et `application`

Kotlin Groovy

buildSrc/src/main/groovy/demo.java-library-conventions.gradle

```
plugins {
    id 'demo.java-common-conventions' (1)

    id 'java-library' (2)
}
```

Kotlin Groovy

buildSrc/src/main/groovy/demo.java-application-conventions.gradle

```
plugins {
    id 'demo.java-common-conventions' (1)

    id 'application' (2)
}
```

Both `java-library-conventions` and `java-application-conventions` apply the `java-common-conventions` plugin (1) so that the configuration performed there is shared by library and application projects alike. Next they apply the `java-library` or `application` plugin respectively (2) thus combining our common configuration logic with specifics for a library or application. While there is no more fine grained configuration in this example, library or application project specific build configuration can go into one of these convention plugin scripts.

Construction des plugins de conventions

Il faut un fichier de build spécifique pour construire les plugins de conventions : dans `buildSrc`
À ne pas confondre avec les build des sous-projets

buildSrc/build.gradle

```
plugins {
    id 'groovy-gradle-plugin' (1)
}

repositories {
    gradlePluginPortal() (2)
}
```

This file is setting the stage to build the convention plugins themselves. By applying one of the plugins for plugin development — `groovy-gradle-plugin` or `kotlin-dsl` — (1) we enable the support for writing convention plugins as build files in `buildSrc`. Which are the convention plugins we already inspected above. Furthermore, we add Gradle's plugin portal as repository (2), which gives us access to community plugins. To use a plugin it needs to be declared as dependency in the `dependencies {}` block.

Simplification des build des projets

app/build.gradle

```
plugins {
    id 'demo.java-application-conventions'
}

dependencies {
    implementation 'org.apache.commons:commons-text'
    implementation project(':utilities')
}

application {
    mainClass = 'demo.app.App' (1)
}
```

Kotlin Groovy

list/build.gradle

```
plugins {
    id 'demo.java-library-conventions'
}
```

Kotlin Groovy

utilities/build.gradle

```
plugins {
    id 'demo.java-library-conventions'
}

dependencies {
    api project(':list')
}
```

Plan

Spécifier la JVM toolchain

Un des intérêts de Gradle est d'assurer la reproductibilité des builds, quel que soit le système.
En particulier, il est possible de spécifier la version des outils (e.g. *Java toolchain*) qui doivent être utilisés.

build.gradle

GROOV

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(17)
    }
}
```

Executing the build (e.g. using `gradle check`) will now handle several things for you and others running your build:

1. Gradle configures all compile, test and javadoc tasks to use the defined toolchain.
2. Gradle detects locally installed toolchains.
3. Gradle chooses a toolchain matching the requirements (any Java 17 toolchain for the example above).
4. If no matching toolchain is found, Gradle can automatically download a matching one based on the configured toolchain download repositories.

Réglage de la toolchain en fonction du contexte

In the example below, we configure all java compilation tasks to use Java 8. Additionally, we introduce a new `Test` task that will run our unit tests using a JDK 17.

Kotlin Groovy

list/build.gradle

```
tasks.withType(JavaCompile).configureEach {
    javaCompiler = javaToolchains.compilerFor {
        languageVersion = JavaLanguageVersion.of(8)
    }
}

task('testsOn17', type: Test) {
    javaLauncher = javaToolchains.launcherFor {
        languageVersion = JavaLanguageVersion.of(17)
    }
}
```

Réglage de la toolchain en fonction du contexte

In addition, in the `application` subproject, we add another Java execution task to run our application with JDK 17.

Kotlin Groovy

application/build.gradle

```
task('runOn17', type: JavaExec) {
    javaLauncher = javaToolchains.launcherFor {
        languageVersion = JavaLanguageVersion.of(17)
    }

    classpath = sourceSets.main.runtimeClasspath
    mainClass = application.mainClass
}
```

Depending on the task, a JRE might be enough while for other tasks (e.g. compilation), a JDK is required. By default, Gradle prefers installed JDKs over JREs if they can satisfy the requirements.

Toolchains tool providers can be obtained from the `javaToolchains` extension.

Three tools are available:

- A `JavaCompiler` which is the tool used by the `JavaCompile` task
- A `JavaLauncher` which is the tool used by the `JavaExec` or `Test` tasks
- A `JavadocTool` which is the tool used by the `Javadoc` task

Conclusion

Un développement de qualité nécessite une gestion avancée du code source et de ses dépendances

⇒ Utiliser **un moteur de production logiciel est incontournable** !

Gradle

- Principe Zéro-conf / Rapide (cache) / extensible
- Syntaxe simple avec Groovy ou Kotlin
- Gestion de projet complexe avec sous-projets
- Fonctionnement par plugins : écosystème très important
- **Reproductibilité des builds**
- **Reproductibilité de l'exécution**
- Et n'oubliez pas de soigner la rédaction de vos builds !

▸ gradle.org